MySQL Performance Schema

Abstract

This is the MySQL Performance Schema extract from the MySQL 5.7 Reference Manual.

For legal information, see the Legal Notices.

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

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Licensing information—MySQL 5.7. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL 5.7, see this document 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 5.7, see this document for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Licensing information—MySQL Cluster. This product may include third-party software, used under license. If you are using a *Community* release of MySQL Cluster NDB 7.5, see this document for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2016-09-27 (revision: 49214)

Table of Contents

Preface and Legal Notices

This is the MySQL Performance Schema extract from the MySQL 5.7 Reference Manual.

Legal Notices

Copyright © 1997, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other 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.

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

Intel and Intel Xeon 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, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

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.

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

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.

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.

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` database are views or temporary tables that use no persistent on-disk storage.
- 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.
- It is easy to add new instrumentation points.
- Instrumentation is versioned. If the instrumentation implementation changes, previously instrumented code will continue 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 as of MySQL 5.7.7. 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 12, *Using the Performance Schema to Diagnose Problems*.

For the Performance Schema to be available, support for it must have been configured when MySQL was built. You can verify whether this is the case by checking the server's help output. If the Performance Schema is available, the output will mention several variables with names that begin with `performance_schema`:

```
shell> mysqld --verbose --help
...
  --performance_schema
                      Enable the performance schema.
  --performance_schema_events_waits_history_long_size=#
                      Number of rows in events_waits_history_long.
...
```

If such variables do not appear in the output, your server has not been built to support the Performance Schema. In this case, see Chapter 3, *Performance Schema Configuration*.

Assuming that the Performance Schema is available, it 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 your `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. If this engine is available (which you should already have checked earlier), you should see it listed with a `SUPPORT` value of `YES` 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;
```

Many examples in this chapter assume performance_schema as the default database.

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                                         |
| setup_timers                                          |
| 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 is expected to increase 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 setup_timers\G
*************************** 1. row ***************************
       Table: setup_timers
Create Table: CREATE TABLE `setup_timers` (
  `NAME` varchar(64) NOT NULL,
  `TIMER_NAME` enum('CYCLE','NANOSECOND','MICROSECOND','MILLISECOND','TICK')
   NOT NULL
) ENGINE=PERFORMANCE_SCHEMA DEFAULT CHARSET=utf8
```

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 8, *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 setup_instruments SET ENABLED = 'YES', TIMED = 'YES';
Query OK, 560 rows affected (0.04 sec)
mysql> UPDATE 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 events_waits_current\G
*************************** 1. row ***************************
            THREAD_ID: 0
             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
          OBJECT_TYPE: NULL
OBJECT_INSTANCE_BEGIN: 142270668
     NESTING_EVENT_ID: 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 3.3.1, “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 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 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 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 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, to see which event timers are selected, query the `setup_timers` tables:

```
mysql> SELECT * FROM setup_timers;
+-------------+-------------+
| NAME        | TIMER_NAME  |
+-------------+-------------+
| idle        | MICROSECOND |
| wait        | CYCLE       |
| stage       | NANOSECOND  |
| statement   | NANOSECOND  |
| transaction | NANOSECOND  |
+-------------+-------------+
```

`setup_instruments` lists the set of instruments for which events can be collected and shows which of them are enabled:

```
mysql> SELECT * FROM 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   |
...
| 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 5, *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 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 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      | NO      |
| events_transactions_history      | NO      |
| 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 3.3.2, “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 3.3.1, “Performance Schema Event Timing”.

Chapter 3 Performance Schema Configuration

Table of Contents

To use the MySQL Performance Schema, these configuration considerations apply:

- The Performance Schema must be configured into MySQL Server at build time to make it available. Performance Schema support is included in binary MySQL distributions. If you are building from source, you must ensure that it is configured into the build as described in Section 3.1, “Performance Schema Build Configuration”.
- The Performance Schema must be enabled at server startup to enable event collection to occur. Specific Performance Schema features can be enabled at server startup or at runtime to control which types of event collection occur. See Section 3.2, “Performance Schema Startup Configuration”, Section 3.3, “Performance Schema Runtime Configuration”, and Section 3.3.2, “Performance Schema Event Filtering”.

3.1 Performance Schema Build Configuration

For the Performance Schema to be available, it must be configured into the MySQL server at build time. Binary MySQL distributions provided by Oracle Corporation are configured to support the Performance Schema. If you use a binary MySQL distribution from another provider, check with the provider whether the distribution has been appropriately configured.

If you build MySQL from a source distribution, enable the Performance Schema by running `CMake` with the `WITH_PERFSCHEMA_STORAGE_ENGINE` option enabled:

```
shell> cmake . -DWITH_PERFSCHEMA_STORAGE_ENGINE=1
```

Configuring MySQL with the `-DWITHOUT_PERFSCHEMA_STORAGE_ENGINE=1` option prevents inclusion of the Performance Schema, so if you want it included, do not use this option. See MySQL Source-Configuration Options.

As of MySQL 5.7.3, it is also possible to enable the Performance Schema but exclude certain parts of the instrumentation. For example, to enable the Performance Schema but exclude stage and statement instrumentation, do this:

```
shell> cmake . -DWITH_PERFSCHEMA_STORAGE_ENGINE=1 \
        -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 may not have all the current tables), run `mysql_upgrade` after starting the server to ensure that the `performance_schema` database exists with all current tables. Then restart the server. One indication that you need to do this 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 verify whether a server was built with Performance Schema support, check its help output. If the Performance Schema is available, the output will mention several variables with names that begin with `performance_schema`:

```
shell> mysqld --verbose --help
...
  --performance_schema
                      Enable the performance schema.
  --performance_schema_events_waits_history_long_size=#
                      Number of rows in events_waits_history_long.
...
```

You can also connect to the server and look for a line that names the `PERFORMANCE_SCHEMA` storage engine in the output from `SHOW ENGINES`:

```
mysql> SHOW ENGINES\G
...
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
...
```

If the Performance Schema was not configured into the server at build time, no row for `PERFORMANCE_SCHEMA` will appear in the output from `SHOW ENGINES`. You might see `performance_schema` listed in the output from `SHOW DATABASES`, but it will have no tables and you will not be able to use it.

A line for `PERFORMANCE_SCHEMA` in the `SHOW ENGINES` output 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.

3.2 Performance Schema Startup Configuration

Assuming that the Performance Schema is available, it 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 your `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 3.3.4, “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. Before MySQL 5.7.6, it may be necessary to tune the values of Performance Schema system variables to find the number of instances that balances insufficient instrumentation against excessive memory consumption. As of MySQL 5.7.6, 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. As of MySQL 5.7.6, 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 10, *Performance Schema System Variables*.

For each autosized parameter that is not set at server startup (or is set to −1), 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.

3.3 Performance Schema Runtime Configuration

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_timers      |
+-------------------+
```

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 8.2, “Performance Schema Setup Tables”.

To see which event timers are selected, query the `setup_timers` tables:

```
mysql> SELECT * FROM setup_timers;
+-------------+-------------+
| NAME        | TIMER_NAME  |
+-------------+-------------+
| idle        | MICROSECOND |
| wait        | CYCLE       |
| stage       | NANOSECOND  |
| statement   | NANOSECOND  |
| transaction | NANOSECOND  |
+-------------+-------------+
```

The `NAME` value indicates the type of instrument to which the timer applies, and `TIMER_NAME` indicates which timer applies to those instruments. The timer applies to instruments where their name begins with a component matching the `NAME` value.

To change the timer, update the `NAME` value. For example, to use the `NANOSECOND` timer for the `wait` timer:

```
mysql> UPDATE setup_timers SET TIMER_NAME = 'NANOSECOND'
    -> WHERE NAME = 'wait';
mysql> SELECT * FROM setup_timers;
+-------------+-------------+
| NAME        | TIMER_NAME  |
+-------------+-------------+
| idle        | MICROSECOND |
| wait        | NANOSECOND  |
| stage       | NANOSECOND  |
```

```
| statement   | NANOSECOND  |
| transaction | NANOSECOND  |
+-------------+-------------+
```

For discussion of timers, see Section 3.3.1, “Performance Schema Event Timing”.

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 3.3.2, “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=file_name` option. 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` argument when you start the server.

3.3.1 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

Two Performance Schema tables provide timer information:

- `performance_timers` lists the available timers and their characteristics.
- `setup_timers` indicates which timers are used for which instruments.

Each timer row in `setup_timers` must refer to one of the timers listed in `performance_timers`.

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_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 |
| TICK        |             105 |                1 |           2416 |
+-------------+-----------------+------------------+----------------+
```

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. The timers in `setup_timers` that you can use are those that do not have `NULL` in the other columns. If the values associated with a given timer name are `NULL`, that timer is not supported on your platform.
- `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. For `TICK`, the frequency may vary by platform (for example, some use 100 ticks/second, others 1000 ticks/second).

- `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.

To see which timers are in effect or to change timers, access the `setup_timers` table:

```
mysql> SELECT * FROM setup_timers;
+-------------+-------------+
| NAME        | TIMER_NAME  |
+-------------+-------------+
| idle        | MICROSECOND |
| wait        | CYCLE       |
| stage       | NANOSECOND  |
| statement   | NANOSECOND  |
| transaction | NANOSECOND  |
+-------------+-------------+
mysql> UPDATE setup_timers SET TIMER_NAME = 'MICROSECOND'
    -> WHERE NAME = 'idle';
mysql> SELECT * FROM setup_timers;
+-------------+-------------+
| NAME        | TIMER_NAME  |
+-------------+-------------+
| idle        | MICROSECOND |
| wait        | CYCLE       |
| stage       | NANOSECOND  |
| statement   | NANOSECOND  |
| transaction | NANOSECOND  |
+-------------+-------------+
```

By default, the Performance Schema uses the best timer available for each instrument type, but you can select a different one.

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, OS X, 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 the sum, minimum, maximum, and average time values in summary tables.

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.

Modifications to the `setup_timers` table affect monitoring immediately. Events already in progress may use the original timer for the begin time and the new timer for the end time. To avoid unpredictable results after you make timer changes, use `TRUNCATE TABLE` to reset Performance Schema statistics.

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.

Before MySQL 5.7.8, while a wait, stage, statement, or transaction event is executing, the respective current-event tables display the event with `TIMER_START` populated, but with `TIMER_END` and `TIMER_WAIT` set to `NULL`:

```
events_waits_current
events_stages_current
events_statements_current
events_transactions_current
```

As of MySQL 5.7.8, current-event timing provides more information. To make it possible to determine how how long a not-yet-completed event has been running, the timer columns are set as follows:

- `TIMER_START` is populated (unchanged from previous behavior)
- `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 relevent consumers are enabled.

3.3.2 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 * FROM 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 3.3.3, “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 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       | NO      |
| events_transactions_history       | NO      |
| 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 4, *Performance Schema Queries*.

3.3.3 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 will not be sent to any destination (that is, will not be consumed), the Performance Schema does not produce it.

Modifications to any of these tables affect monitoring immediately, with some exceptions:

- Modifications to some instruments in the `setup_instruments` table 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. This restriction is lifted as of MySQL 5.7.12.
- 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 to clear aggregate information for previously collected events. Except for `events_statements_summary_by_digest` and the memory summary tables, the effect of `TRUNCATE TABLE` for summary tables is to reset the summary columns to 0 or `NULL`, not to remove rows.

The following sections describe how to use specific tables to control Performance Schema pre-filtering.

3.3.3.1 Pre-Filtering by Instrument

The `setup_instruments` table lists the available instruments:

```
mysql> SELECT * FROM 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   |
...
| 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 3.3.1, “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 3.3.3, “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 3.3.4, “Naming Instruments or Consumers for Filtering Operations”.

- Disable all instruments:

  ```
  mysql> UPDATE setup_instruments SET ENABLED = 'NO';
  ```

  Now no events will be collected.

- Disable all file instruments, adding them to the current set of disabled instruments:

  ```
  mysql> UPDATE setup_instruments SET ENABLED = 'NO'
      -> WHERE NAME LIKE 'wait/io/file/%';
  ```

- Disable only file instruments, enable all other instruments:

  ```
  mysql> UPDATE setup_instruments
      -> SET ENABLED = IF(NAME LIKE 'wait/io/file/%', 'NO', 'YES');
  ```

- Enable all but those instruments in the `mysys` library:

  ```
  mysql> UPDATE setup_instruments
      -> SET ENABLED = CASE WHEN NAME LIKE '%/mysys/%' THEN 'YES' ELSE 'NO' END;
  ```

- Disable a specific instrument:

  ```
  mysql> UPDATE setup_instruments SET ENABLED = 'NO'
      -> WHERE NAME = 'wait/synch/mutex/mysys/TMPDIR_mutex';
  ```

- To toggle the state of an instrument, “flip” its `ENABLED` value:

  ```
  mysql> UPDATE setup_instruments
      -> SET ENABLED = IF(ENABLED = 'YES', 'NO', 'YES')
  ```

```
    -> WHERE NAME = 'wait/synch/mutex/mysys/TMPDIR_mutex';
```

- Disable timing for all events:

```
mysql> UPDATE setup_instruments SET TIMED = 'NO';
```

3.3.3.2 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 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 3.3.1, “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.

3.3.3.3 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 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 (combination of host and user), 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.

Before MySQL 5.7.6, there is no `ENABLED` column. The `INSTRUMENTED` value for the thread becomes `YES` if any row matches, `NO` otherwise.

Before MySQL 5.7.8, there is no `HISTORY` column. The Performance Schema logs historical events either for all threads or no threads, depending on which history consumers are enabled or disabled.

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

3.3.3.4 Pre-Filtering by Consumer

The setup_consumers table lists the available consumer types and which are enabled:

```
mysql> SELECT * FROM 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    | NO      |
| events_transactions_history    | NO      |
| 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:

```
mysql> UPDATE 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 3.3.3.5, “Example Consumer Configurations”.

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_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

This 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 will be `NULL`).

3.3.3.5 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 and the Performance Schema will execute less code on your behalf and you will have 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 3.3.3.4, “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

Server configuration state:

```
mysql> SELECT * FROM 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 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`
- Table `setup_timers`

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 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 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      | YES     |
| events_statements_history_long | NO      |
| events_transactions_current    | YES     |
```

```
| events_transactions_history      | YES     |
| 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 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 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        | 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_long`, where *xxx* is `waits`, `stages`, `statements`, `transactions`

This configuration collects event history per thread and globally:

```
mysql> SELECT * FROM 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`

3.3.4 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:

```
mysql> UPDATE setup_instruments
    -> SET ENABLED = 'NO'
    -> WHERE NAME = 'wait/synch/mutex/myisammrg/MYRG_INFO::mutex';
mysql> UPDATE 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:

```
mysql> UPDATE setup_instruments
```

```
    -> SET ENABLED = 'NO'
    -> WHERE NAME LIKE 'wait/synch/mutex/%';
mysql> UPDATE 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/%'` will match other instruments that have a component of `'/file/'` anywhere in the name. Even less suitable is the pattern `'%file%'` because it will match instruments with `'file'` anywhere in the name, such as `wait/synch/mutex/sql/LOCK_des_key_file`.

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

```
mysql> SELECT NAME FROM setup_instruments WHERE NAME LIKE 'pattern';
mysql> SELECT NAME FROM setup_consumers WHERE NAME LIKE 'pattern';
```

For information about the types of names that are supported, see Chapter 5, *Performance Schema Instrument Naming Conventions*.

3.3.5 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 * FROM setup_instruments WHERE NAME LIKE 'wait/io/file/innodb/%';
+-------------------------------------+---------+-------+
| NAME                                | ENABLED | TIMED |
+-------------------------------------+---------+-------+
| 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   |
+-------------------------------------+---------+-------+
```

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 3.3.3, “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 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 |
+-----------+-----------------+
```

Chapter 5 Performance Schema Instrument Naming Conventions

An instrument name consists of a sequence of components 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
```

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

The interpretation of a given component in a name depends on the components 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 component of an instrument prefix indicates the type of instrument. This component also determines which event timer in the `setup_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 component (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`.

Top-Level Instrument Components

- `idle`: An instrumented idle event. This instrument has no further components.
- `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 components.
- `wait`: An instrumented wait event.

Idle Instrument Components

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 8.3.5, “The socket_instances Table”.

Memory Instrument Components

Most memory instrumentation is disabled by default, and can be enabled or disabled dynamically 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. The 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 Components

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 Components

- 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 8.6, “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.

Wait Instrument Components

- 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 (disabled by default).

- `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/rwlock`

    A 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` were introduced in MySQL 5.7 to optimize concurrency and improve scalability for read-write workloads.

Chapter 6 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 will not be 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 will be 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 will increase 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 will miss 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 will be allocated for file-related classes, instances, or handles, and all file events will be 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 Syntax.

Chapter 7 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.

Most tables in the `performance_schema` database are read only and cannot be modified. 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_`.

`TRUNCATE TABLE` can also be used with summary tables, but except for `events_statements_summary_by_digest` and the memory summary tables, the effect is to reset the summary columns to 0 or `NULL`, not to remove rows.

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.

Chapter 8 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.

8.1 Performance Schema Table Index

The following table lists each Performance Schema table and provides a short description of each one.

Table 8.1 Performance Schema Tables

Table Name	Description
`accounts`	Connection statistics per client account

Table Name	Description
cond_instances	synchronization object instances
events_stages_current	Current stage events
events_stages_history	Most recent stage events for each 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_history	Most recent statement events for each 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 for each thread
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 for each 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

Table Name	Description
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
global_status	Global status variables
global_variables	Global system variables
host_cache	Information from the internal host cache
hosts	Connection statistics per client host name
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
objects_summary_global_by_type	Object summaries
performance_timers	Which event timers are available
prepared_statements_instances	Prepared statement instances and statistics
replication_connection_configuration	Configuration parameters for connecting to the master
replication_connection_status	Current status of the connection to the master
replication_applier_configuration	Configuration parameters for the transaction applier on the slave
replication_applier_status	Current status of the transaction applier on the slave
replication_applier_status_by_coordinator	SQL or coordinator thread applier status
replication_applier_status_by_worker	Worker thread applier status (empty unless slave is multi-threaded)
rwlock_instances	Lock synchronization object instances
session_account_connect_attrs	Connection attributes per for the 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

Table Name	Description
setup_instruments	Classes of instrumented objects for which events can be collected
setup_objects	Which objects should be monitored
setup_timers	Current event timer
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_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
users	Connection statistics per client user name
user_variables_by_thread	User-defined variables per thread
variables_by_thread	Session system variables per session

8.2 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_timers: The current event timer

8.2.1 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 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 (combination of host and user). 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 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 3.3.3.3, “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`.

  This column was added in MySQL 5.7.6. For earlier versions in which it is not present, the Performance Schema enables instrumentation only for foreground threads matched by some row in the table; instrumentation is implicitly disabled for nonmatching threads.

- `HISTORY`

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

  This column was added in MySQL 5.7.8. For earlier versions in which it is not present, the Performance Schema logs historical events either for all threads or no threads, depending on which history consumers are enabled or disabled.

8.2.2 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 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    | NO      |
| events_transactions_history    | NO      |
| 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 3.3.3.4, “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.

8.2.3 The setup_instruments Table

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

```
mysql> SELECT * FROM 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   |
...
| 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   |
...
```

Each instrument added to the source code provides a row for this 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 `*_instances` tables.

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 3.3.3, “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 5, *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`. This column can be modified. A disabled instrument produces no events.

- `TIMED`

  Whether the instrument is timed. This column can be modified.

  For memory instruments, the `TIMED` column in `setup_instruments` is ignored because memory operations are not timed.

  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.

8.2.4 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 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 3.3.3, “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). Before MySQL 5.7.2, the value is always `'TABLE'`.

  `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.

8.2.5 The setup_timers Table

The `setup_timers` table shows the currently selected event timers:

```
mysql> SELECT * FROM setup_timers;
+-------------+-------------+
| NAME        | TIMER_NAME  |
+-------------+-------------+
| idle        | MICROSECOND |
| wait        | CYCLE       |
| stage       | NANOSECOND  |
| statement   | NANOSECOND  |
| transaction | NANOSECOND  |
+-------------+-------------+
```

The `setup_timers.TIMER_NAME` value can be changed to select a different timer. The value can be any of the values in the `performance_timers.TIMER_NAME` column. For an explanation of how event timing occurs, see Section 3.3.1, “Performance Schema Event Timing”.

Modifications to the `setup_timers` table affect monitoring immediately. Events already in progress may use the original timer for the begin time and the new timer for the end time. To avoid unpredictable results after you make timer changes, use `TRUNCATE TABLE` to reset Performance Schema statistics.

The `setup_timers` table has these columns:

- `NAME`

  The type of instrument the timer is used for.

- `TIMER_NAME`

  The timer that applies to the instrument type. This column can be modified.

8.3 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 5, *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 12, *Using the Performance Schema to Diagnose Problems*

8.3.1 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, will cause 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.

8.3.2 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 will not be 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` will be 0. To list all the files currently opened by the server, use `WHERE OPEN_COUNT > 0`.

8.3.3 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 will compete against each other, so that the first query to obtain a lock on the mutex will cause 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`.

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

8.3.4 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 was introduced in MySQL 5.7 to optimize concurrency and improve 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.

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.

8.3.5 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 8.15.9, “Socket Summary Tables”).

```
mysql> SELECT * FROM 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 5, *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 `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)

8.4 Performance Schema Wait Event Tables

These tables store wait events:

- `events_waits_current`: Current wait events
- `events_waits_history`: The most recent wait events for each thread
- `events_waits_history_long`: The most recent wait events overall

The following sections describe those tables. There are also summary tables that aggregate information about wait events; see Section 8.15.1, “Event Wait Summary Tables”.

Wait Event Configuration

To enable collection of wait events, enable the relevant instruments and consumers.

The `setup_instruments` table contains instruments with names that begin with `wait`. For example:

```
mysql> SELECT * FROM setup_instruments
    -> WHERE NAME LIKE 'wait/io/file/innodb%';
+---------------------------------------+---------+-------+
| NAME                                  | ENABLED | TIMED |
+---------------------------------------+---------+-------+
| 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   |
+---------------------------------------+---------+-------+
mysql> SELECT * FROM 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    |
+----------------------------------------+---------+-------+
```

To modify collection of wait events, change the `ENABLED` and `TIMING` columns of the relevant instruments. For example:

```
mysql> UPDATE setup_instruments SET ENABLED = 'YES', TIMED = 'YES'
    -> WHERE NAME LIKE 'wait/io/socket/sql/%';
```

The `setup_consumers` table contains consumer values with names corresponding to the current and recent wait event table names. These consumers may be used to filter collection of wait events. The wait consumers are disabled by default:

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

To enable all wait consumers, do this:

```
mysql> UPDATE setup_consumers SET ENABLED = 'YES'
    -> WHERE NAME LIKE '%waits%';
```

The `setup_timers` table contains a row with a `NAME` value of `wait` that indicates the unit for wait event timing. The default unit is `CYCLE`.

```
mysql> SELECT * FROM setup_timers WHERE NAME = 'wait';
+------+------------+
| NAME | TIMER_NAME |
+------+------------+
| wait | CYCLE      |
+------+------------+
```

To change the timing unit, modify the `TIMER_NAME` value:

```
mysql> UPDATE setup_timers SET TIMER_NAME = 'NANOSECOND'
    -> WHERE NAME = 'wait';
```

For additional information about configuring event collection, see Chapter 3, *Performance Schema Configuration*.

8.4.1 The events_waits_current Table

The `events_waits_current` table contains current wait events, one row per thread showing the current status of the thread's most recent monitored wait event.

The `events_waits_current` table can be truncated with `TRUNCATE TABLE`.

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, up to a fixed number of rows.

For information about configuration of wait event collection, see Section 8.4, “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 5, *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` and `TIMER_WAIT` are `NULL` before MySQL 5.7.8. As of 5.7.8, `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 3.3.1, “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` is `NULL` before MySQL 5.7.5. For table I/O events as of 5.7.5, this column 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.

8.4.2 The events_waits_history Table

The `events_waits_history` table contains the most recent *`N`* wait events per thread. The value of *`N`* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_waits_history_size` system variable at server startup. Wait events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full.

The `events_waits_history` table has the same structure as `events_waits_current`. See Section 8.4.1, “The events_waits_current Table”.

The `events_waits_history` table can be truncated with `TRUNCATE TABLE`.

For information about configuration of wait event collection, see Section 8.4, “Performance Schema Wait Event Tables”.

8.4.3 The events_waits_history_long Table

The `events_waits_history_long` table contains the most recent *`N`* wait events. The value of *`N`* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_waits_history_long_size` system variable at server startup. Wait events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full. When a thread ends, its rows are removed from the table.

The `events_waits_history_long` table has the same structure as `events_waits_current`. See Section 8.4.1, “The events_waits_current Table”.

The `events_waits_history_long` table can be truncated with `TRUNCATE TABLE`.

For information about configuration of wait event collection, see Section 8.4, “Performance Schema Wait Event Tables”.

8.5 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`: Current stage events
- `events_stages_history`: The most recent stage events for each thread
- `events_stages_history_long`: The most recent stage events overall

The following sections describe those tables. There are also summary tables that aggregate information about stage events; see Section 8.15.2, “Stage Summary Tables”.

Stage Event Configuration

To enable collection of stage events, enable the relevant instruments and consumers.

The `setup_instruments` table contains instruments with names that begin with `stage`. Other than those instruments that provide statement progress information, these instruments are disabled by default. For example:

```
mysql> SELECT * FROM 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/checking privileges on cached query     | NO      | NO    |
| stage/sql/checking query cache for query          | 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    |
+----------------------------------------------------+---------+-------+
```

As of MySQL 5.7.7, stage event instruments that provide statement progress information now are enabled and timed by default:

```
mysql> SELECT * FROM setup_instruments WHERE
    -> ENABLED='YES' AND NAME LIKE "stage/%";
+----------------------------------------------------+---------+-------+
| NAME                                               | ENABLED | TIMED |
+----------------------------------------------------+---------+-------+
| stage/sql/copy to tmp table                        | 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   |
+----------------------------------------------------+---------+-------+
```

To modify collection of stage events, change the `ENABLED` and `TIMING` columns of the relevant instruments. For example:

```
mysql> UPDATE setup_instruments SET ENABLED = 'YES', TIMED = 'YES'
    -> WHERE NAME = 'stage/sql/altering table';
```

The `setup_consumers` table contains consumer values with names corresponding to the current and recent stage event table names. These consumers may be used to filter collection of stage events. The stage consumers are disabled by default:

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

To enable all stage consumers, do this:

```
mysql> UPDATE setup_consumers SET ENABLED = 'YES'
    -> WHERE NAME LIKE '%stages%';
```

The `setup_timers` table contains a row with a `NAME` value of `stage` that indicates the unit for stage event timing. The default unit is `NANOSECOND`.

```
mysql> SELECT * FROM setup_timers WHERE NAME = 'stage';
+-------+------------+
| NAME  | TIMER_NAME |
+-------+------------+
| stage | NANOSECOND |
```

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

To change the timing unit, modify the `TIMER_NAME` value:

```
mysql> UPDATE setup_timers SET TIMER_NAME = 'MICROSECOND'
    -> WHERE NAME = 'stage';
```

For additional information about configuring event collection, see Chapter 3, *Performance Schema Configuration*.

Stage Event Progress Information

As of MySQL 5.7.5, 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 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:

```
mysql> UPDATE setup_instruments SET ENABLED='YES'
    -> WHERE NAME='stage/sql/copy to tmp table';
mysql> UPDATE 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.

8.5.1 The events_stages_current Table

The `events_stages_current` table contains current stage events, one row per thread showing the current status of the thread's most recent monitored stage event.

The `events_stages_current` table can be truncated with `TRUNCATE TABLE`.

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, up to a fixed number of rows.

For information about configuration of stage event collection, see Section 8.5, “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 5, *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` and `TIMER_WAIT` are `NULL` before MySQL 5.7.8. As of 5.7.8, `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 3.3.1, “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.

  These columns were added in MySQL 5.7.5.

- `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`.

8.5.2 The events_stages_history Table

The `events_stages_history` table contains the most recent *`N`* stage events per thread. The value of *`N`* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_stages_history_size` system variable at server startup. Stage events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full.

The `events_stages_history` table has the same structure as `events_stages_current`. See Section 8.5.1, “The events_stages_current Table”.

The `events_stages_history` table can be truncated with `TRUNCATE TABLE`.

For information about configuration of stage event collection, see Section 8.5, “Performance Schema Stage Event Tables”.

8.5.3 The events_stages_history_long Table

The `events_stages_history_long` table contains the most recent *`N`* stage events. The value of *`N`* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_stages_history_long_size` system variable at server startup. Stage events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full. When a thread ends, its rows are removed from the table.

The `events_stages_history_long` table has the same structure as `events_stages_current`. See Section 8.5.1, “The events_stages_current Table”.

The `events_stages_history_long` table can be truncated with `TRUNCATE TABLE`.

For information about configuration of stage event collection, see Section 8.5, “Performance Schema Stage Event Tables”.

8.6 Performance Schema Statement Event Tables

The Performance Schema instruments statement execution. Statement events occur at a high level of 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`: Current statement events
- `events_statements_history`: The most recent statement events for each thread
- `events_statements_history_long`: The most recent statement events overall
- `prepared_statements_instances`: Prepared statement instances and statistics (added in MySQL 5.7.4)

The following sections describe those tables. There are also summary tables that aggregate information about statement events; see Section 8.15.3, “Statement Summary Tables”.

Statement Event Configuration

To enable collection of statement events, enable the relevant instruments and consumers.

The `setup_instruments` table contains instruments with names that begin with `statement`. These instruments are enabled by default:

```
mysql> SELECT * FROM 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   |
+----------------------------------------+---------+-------+
```

To modify collection of statement events, change the `ENABLED` and `TIMING` columns of the relevant instruments. For example:

```
mysql> UPDATE setup_instruments SET ENABLED = 'NO'
    -> WHERE NAME LIKE 'statement/com/%';
```

The `setup_consumers` table contains consumer values with names corresponding to the current and recent statement event table names, and the statement digest consumer. These consumers may be used to filter collection of statement events and statement digesting. `events_statements_current`, `events_statements_history`, and `statements_digest` are enabled by default (before MySQL 5.7.5, `events_statements_history` is disabled by default):

```
mysql> SELECT * FROM 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 enable all statement consumers, do this:

```
mysql> UPDATE setup_consumers SET ENABLED = 'YES'
    -> WHERE NAME LIKE '%statements%';
```

The `setup_timers` table contains a row with a `NAME` value of `statement` that indicates the unit for statement event timing. The default unit is `NANOSECOND`.

```
mysql> SELECT * FROM setup_timers WHERE NAME = 'statement';
+-----------+-------------+
| NAME      | TIMER_NAME |
+-----------+-------------+
| statement | NANOSECOND |
+-----------+-------------+
```

To change the timing unit, modify the `TIMER_NAME` value:

```
mysql> UPDATE setup_timers SET TIMER_NAME = 'MICROSECOND'
    -> WHERE NAME = 'statement';
```

For additional information about configuring event collection, see Chapter 3, *Performance Schema 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. Before MySQL 5.7.2, monitoring occurs only for top-level statements. Statements within stored programs and subqueries are not seen separately. As of 5.7.2, 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 replication slave (as of MySQL 5.7.2)
- As an event from the Event Scheduler (as of MySQL 5.7.2)

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 replication slave:

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 slave 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 abtract `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.

The abstract instrument names in the preceding discussion are as of MySQL 5.7.3. In earlier 5.7 versions, there was some renaming before those names were settled on:

- `statement/abstract/new_packet` was `statement/com/` before MySQL 5.7.3.
- `statement/abstract/Query` was `statement/com/Query` before MySQL 5.7.3.
- `statement/abstract/relay_log` was `statement/rpl/relay_log` in MySQL 5.7.2 and did not exist before that.

8.6.1 The events_statements_current Table

The `events_statements_current` table contains current statement events, one row per thread showing the current status of the thread's most recent monitored statement event.

The `events_statements_current` table can be truncated with `TRUNCATE TABLE`.

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, up to a fixed number of rows.

For information about configuration of statement event collection, see Section 8.6, “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 5, *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 8.6, “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` and `TIMER_WAIT` are `NULL` before MySQL 5.7.8. As of 5.7.8, `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 3.3.1, “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`.

As of MySQL 5.7.6, the maximum number of bytes to display can be changed by changing the `performance_schema_max_sql_text_length` system variable at server startup. Before 5.7.6, the maximum is fixed at 1024.

- `DIGEST`

  The statement digest MD5 value as a string of 32 hexadecimal characters, or `NULL` if the `statement_digest` consumer is `no`. For more information about statement digesting, see Performance Schema Statement Digests.

- `DIGEST_TEXT`

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

  The `performance_schema_max_digest_length` system variable determines the maximum number of bytes available for computing statement digests. However, the display length of statement digests may be longer than the available buffer size due to encoding of statement components 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.

  This variable was added in MySQL 5.7.8. In MySQL 5.7.6 and 5.7.7, use `max_digest_length` instead. Before 5.7.6, the value cannot be changed.

- `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 read from storage engines during statement execution.

- `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`

  Before MySQL 5.7.2, only `NESTING_EVENT_ID` and `NESTING_EVENT_TYPE` exist and are always `NULL`.

  As of MySQL 5.7.2, all three columns exist and 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 = NULL
NESTING_EVENT_TYPE = NULL
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
```

8.6.2 The events_statements_history Table

The `events_statements_history` table contains the most recent *`N`* statement events per thread. The value of *`N`* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_statements_history_size` system variable at server startup. Statement events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full.

The `events_statements_history` table has the same structure as `events_statements_current`. See Section 8.6.1, “The events_statements_current Table”.

The `events_statements_history` table can be truncated with `TRUNCATE TABLE`.

For information about configuration of statement event collection, see Section 8.6, “Performance Schema Statement Event Tables”.

8.6.3 The events_statements_history_long Table

The `events_statements_history_long` table contains the most recent *`N`* statement events. 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. Statement events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full. When a thread ends, its rows are removed from the table.

The `events_statements_history_long` table has the same structure as `events_statements_current`. See Section 8.6.1, “The events_statements_current Table”.

The `events_statements_history_long` table can be truncated with `TRUNCATE TABLE`.

For information about configuration of statement event collection, see Section 8.6, “Performance Schema Statement Event Tables”.

8.6.4 The prepared_statements_instances Table

As of MySQL 5.7.4, 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`
`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;
  ```

- 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 8.15.3, “Statement Summary Tables”).

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

8.7 Performance Schema Transaction Tables

As of MySQL 5.7.3, 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`: Current transaction events
- `events_transactions_history`: The most recent transaction events for each thread
- `events_transactions_history_long`: The most recent transaction events overall

The following sections describe those tables. There are also summary tables that aggregate information about transaction events; see Section 8.15.4, “Transaction Summary Tables”.

Transaction Event Configuration

To enable collection of transaction events, enable the relevant instruments and consumers.

The `setup_instruments` table contains an instrument named `transaction`. This instrument is disabled by default:

```
mysql> SELECT * FROM setup_instruments WHERE NAME = 'transaction';
+-------------+---------+-------+
| NAME        | ENABLED | TIMED |
+-------------+---------+-------+
| transaction | NO      | NO    |
+-------------+---------+-------+
```

To enable collection of transaction events, including timing information, do this:

```
mysql> UPDATE setup_instruments SET ENABLED = 'YES', TIMED = 'YES'
    -> WHERE NAME = 'transaction';
```

The `setup_consumers` table contains consumer values with names corresponding to the current and recent transaction event table names. These consumers may be used to filter collection of transaction events:

```
mysql> SELECT * FROM setup_consumers WHERE NAME LIKE '%transactions%';
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| events_transactions_current    | NO      |
| events_transactions_history    | NO      |
| events_transactions_history_long | NO      |
+--------------------------------+---------+
```

To enable all transaction consumers, do this:

```
mysql> UPDATE setup_consumers SET ENABLED = 'YES'
    -> WHERE NAME LIKE '%transactions%';
```

To enable collection of transaction events only for specific transaction event tables, enable the corresponding transaction consumers.

The `setup_timers` table contains a row with a `NAME` value of `transaction` that indicates the unit for transaction event timing. The default unit is `NANOSECOND`.

```
mysql> SELECT * FROM setup_timers WHERE NAME = 'transaction';
+-------------+------------+
| NAME        | TIMER_NAME |
+-------------+------------+
| transaction | NANOSECOND |
+-------------+------------+
```

To change the timing unit, modify the `TIMER_NAME` value:

```
mysql> UPDATE setup_timers SET TIMER_NAME = 'MICROSECOND'
    -> WHERE NAME = 'transaction';
```

For additional information about configuring event collection, see Chapter 3, *Performance Schema 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.

8.7.1 The events_transactions_current Table

The `events_transactions_current` table (added in MySQL 5.7.3) contains current transaction events, one row per thread showing the current status of the thread's most recent monitored transaction event. For example:

```
mysql> SELECT * FROM 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
```

The `events_transactions_current` table can be truncated with `TRUNCATE TABLE`.

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, up to a fixed number of rows.

For information about configuration of transaction event collection, see Section 8.7, “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 5, *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

  This column changed in MySQL 5.7.6.

  - Versions of MySQL prior to 5.7.6:

    If gtid_mode=OFF, the value is NULL. If gtid_mode=ON, this is the value of gtid_next when the transaction started. If gtid_next=AUTOMATIC the value is AUTOMATIC, otherwise the value is a GTID in UUID:NUMBER format.

  - Versions of MySQL 5.7.6 and later:

    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

  The XA transaction identifier. It has the format described in XA Transaction SQL Syntax. This column was removed in MySQL 5.7.7 and replaced with the XID_FORMAT_ID, XID_GTRID, and XID_BQUAL columns representing the components of XID values.

- XID_FORMAT_ID, XID_GTRID, and XID_BQUAL

  The components of the XA transaction identifier. They have the format described in XA Transaction SQL Syntax. These columns were added in MySQL 5.7.7 as replacements for the XID column.

- 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).

- 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` and `TIMER_WAIT` are `NULL` before MySQL 5.7.8. As of 5.7.8, `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 3.3.1, “Performance Schema Event Timing”.

- `ACCESS_MODE`

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

- `ISOLATION_LEVEL`

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

- `AUTOCOMMIT`

  Whether autcommit 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` will not appear because transactions cannot be nested.)

8.7.2 The events_transactions_history Table

The `events_transactions_history` table (added in MySQL 5.7.3) contains the most recent *`N`* transaction events per thread. The value of *`N`* is autosized at server startup. To set the table size explicitly, set the `performance_schema_events_transactions_history_size` system variable at server startup. Transaction events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full.

The `events_transactions_history` table has the same structure as `events_transactions_current`. See Section 8.7.1, “The events_transactions_current Table”.

The events_transactions_history table can be truncated with TRUNCATE TABLE.

For information about configuration of transaction event collection, see Section 8.7, “Performance Schema Transaction Tables”.

8.7.3 The events_transactions_history_long Table

The events_transactions_history_long table (added in MySQL 5.7.3) contains the most recent *N* transaction events. 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. Transaction events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full. When a thread ends, its rows are removed from the table.

The events_transactions_history_long table has the same structure as events_transactions_current. See Section 8.7.1, “The events_transactions_current Table”.

The events_transactions_history_long table can be truncated with TRUNCATE TABLE.

For information about configuration of transaction event collection, see Section 8.7, “Performance Schema Transaction Tables”.

8.8 Performance Schema Connection Tables

The Performance Schema provides statistics about connections to the server. When a client connects, it does so under a particular user name and from a particular host. The Performance Schema tracks connections per account (user name plus host name) and separately per user name and per 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

There are also summary tables that aggregate information about connections. See Section 8.15.8, “Connection Summary Tables”.

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

The connection tables all have CURRENT_CONNECTIONS and TOTAL_CONNECTIONS columns to track the current and total number of connections per “tracking value” on which 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 name plus host name combination. The users and hosts tables have a USER and HOST column, respectively, to track connections per user name and per host name.

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 rows, for hosta and hostb, each row counting two connections per host name.
- The users table has 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 to the connection, 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.

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

Each connection table can be truncated with `TRUNCATE TABLE`, which has this effect:

- Rows with `CURRENT_CONNECTIONS = 0` are deleted.
- For rows with `CURRENT_CONNECTIONS > 0`, `TOTAL_CONNECTIONS` is reset to `CURRENT_CONNECTIONS`.
- Connection summary tables that depend on the connection table are truncated implicitly (summary values are set to 0). For more information about implicit truncation, see Section 8.15.8, “Connection Summary Tables”.

8.8.1 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 8.8, “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.

8.8.2 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 8.8, “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.

8.8.3 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 8.8, “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.

8.9 Performance Schema Connection Attribute Tables

Application programs can provide key/value pairs as connection attributes to be passed to the server at connect time. For the C API, define the attribute set using the mysql_options() and mysql_options4() functions. Other MySQL Connectors may provide their own attribute-definition methods.

These 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

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.

The set of connection attributes visible on a given connection varies depending on your platform and MySQL Connector used to establish the connection.

The `libmysqlclient` client library (provided in MySQL and MySQL Connector/C distributions) 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/J defines these attributes:

- `_client_license`: The connector license type
- `_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)

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.

Some MySQL client programs define additional attributes:

- `mysqlbinlog` defines the `_client_role` attribute as `binary_log_listener`.
- Replication slave connections define `program_name` as `mysqld`, `_client_role` as `binary_log_listener`, and `_client_replication_channel_name` as the channel name.

- `FEDERATED` storage engine connections define `program_name` as `mysqld` and `_client_role` as `federated_storage`.

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; and 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 will accept. If a client attempts to send more than 64KB of attribute data, the server rejects the connection.
- 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_warnings` system variable is greater than zero:

```
[Warning] Connection attributes of length N were truncated
```

8.9.1 The session_account_connect_attrs Table

Application programs can provide key/value connection attributes to be passed to the server at connect time, using the `mysql_options()` and `mysql_options4()` C API functions. For descriptions of common attributes, see Section 8.9, “Performance Schema Connection Attribute Tables”.

The `session_account_connect_attrs` table contains connection attributes only for sessions for your own account. To see connection attributes for all sessions, look in the `session_connect_attrs` table.

The `session_account_connect_attrs` table contains 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.

8.9.2 The session_connect_attrs Table

Application programs can provide key/value connection attributes to be passed to the server at connect time, using the `mysql_options()` and `mysql_options4()` C API functions. For descriptions of common attributes, see Section 8.9, “Performance Schema Connection Attribute Tables”.

The `session_connect_attrs` table contains connection attributes for all sessions. To see connection attributes only for sessions for your own account, look in the `session_account_connect_attrs` table.

The `session_connect_attrs` table contains 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.

8.10 Performance Schema User Variable Tables

As of MySQL 5.7.5, 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 contains 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.

8.11 Performance Schema Replication Tables

As of MySQL 5.7.2, the Performance Schema provides tables that expose replication information. This is similar to the information available from the `SHOW SLAVE STATUS` statement, but representation in table form is more accessible and has usability benefits:

- `SHOW SLAVE STATUS` output is useful for visual inspection, but not so much for programmatic use. By contrast, using the Performance Schema tables, information about slave 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 multi-threaded slave operation, `SHOW SLAVE 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 avilable from `SHOW SLAVE 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 several replication-related tables:

- Tables that contain information about the connection of the slave server to the master server:
  - `replication_connection_configuration`: Configuration parameters for connecting to the master
  - `replication_connection_status`: Current status of the connection to the master
- Tables that contain general (not thread-specific) information about the transaction applier:
  - `replication_applier_configuration`: Configuration parameters for the transaction applier on the slave. Renamed from `replication_execute_configuration` in MySQL 5.7.6.
  - `replication_applier_status`: Current status of the transaction applier on the slave. Renamed from `replication_execute_status` in MySQL 5.7.6.
- Tables that contain information about specific threads responsible for applying transactions received from the master:
  - `replication_applier_status_by_coordinator`: Status of the applier (formerly SQL or coordinator) thread. Renamed from `replication_execute_status_by_coordinator` in MySQL 5.7.6.
  - `replication_applier_status_by_worker`: Worker thread applier status (empty unless slave is multi-threaded). Renamed from `replication_execute_status_by_worker` in MySQL 5.7.6.
- Tables that contain information about replication group members:
  - `replication_group_members`: Provides network and status information for group members.
  - `replication_group_member_stats`: Provides statistical information about group members and transaction in which they participate.

The following sections describe each replication table in more detail, including the correspondence between the columns produced by `SHOW SLAVE 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 SLAVE 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 MASTER TO`, the tables are empty.
- After `CHANGE MASTER TO`, the configuration parameters can be seen in the tables. At this time, there are no active slave threads, so the `THREAD_ID` columns are `NULL` and the `SERVICE_STATE` columns have a value of `OFF`.
- After `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 master server has a value of `CONNECTING` while it establishes the connection, and `ON` thereafter as long as the connection lasts.
- After `STOP SLAVE`, 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 SLAVE` or threads dying due to an error.
- The `replication_applier_status_by_worker` table is nonempty only when the slave is operating in multi-threaded mode. That is, if the `slave_parallel_workers` system variable is greater than 0, this table is populated when `START SLAVE` is executed, and the number of rows shows the number of workers.

SHOW SLAVE STATUS Information Not In the Replication Tables

The information in the Performance Schema replication tables differs somewhat from the information available from `SHOW SLAVE 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 SLAVE 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.
- 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_coordinator` table (and `replication_applier_status_by_worker` if the slave is multi-threaded). 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 `Slave_IO_State` and `Slave_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 slave. 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.

Status Variables Moved to Replication Tables

As of MySQL version 5.7.5, the following status variables (previously monitored using `SHOW STATUS`) were moved to the Perfomance Schema replication tables:

- `Slave_retried_transactions`
- `Slave_last_heartbeat`
- `Slave_received_heartbeats`
- `Slave_heartbeat_period`
- `Slave_running`

These status variables are now only relevant when a single replication channel is being used because they *only* report the status of the default replication channel. When multiple replication channels exist, use the Performance Schema replication tables described in this section, which report these variables for each existing replication channel.

Replication Channels

The first column of the replication Performance Schema tables is `CHANNEL_NAME`. This enables the tables to be viewed per replication channel, added in MySQL 5.7.6. When you are using multiple replication channels on a slave, you can filter the tables per replication channel to monitor a specific replication channel. See Replication Channels and Multi-Source Replication Monitoring for more information.

8.11.1 The replication_connection_configuration Table

This table shows the configuration parameters used by the slave server for connecting to the master server. Parameters stored in the table can be changed at runtime with the `CHANGE MASTER TO` statement, as indicated in the column descriptions. This table was added in MySQL 5.7.2.

Compared to the `replication_connection_status` table, `replication_connection_configuration` changes less frequently. It contains values that define how the slave connects to the master and that remain constant during the connection, whereas `replication_connection_status` contains values that change during the connection.

The `replication_connection_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.

- `HOST`

  The master host that the slave is connected to. (`CHANGE MASTER TO` option: `MASTER_HOST`)

- `PORT`

  The port used to connect to the master. (`CHANGE MASTER TO` option: `MASTER_PORT`)

- `USER`

  The user name of the account used to connect to the master. (`CHANGE MASTER TO` option: `MASTER_USER`)

- `NETWORK_INTERFACE`

  The network interface that the slave is bound to, if any. (`CHANGE MASTER TO` option: `MASTER_BIND`)

- `AUTO_POSITION`

  1 if autopositioning is in use; otherwise 0. (`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 slave to connect to the master, if any.

  `SSL_ALLOWED` has these values:

  - `Yes` if an SSL connection to the master is permitted
  - `No` if an SSL connection to the master is not permitted
  - `Ignored` if an SSL connection is permitted but the slave server does not have SSL support enabled

  `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`.

  Prior to MySQL 5.7.4, the value of `SSL_CRL_PATH` was not displayed correctly. (Bug #18174719)

- `CONNECTION_RETRY_INTERVAL`

  The number of seconds between connect retries. (`CHANGE MASTER TO` option: `MASTER_CONNECT_RETRY`)

- `CONNECTION_RETRY_COUNT`

The number of times the slave can attempt to reconnect to the master in the event of a lost connection. (`CHANGE MASTER TO` option: `MASTER_RETRY_COUNT`)

- `HEARTBEAT_INTERVAL`

  The replication heartbeat interval on a slave, measured in seconds. Added in MySQL 5.7.5.

The following table shows the correspondence between `replication_connection_configuration` columns and `SHOW SLAVE STATUS` columns.

`replication_connection_configuration` Column	**`SHOW SLAVE STATUS` Column**
`HOST`	`Master_Host`
`PORT`	`Master_Port`
`USER`	`Master_User`
`NETWORK_INTERFACE`	`Master_Bind`
`AUTO_POSITION`	`Auto_Position`
`SSL_ALLOWED`	`Master_SSL_Allowed`
`SSL_CA_FILE`	`Master_SSL_CA_File`
`SSL_CA_PATH`	`Master_SSL_CA_Path`
`SSL_CERTIFICATE`	`Master_SSL_Cert`
`SSL_CIPHER`	`Master_SSL_Cipher`
`SSL_KEY`	`Master_SSL_Key`
`SSL_VERIFY_SERVER_CERTIFICATE`	`Master_SSL_Verify_Server_Cert`
`SSL_CRL_FILE`	`Master_SSL_Crl`
`SSL_CRL_PATH`	`Master_SSL_Crlpath`
`CONNECTION_RETRY_INTERVAL`	`Connect_Retry`
`CONNECTION_RETRY_COUNT`	`Master_Retry_Count`

8.11.2 The replication_connection_status Table

This table shows the current status of the I/O thread that handles the slave server connection to the master server. This table was added in MySQL 5.7.2.

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 slave connects to the master 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`

  This column is reserved for future use. Added in MySQL 5.7.6.

- SOURCE_UUID

  The server_uuid value from the master.

- 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 master).

- RECEIVED_TRANSACTION_SET

  The set of global transaction IDs (GTIDs) corresponding to all transactions received by this slave. 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 slave's error log.

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

- LAST_ERROR_TIMESTAMP

  A timestamp in YYMMDD HH:MM:SS format that shows when the most recent I/O error took place.

- LAST_HEARTBEAT_TIMESTAMP

  A timestamp in YYMMDD HH:MM:SS format that shows when the most recent heartbeat signal was received by a replication slave. Added in MySQL 5.7.5.

- COUNT_RECEIVED_HEARTBEATS

  The total number of heartbeat signals that a replication slave received since the last time it was restarted or reset, or a CHANGE MASTER TO statement was issued. Added in MySQL 5.7.5.

The following table shows the correspondence between replication_connection_status columns and SHOW SLAVE STATUS columns.

replication_connection_status Column	SHOW SLAVE STATUS Column
SOURCE_UUID	Master_UUID
THREAD_ID	None
SERVICE_STATE	Slave_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

8.11.3 The replication_applier_configuration Table

This table shows the configuration parameters that affect transactions applied by the slave server. Parameters stored in the table can be changed at runtime with the `CHANGE MASTER TO` statement, as indicated in the column descriptions. This table was added in MySQL 5.7.2 with the name `replication_execute_configuration`, and renamed to `replication_applier_configuration` in MySQL 5.7.6.

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 slave must lag the master. (`CHANGE MASTER TO` option: `MASTER_DELAY`)

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

`replication_applier_configuration` Column	**`SHOW SLAVE STATUS` Column**
`DESIRED_DELAY`	`SQL_Delay`

8.11.4 The replication_applier_status Table

This table shows the current general transaction execution status on the slave server. This table was added in MySQL 5.7.2 with the name `replication_execute_status`, and renamed to `replication_applier_configuration` in MySQL 5.7.6.

This 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 slave is multi-threaded).

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 slave is waiting for `DESIRED_DELAY` seconds to pass since the master applied an event, 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.)

- `COUNT_TRANSACTIONS_RETRIES`

  Added in MySQL 5.7.5, shows the number of retries that were made because the slave SQL thread failed to apply a transaction.

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

`replication_applier_status` Column	`SHOW SLAVE STATUS` Column
`SERVICE_STATE`	None
`REMAINING_DELAY`	`SQL_Remaining_Delay`

8.11.5 The replication_applier_status_by_coordinator Table

For a multi-threaded slave, the slave uses multiple worker threads and a coordinator thread to manage them, and this table shows the status of the coordinator thread. In MySQL 5.7.9 and later, for a single-threaded slave, this table is empty. (Previously, this table showed the applier thread status for a single-threaded slave; this information can now be found in the `replication_applier_status_by_worker` table in such cases. See Bug #74765, Bug #20001173.) This table was added in MySQL 5.7.2 as `replication_execute_status_by_coordinator`, and renamed `replication_applier_status_by_coordinator` in MySQL 5.7.6. For a multi-threaded slave, the `replication_applier_status_by_worker` table shows the status of the worker threads.

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 of the empty string mean “no error.” If the `LAST_ERROR_MESSAGE` value is not empty, the error values also appear in the slave's error log.

  Issuing `RESET MASTER` or `RESET SLAVE` 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 Codes and Messages.

- `LAST_ERROR_TIMESTAMP`

  A timestamp in `YYMMDD HH:MM:SS` format that shows when the most recent SQL/coordinator error occurred.

The following table shows the correspondence between `replication_applier_status_by_coordinator` columns and `SHOW SLAVE STATUS` columns.

`replication_applier_status_by_coordinator` Column	`SHOW SLAVE STATUS` Column
`THREAD_ID`	None

replication_applier_status_by_coordinator Column	SHOW SLAVE STATUS Column
SERVICE_STATE	Slave_SQL_Running
LAST_ERROR_NUMBER	Last_SQL_Errno
LAST_ERROR_MESSAGE	Last_SQL_Error
LAST_ERROR_TIMESTAMP	Last_SQL_Error_Timestamp

8.11.6 The replication_applier_status_by_worker Table

In MySQL 5.7.9 and later, if the slave is not multi-threaded, this table shows the status of the applier thread. (Previously, this table was empty in such cases, and this information was reported in the `replication_applier_status_by_coordinator` table; see Bug #74765, Bug #20001173.) Otherwise, the slave uses multiple worker threads and a coordinator thread to manage them, and this table shows the status of the worker threads. This table was added in MySQL 5.7.2 as `replication_execute_status_by_worker`, and renamed `replication_applier_status_by_worker` in MySQL 5.7.6. For a multi-threaded slave, the `replication_applier_status_by_coordinator` table shows the status of the coordinator thread.

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 SLAVE`, 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_SEEN_TRANSACTION`

  The transaction that the worker has last seen. The worker has not necessarily applied this transaction because it could still be in the process of doing so.

  If the `gtid_mode` system variable value is `OFF`, this column is `ANONYMOUS`, indicating that transactions do not have global transaction identifiers (GTIDs) and are identified by file and position only.

  If `gtid_mode` is `ON`, the column value is defined as follows:

  - If no transaction has executed, the column is empty.
  - When a transaction has executed, the column is set from `gtid_next` as soon as `gtid_next` is set. From this moment, the column always shows a GTID.
  - The GTID is preserved until the next transaction is executed. If an error occurs, the column value is the GTID of the transaction being executed by the worker when the error occurred.

  - When the next GTID log event is picked up by this worker thread, this column is updated from `gtid_next` soon after `gtid_next` is set.

- `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 slave's error log.

  Issuing `RESET MASTER` or `RESET SLAVE` 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 Codes and Messages.

- `LAST_ERROR_TIMESTAMP`

  A timestamp in `YYMMDD HH:MM:SS` format that shows when the most recent worker error occurred.

The following table shows the correspondence between `replication_applier_status_by_worker` columns and `SHOW SLAVE STATUS` columns.

`replication_applier_status_by_worker` Column	**`SHOW SLAVE STATUS` Column**
`WORKER_ID`	None
`THREAD_ID`	None
`SERVICE_STATE`	None
`LAST_SEEN_TRANSACTION`	None
`LAST_ERROR_NUMBER`	`Last_SQL_Errno`
`LAST_ERROR_MESSAGE`	`Last_SQL_Error`
`LAST_ERROR_TIMESTAMP`	`Last_SQL_Error_Timestamp`

8.11.7 The replication_group_members Table

This table shows network and status information for replication group members. It was added in MySQL 5.7.6 and is reserved for future use.

The `replication_group_members` table has the following columns:

- `CHANNEL_NAME`

  Name of the group replication channel.

- `MEMBER_ID`

  Identifier for this member; same as the server UUID.

- `MEMBER_HOST`

  Network address of this member (host name or IP address).

- `MEMBER_PORT`

  Port on which the server is listening.

- `MEMBER_STATE`

  Current state of this member; can be any one of the following:

  - `OFFLINE`: The group replication plugin is installed but has not been started.
  - `RECOVERING`: The server has joined a group from which it is retrieving data.
  - `ONLINE`: The member is in a fully functioning state.

8.11.8 The replication_group_member_stats Table

This table shows statistical information for replication group members. It was added in MySQL 5.7.6 and is reserved for future use.

The `replication_group_member_stats` table has the following columns:

- `CHANNEL_NAME`

  Name of the group replication channel

- `VIEW_ID`

  Current view identifier for this group.

- `MEMBER_ID`

  Identifier for this member; same as the server UUID.

- `COUNT_TRANSACTIONS_IN_QUEUE`

  Number of transactions pending certification

- `COUNT_TRANSACTIONS_CHECKED`

  Number of transactions already certified by this member.

- `COUNT_CONFLICTS_DETECTED`

  Number of transactions that were negatively certified.

- `COUNT_TRANSACTIONS_VALIDATING`

  Number of transactions with which one can execute certification with them, but have not been garbage collected.

- `TRANSACTIONS_COMMITTED_ALL_MEMBERS`

  Set of stable group transactions.

- `LAST_CONFLICT_FREE_TRANSACTION`

  Latest transaction certified without conflicts.

8.12 Performance Schema Lock Tables

The Performance Schema exposes lock information through these tables:

- `metadata_locks`: Metadata locks held and requested
- `table_handles`: Table locks held and requested

The following sections describe these tables in more detail.

8.12.1 The metadata_locks Table

As of MySQL 5.7.3, 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 or 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 is disabled by default. To enable it, enable the `wait/lock/metadata/sql/mdl` instrument in the `setup_instruments` table.

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 or leaving lock release operations. These status values were added in MySQL 5.7.11.

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`, 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 using 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 are for locks that are released at statement or transaction end, respectively. The `EXPLICIT` value is for locks that survive statement or transaction end and are released explicitly, 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 earlier in this section.

- `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.

8.12.2 The table_handles Table

As of MySQL 5.7.3, 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.

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`.

8.13 Performance Schema System Variable Tables

> **Note**
>
> The value of the `show_compatibility_56` system variable affects the information available from the tables described here. For details, see the description of that variable in Server System Variables.

The MySQL server maintains many system variables that indicate how it is configured (see Server System Variables). As of MySQL 5.7.6, 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. (In MySQL 5.7.6 and 5.7.7, the table does not fully reflect all system variable values in effect for the current session; it includes no rows for global variables that have no session counterpart. This is corrected in MySQL 5.7.8.)
- `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.

The session variable tables (`session_variables`, `variables_by_thread`) contain information only for active sessions, not terminated sessions.

`TRUNCATE TABLE` is not supported for Performance Schema system variable tables.

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 `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 contains system variable information only about foreground threads. If not all threads are instrumented by the Performance Schema, this table will miss some rows. In this case, the `Performance_schema_thread_instances_lost` status variable will be greater than zero.

8.14 Performance Schema Status Variable Tables

> **Note**
>
> The value of the `show_compatibility_56` system variable affects the information available from the tables described here. For details, see the description of that variable in Server System Variables.

The MySQL server maintains many status variables that provide information about its operation (see Server Status Variables). As of MySQL 5.7.6, 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. (In MySQL 5.7.6 and 5.7.7, the table does not fully reflect all status variable values in effect for the current session; it includes no rows for global variables that have no session counterpart. This is corrected in MySQL 5.7.8.)

- `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 8.15.11, “Performance Schema 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 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 `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 contains status variable information only about foreground threads. If the `performance_schema_max_thread_instances` system variable is not autoscaled (set to −1) and the

maximum permitted number of instrumented thread objects is not greater than the number of background threads, the table will be 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.

8.15 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.

Event Wait Summaries:

- `events_waits_summary_global_by_event_name`: Wait events summarized per event name
- `events_waits_summary_by_instance`: Wait events summarized per instance
- `events_waits_summary_by_thread_by_event_name`: Wait events summarized per thread and event name

Stage Summaries:

- `events_stages_summary_by_thread_by_event_name`: Stage waits summarized per thread and event name
- `events_stages_summary_global_by_event_name`: Stage waits summarized per event name

Statement Summaries:

- `events_statements_summary_by_digest`: Statement events summarized per schema and digest value
- `events_statements_summary_by_thread_by_event_name`: Statement events summarized per thread and event name
- `events_statements_summary_global_by_event_name`: Statement events summarized per event name
- `events_statements_summary_by_program`: Statement events summarized per stored program (stored procedures and functions, triggers, and events) (added in MySQL 5.7.2)

- `prepared_statements_instances`: Prepared statement instances and statistics (added in MySQL 5.7.4)

Transaction Summaries:

- `events_transactions_summary_by_account_by_event_name`: Transaction events per account and event name (added in MySQL 5.7.3)
- `events_transactions_summary_by_host_by_event_name`: Transaction events per host name and event name (added in MySQL 5.7.3)
- `events_transactions_summary_by_thread_by_event_name`: Transaction events per thread and event name (added in MySQL 5.7.3)
- `events_transactions_summary_by_user_by_event_name`: Transaction events per user name and event name (added in MySQL 5.7.3)
- `events_transactions_summary_global_by_event_name`: Transaction events per event name (added in MySQL 5.7.3)

Object Wait Summaries:

- `objects_summary_global_by_type`: Object summaries

File I/O Summaries:

- `file_summary_by_event_name`: File events summarized per event name
- `file_summary_by_instance`: File events summarized per file instance

Table I/O and Lock Wait Summaries:

- `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

Connection Summaries:

- `events_waits_summary_by_account_by_event_name`: Wait events summarized per account and event name
- `events_waits_summary_by_user_by_event_name`: Wait events summarized per user name and event name
- `events_waits_summary_by_host_by_event_name`: Wait events summarized per host name and event name
- `events_stages_summary_by_account_by_event_name`: Stage events summarized per account and event name
- `events_stages_summary_by_user_by_event_name`: Stage events summarized per user name and event name
- `events_stages_summary_by_host_by_event_name`: Stage events summarized per host name and event name
- `events_statements_summary_by_digest`: Statement events summarized per schema and digest value

- `events_statements_summary_by_account_by_event_name`: Statement events summarized per account and event name
- `events_statements_summary_by_user_by_event_name`: Statement events summarized per user name and event name
- `events_statements_summary_by_host_by_event_name`: Statement events summarized per host name and event name

Socket Summaries:

- `socket_summary_by_instance`: Socket waits and I/O summarized per instance
- `socket_summary_by_event_name`: Socket waits and I/O summarized per event name

Memory Summaries:

- `memory_summary_global_by_event_name`: Memory operations summarized globally per event name (added in MySQL 5.7.2)
- `memory_summary_by_thread_by_event_name`: Memory operations summarized per thread and event name (added in MySQL 5.7.2)
- `memory_summary_by_account_by_event_name`: Memory operations summarized per account and event name (added in MySQL 5.7.2)
- `memory_summary_by_user_by_event_name`: Memory operations summarized per user and event name (added in MySQL 5.7.2)
- `memory_summary_by_host_by_event_name`: Memory operations summarized per host and event name (added in MySQL 5.7.2)

Status Variable Summaries:

- `status_by_account`: Status variables summarized by account (added in MySQL 5.7.6)
- `status_by_host`: Status variables summarized by host name (added in MySQL 5.7.6)
- `status_by_user`: Status variables summarized by user name (added in MySQL 5.7.6)

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`. Except for `events_statements_summary_by_digest` and the memory summary tables, 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.

8.15.1 Event Wait Summary Tables

The Performance Schema maintains tables for collecting current and recent wait events, and aggregates that information in summary tables. Section 8.4, “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.

Each event waits 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_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, so these instances are 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_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 event waits 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.

Example wait event summary information:

```
mysql> SELECT * FROM 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
```

```
...
```

`TRUNCATE TABLE` is permitted for wait summary tables. It resets the summary columns to zero rather than removing rows.

8.15.2 Stage Summary Tables

The Performance Schema maintains tables for collecting current and recent stage events, and aggregates that information in summary tables. Section 8.5, “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 recent stage event tables, and how to control stage event collection.

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_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_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 event wait summary tables (see Section 8.15.1, “Event Wait Summary Tables”), except that the stage summary tables aggregate events from `events_stages_current` rather than `events_waits_current`.

Example stage event summary information:

```
mysql> SELECT * FROM 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
...
```

`TRUNCATE TABLE` is permitted for stage summary tables. It resets the summary columns to zero rather than removing rows.

8.15.3 Statement Summary Tables

The Performance Schema maintains tables for collecting current and recent statement events, and aggregates that information in summary tables. Section 8.6, “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 recent statement event tables, and how to control statement event collection.

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_digest` has `SCHEMA_NAME` and `DIGEST` columns. Each row summarizes events for given schema/digest values. (The `DIGEST_TEXT` column contains the corresponding normalized statement digest text, but is neither a grouping nor summary column.)

  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_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_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:

- `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 event wait summary tables (see Section 8.15.1, “Event Wait 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.

The `events_statements_summary_by_digest` table has these additional summary columns:

- `FIRST_SEEN_TIMESTAMP`, `LAST_SEEN_TIMESTAMP`

  The times at which a statement with the given digest value were first seen and most recently seen.

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.

Example statement event summary information:

```
mysql> SELECT * FROM events_statements_summary_global_by_event_name\G
*************************** 1. row ***************************
                EVENT_NAME: statement/sql/select
                COUNT_STAR: 25
            SUM_TIMER_WAIT: 1535983999000
            MIN_TIMER_WAIT: 209823000
            AVG_TIMER_WAIT: 61439359000
            MAX_TIMER_WAIT: 1363397650000
             SUM_LOCK_TIME: 20186000000
                SUM_ERRORS: 0
              SUM_WARNINGS: 0
         SUM_ROWS_AFFECTED: 0
             SUM_ROWS_SENT: 388
         SUM_ROWS_EXAMINED: 370
SUM_CREATED_TMP_DISK_TABLES: 0
     SUM_CREATED_TMP_TABLES: 0
       SUM_SELECT_FULL_JOIN: 0
 SUM_SELECT_FULL_RANGE_JOIN: 0
           SUM_SELECT_RANGE: 0
     SUM_SELECT_RANGE_CHECK: 0
            SUM_SELECT_SCAN: 6
      SUM_SORT_MERGE_PASSES: 0
             SUM_SORT_RANGE: 0
              SUM_SORT_ROWS: 0
              SUM_SORT_SCAN: 0
          SUM_NO_INDEX_USED: 6
     SUM_NO_GOOD_INDEX_USED: 0
...
```

`TRUNCATE TABLE` is permitted for statement summary tables. For `events_statements_summary_by_digest`, it empties the table. For the other statement summary tables, it resets the summary columns to zero rather than removing rows.

Statement Digest Aggregation Rules

If the `statement_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. To do this, set the `performance_schema_digests_size` system variable to a larger value at server startup. The default size is 200.

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 3.3.3, “Event Pre-Filtering”.

8.15.4 Transaction Summary Tables

As of MySQL 5.7.3, the Performance Schema maintains tables for collecting current and recent transaction events, and aggregates that information in summary tables. Section 8.7, “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 recent transaction event tables, and how to control transaction event collection, which is disabled by default.

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 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 event wait summary tables (see Section 8.15.1, “Event Wait 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.

- 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.

Example transaction event summary information:

```
mysql> SELECT * FROM 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
```

TRUNCATE TABLE is permitted for transaction summary tables. It resets the summary columns to zero rather than removing rows.

Transaction Aggregation Rules

Transaction events are collected regardless of isolation level, access mode, or autocommit mode.

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.

8.15.5 Object Wait Summary Table

The objects_summary_global_by_type table aggregates object wait events. It 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 8.15.1, “Event Wait Summary Tables”.

Example object wait event summary information:

```
mysql> SELECT * FROM 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
...
```

`TRUNCATE TABLE` is permitted for the object summary table. It resets the summary columns to zero rather than removing rows.

8.15.6 File I/O Summary Tables

The file I/O summary tables aggregate information about I/O operations.

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.

Example file I/O event summary information:

```
mysql> SELECT * FROM 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 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
...
```

`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 will not. You may be able to ensure that I/O does occur by flushing caches or restarting the server to reset its state.

8.15.7 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

8.15.7.1 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.

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.

8.15.7.2 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 structure of table_io_waits_summary_by_index_usage is 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.

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.

8.15.7.3 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.

`TRUNCATE TABLE` is permitted for table lock summary tables. It resets the summary columns to zero rather than removing rows.

8.15.8 Connection Summary Tables

The connection summary tables are similar to the corresponding `events_xxx_summary_by_thread_by_event_name` tables, except that aggregation occurs per account, host, or user, rather than by thread. Section 8.8, “Performance Schema Connection Tables” describes the events on which connection summaries are based.

The Performance Schema maintains summary tables that aggregate connection statistics by event name and account, host, or user. Separate groups of tables are available that aggregate wait, stage, and statement events, which results in this set of connection summary tables:

- `events_waits_summary_by_account_by_event_name`: Wait events summarized per account and event name
- `events_waits_summary_by_host_by_event_name`: Wait events summarized per host name and event name
- `events_waits_summary_by_user_by_event_name`: Wait events summarized per user name and event name
- `events_stages_summary_by_account_by_event_name`: Stage events summarized per account and event name
- `events_stages_summary_by_host_by_event_name`: Stage events summarized per host name and event name
- `events_stages_summary_by_user_by_event_name`: Stage events summarized per user name and event name
- `events_statements_summary_by_account_by_event_name`: Statement events summarized per account and event name
- `events_statements_summary_by_host_by_event_name`: Statement events summarized per host name and event name
- `events_statements_summary_by_user_by_event_name`: Statement events summarized per user name and event name

In other words, the connection summary tables have names of the form `events_xxx_summary_yyy_by_event_name`, where *`xxx`* is `waits`, `stages`, or `statements`, and *`yyy`* is `account`, `user`, or `host`.

The connection summary tables provide an intermediate aggregation level:

- *`xxx`*`_summary_by_thread_by_event_name` tables are more detailed than connection summary tables
- *`xxx`*`_summary_global_by_event_name` tables are less detailed than connection summary tables

Each connection 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.

- For tables with `_by_account` in the name, the `USER`, `HOST`, and `EVENT_NAME` columns group events per account and event name.
- For tables with `_by_host` in the name, the `HOST` and `EVENT_NAME` columns group events per host name and event name.
- For tables with `_by_user` in the name, the `USER` and `EVENT_NAME` columns group events per user name and event name.

Each connection 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 are similar to the columns of the same names in the `events_waits_summary_by_instance` table. Connection summary tables for statements have additional `SUM_xxx` columns that aggregate statement types.

`TRUNCATE TABLE` is permitted for connection summary tables. It resets the summary columns to zero rather than removing rows. In addition, connection summary tables are implicitly truncated if a connection table on which they depend is truncated. Table 8.2, "Effect of Implicit Table Truncation", describes the relationship between connection table truncation and implicitly truncated tables.

Table 8.2 Effect of Implicit Table Truncation

Truncated Table	Implicitly Truncated Summary Tables
`accounts`	Tables with names matching `%_by_account%`, `%_by_thread%`
`hosts`	Tables with names matching `%_by_account%`, `%_by_host%`, `%_by_thread%`
`users`	Tables with names matching `%_by_account%`, `%_by_user%`, `%_by_thread%`

8.15.9 Socket Summary Tables

These socket summary tables aggregate timer and byte count information for socket operations:

- `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.
- `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.

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 8.15.1, "Event Wait 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_instance` has an `OBJECT_INSTANCE_BEGIN` column. Each row summarizes events for a given object.
- `socket_summary_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

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.

`TRUNCATE TABLE` is permitted for socket summary tables. Except for `events_statements_summary_by_digest`, tt resets the summary columns to zero rather than removing rows.

8.15.10 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 a 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.

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.
- `memory_summary_by_host_by_event_name` has `HOST` and `EVENT_NAME` columns. Each row summarizes events for a given host.
- `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.

- `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`

  These columns aggregate the number of calls to malloc-like and free-like functions.

- `SUM_NUMBER_OF_BYTES_ALLOC`, `SUM_NUMBER_OF_BYTES_FREE`

  These columns indicate the aggregate size of allocated and freed memory blocks.

- `CURRENT_COUNT_USED`

  This column is the aggregate 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`

  This column is the aggregate 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`

  These columns are the low and high water marks corresponding to the `CURRENT_COUNT_USED` column.

- `LOW_NUMBER_OF_BYTES_USED`, `HIGH_NUMBER_OF_BYTES_USED`

  These columns are the low and high water marks corresponding to the `CURRENT_NUMBER_OF_BYTES_USED` column.

Memory summary tables do not contain timing columns because memory events are not timed.

Example memory event summary information:

```
mysql> SELECT * FROM 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
```

`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.

Memory Instrumentation Behavior

Most memory instrumentation is disabled by default, and can be enabled or disabled dynamically 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*.

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. The built-in memory instruments are displayed only in the memory_summary_global_by_event_name table.

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 thread is instrumented, and a memory block is not instrumented, the free operation is not instrumented; no statistics are changed.
- If a thread is not instrumented, and a memory block is instrumented, the free operation is instrumented, and statistics are changed.

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: Incremented by 1
- CURRENT_COUNT_USED: Incremented 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: Incremented by 1

- `CURRENT_COUNT_USED`: Iecremented 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
- `HIGH_COUNT_USED` and `HIGH_NUMBER_OF_BYTES_USED` are higher estimates

“Lower estimates” means that 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.

“Higher estimates” means that 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.

8.15.11 Performance Schema Status Variable Summary Tables

> **Note**
>
> The value of the `show_compatibility_56` system variable affects the information available from the tables described here. For details, see the description of that variable in Server System Variables.

As of MySQL 5.7.6, the Performance Schema makes status variable information available in the tables described in Section 8.14, “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 meaning of “account” in these tables is similar to its meaning in the MySQL grant tables in the `mysql` database, in the sense that the term refers to a combination of user and host values. Where they differ is that in grant tables, the host part of an account can be a pattern, whereas in 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.

8.16 Performance Schema Miscellaneous Tables

The following sections describe tables that do not fall into the table categories discussed in the preceding sections:

- `host_cache`: Information from the internal host cache
- `performance_timers`: Which event timers are available
- `threads`: Information about server threads

8.16.1 The host_cache Table

The host_cache table provides access to the contents of the host cache, which contains client host name and IP address information and is used to avoid DNS lookups. (See DNS Lookup Optimization and the Host Cache.) The host_cache table exposes the contents of the host cache so that it can be examined using SELECT statements. The Performance Schema must be enabled or this table is empty.

FLUSH HOSTS and TRUNCATE TABLE host_cache have the same effect: They clear the host cache. This also empties the host_cache table (because it is the visible representation of the cache) and unblocks any blocked hosts (see Host 'host_name' is blocked.) FLUSH HOSTS requires the RELOAD privilege. TRUNCATE TABLE requires the DROP privilege for the host_cache table.

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 calls to DNS can be avoided. While HOST_VALIDATED is NO, DNS resolution is attempted again for each connect, 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 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).

- 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 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 user from the client host can possibly log in. 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 a proxy user A is proxied to another user B who does not exist.

- `COUNT_PROXY_USER_ACL_ERRORS`

  The number of errors detected when a 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 did not exist or the user had no privileges for accessing 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 table 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 last 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 last error seen from the client in the `IP` column.

8.16.2 The performance_timers Table

The `performance_timers` table shows which event timers are available:

```
mysql> SELECT * FROM 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 |
| TICK        |             105 |                1 |           2416 |
+-------------+-----------------+------------------+----------------+
```

The timers in `setup_timers` that you can use are those that do not have `NULL` in the other columns. If the values associated with a given timer name are `NULL`, that timer is not supported on your platform.

The `performance_timers` table has these columns:

- `TIMER_NAME`

The name by which to refer to the timer when configuring the `setup_timers` table.

- `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.

8.16.3 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 threads\G
*************************** 1. row ***************************
          THREAD_ID: 1
               NAME: thread/sql/main
               TYPE: BACKGROUND
     PROCESSLIST_ID: NULL
   PROCESSLIST_USER: NULL
   PROCESSLIST_HOST: NULL
     PROCESSLIST_DB: NULL
PROCESSLIST_COMMAND: NULL
   PROCESSLIST_TIME: 80284
  PROCESSLIST_STATE: NULL
   PROCESSLIST_INFO: NULL
   PARENT_THREAD_ID: NULL
               ROLE: NULL
       INSTRUMENTED: YES
            HISTORY: YES
    CONNECTION_TYPE: NULL
       THREAD_OS_ID: 489803
...
*************************** 4. row ***************************
          THREAD_ID: 51
               NAME: thread/sql/one_connection
               TYPE: FOREGROUND
     PROCESSLIST_ID: 34
   PROCESSLIST_USER: isabella
   PROCESSLIST_HOST: localhost
     PROCESSLIST_DB: performance_schema
PROCESSLIST_COMMAND: Query
   PROCESSLIST_TIME: 0
  PROCESSLIST_STATE: Sending data
   PROCESSLIST_INFO: SELECT * FROM threads
   PARENT_THREAD_ID: 1
               ROLE: NULL
       INSTRUMENTED: YES
            HISTORY: YES
```

```
    CONNECTION_TYPE: SSL/TLS
       THREAD_OS_ID: 755399
...
```

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 3.3.3.3, “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.

The `threads` table columns with names having a prefix of `PROCESSLIST_` provide information similar to that available from the `INFORMATION_SCHEMA.PROCESSLIST` table or the `SHOW PROCESSLIST` statement. Thus, all three sources provide thread-monitoring information. Use of `threads` differs from use of the other two sources in these ways:

- Access to `threads` does not require a mutex and has minimal impact on server performance. `INFORMATION_SCHEMA.PROCESSLIST` and `SHOW PROCESSLIST` have negative performance consequences because they require a mutex.
- `threads` provides additional information for each thread, such as whether it is a foreground or background thread, and the location within the server associated with the thread.
- `threads` provides information about background threads, so it can be used to monitor activity the other thread information sources cannot.
- 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 these reasons, DBAs who perform server monitoring using `INFORMATION_SCHEMA.PROCESSLIST` or `SHOW PROCESSLIST` may wish to monitor using the `threads` table instead.

> **Note**
>
> For `INFORMATION_SCHEMA.PROCESSLIST` and `SHOW PROCESSLIST`, 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 should not be given that privilege.

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 slaves, and slave I/O and SQL threads.

- `PROCESSLIST_ID`

  For threads that are displayed in the `INFORMATION_SCHEMA.PROCESSLIST` table, this is the same value displayed in the `ID` column of that table. It is also the value displayed in the `Id` column of `SHOW PROCESSLIST` output, and the value that `CONNECTION_ID()` would return within that thread.

  For background threads (threads 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 * FROM 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 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 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 there is none.

- `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 Thread 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.

- `PROCESSLIST_STATE`

  An action, event, or state that indicates what the thread is doing. For descriptions of `PROCESSLIST_STATE` values, see Examining Thread 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 not executing any 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.

  The `HISTORY` column was added in MySQL 5.7.8. For earlier versions in which it is not present, the Performance Schema logs historical events either for all threads or no threads, depending on which history consumers are enabled or disabled.

- `CONNECTION_TYPE`

  The protocol used to establish the connection, or `NULL` for background threads. Permitted values are `TCP/IP` (TCP/IP connection established without SSL), `SSL/TLS` (TCP/IP connection established with SSL), `Socket` (Unix socket file connection), `Named Pipe` (Windows named pipe connection), and `Shared Memory` (Windows shared memory connection).

  This column was added in MySQL 5.7.8.

- `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.

This column was added in MySQL 5.7.9.

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 6, *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 10 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                     | 200   |
| 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 Section 3.2, “Performance Schema Startup Configuration”.

Performance Schema system variables have the following meanings:

- `performance_schema`

Command-Line Format	--performance_schema=#	
System Variable	Name	performance_schema
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	Type	boolean
	Default	ON

The value of this variable is ON or OFF to indicate whether the Performance Schema is enabled. By default, the value is ON by default. 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.

As of MySQL 5.7.8, 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, depending on the setting of the show_compatibiliy_56 system variable.

- performance_schema_accounts_size

Command-Line Format	--performance_schema_accounts_size=#	
System Variable	Name	performance_schema_accounts_size
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	Type	integer
	Default	-1 (autosized)
	Min Value	-1 (autosized)
	Max Value	1048576
Permitted Values (>= 5.7.6)	Type	integer
	Default	-1 (autoscaled)
	Min Value	-1 (autoscaled)
	Max 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	Name	performance_schema_digests_size

	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`
	Min Value	`-1`
	Max 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.

- `performance_schema_events_stages_history_long_size`

Command-Line Format	`--performance_schema_events_stages_history_long_size=#`	
System Variable	**Name**	`performance_schema_events_stages_history_long_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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	**Name**	`performance_schema_events_stages_history_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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	**Name**	`performance_schema_events_statements_history_long_size`
	Variable Scope	Global

	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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	**Name**	`performance_schema_events_statements_history_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

The number of rows per thread in the `events_statements_history` table.

- `performance_schema_events_transactions_history_long_size`

Introduced	5.7.3	
Command-Line Format	`--performance_schema_events_transactions_history_long_size=#`	
System Variable	**Name**	`performance_schema_events_transactions_history_long_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

The number of rows in the `events_transactions_history_long` table. This variable was added in MySQL 5.7.3.

- `performance_schema_events_transactions_history_size`

Introduced	5.7.3	
Command-Line Format	`--performance_schema_events_transactions_history_size=#`	
System Variable	**Name**	`performance_schema_events_transactions_history_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

The number of rows per thread in the `events_transactions_history` table. This variable was added in MySQL 5.7.3.

- `performance_schema_events_waits_history_long_size`

Command-Line Format	`--performance_schema_events_waits_history_long_size=#`	
System Variable	**Name**	`performance_schema_events_waits_history_long_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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	**Name**	`performance_schema_events_waits_history_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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	**Name**	`performance_schema_hosts_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
	Min Value	`-1 (autosized)`
	Max Value	`1048576`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

	Min Value	`-1 (autoscaled)`
	Max 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	**Name**	`performance_schema_max_cond_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`80`

The maximum number of condition instruments.

- `performance_schema_max_cond_instances`

Command-Line Format	`--performance_schema_max_cond_instances=#`	
System Variable	**Name**	`performance_schema_max_cond_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The maximum number of instrumented condition objects.

- `performance_schema_max_digest_length`

Introduced	5.7.8	
Command-Line Format	`--performance_schema_max_digest_length=#`	
System Variable	**Name**	`performance_schema_max_digest_length`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`1024`

	Min Value	`0`
	Max Value	`1048576`

The maximum number of bytes available for computing statement digests (see Performance Schema Statement Digests). This variable is like `max_digest_length`, but applies to the Performance Schema only. For more information, see the description of that variable in Server System Variables

This variable was added in MySQL 5.7.8. In MySQL 5.7.6 and 5.7.7, use `max_digest_length` instead. Before 5.7.6, the value cannot be changed.

- `performance_schema_max_file_classes`

Command-Line Format	`--performance_schema_max_file_classes=#`	
System Variable	**Name**	`performance_schema_max_file_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.8)	**Type**	`integer`
	Default	`50`
Permitted Values (>= 5.7.9)	**Type**	`integer`
	Default	`80`

The maximum number of file instruments.

- `performance_schema_max_file_handles`

Command-Line Format	`--performance_schema_max_file_handles=#`	
System Variable	**Name**	`performance_schema_max_file_handles`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`32768`

The maximum number of opened file objects.

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	**Name**	`performance_schema_max_file_instances`

	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The maximum number of instrumented file objects.

- `performance_schema_max_index_stat`

Introduced	5.7.6	
Command-Line Format	`--performance_schema_max_index_stat=#`	
System Variable	**Name**	`performance_schema_max_index_stat`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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`.

This variable was added in MySQL 5.7.6.

- `performance_schema_max_memory_classes`

Introduced	5.7.2	
Command-Line Format	`--performance_schema_max_memory_classes=#`	
System Variable	**Name**	`performance_schema_max_memory_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.4)	**Type**	`integer`
	Default	`250`
Permitted Values (>= 5.7.5)	**Type**	`integer`
	Default	`320`

The maximum number of memory instruments. This variable was added in MySQL 5.7.2.

- `performance_schema_max_metadata_locks`

Introduced	5.7.3	
Command-Line Format	`--performance_schema_max_metadata_locks=#`	
System Variable	**Name**	`performance_schema_max_metadata_locks`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

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.

This variable was added in MySQL 5.7.3.

- `performance_schema_max_mutex_classes`

Command-Line Format	`--performance_schema_max_mutex_classes=#`	
System Variable	**Name**	`performance_schema_max_mutex_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`200`

The maximum number of mutex instruments.

- `performance_schema_max_mutex_instances`

Command-Line Format	`--performance_schema_max_mutex_instances=#`	
System Variable	**Name**	`performance_schema_max_mutex_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The maximum number of instrumented mutex objects.

- `performance_schema_max_prepared_statements_instances`

Introduced	5.7.4	
Command-Line Format	`--performance_schema_max_prepared_statements_instances=#`	
System Variable	**Name**	`performance_schema_max_prepared_statements_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

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. The default value of this variable is autosized based on the value of the `max_prepared_stmt_count` system variable.

This variable was added in MySQL 5.7.4.

- `performance_schema_max_rwlock_classes`

Command-Line Format	`--performance_schema_max_rwlock_classes=#`	
System Variable	**Name**	`performance_schema_max_rwlock_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.2)	**Type**	`integer`
	Default	`30`
Permitted Values (>= 5.7.3)	**Type**	`integer`
	Default	`40`

The maximum number of rwlock instruments.

- `performance_schema_max_program_instances`

Introduced	5.7.2	
Command-Line Format	`--performance_schema_max_program_instances=#`	
System Variable	**Name**	`performance_schema_max_program_instances`
	Variable Scope	Global
	Dynamic Variable	No

Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`5000`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

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.

This variable was added in MySQL 5.7.2.

- `performance_schema_max_rwlock_instances`

Command-Line Format	`--performance_schema_max_rwlock_instances=#`	
System Variable	**Name**	`performance_schema_max_rwlock_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The maximum number of instrumented rwlock objects.

- `performance_schema_max_socket_classes`

Command-Line Format	`--performance_schema_max_socket_classes=#`	
System Variable	**Name**	`performance_schema_max_socket_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`10`

The maximum number of socket instruments.

- `performance_schema_max_socket_instances`

Command-Line Format	`--performance_schema_max_socket_instances=#`	
System Variable	**Name**	`performance_schema_max_socket_instances`
	Variable Scope	Global
	Dynamic Variable	No

Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The maximum number of instrumented socket objects.

- `performance_schema_max_sql_text_length`

Introduced	5.7.6	
Command-Line Format	`--performance_schema_max_sql_text_length=#`	
System Variable	**Name**	`performance_schema_max_sql_text_length`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`1024`
	Min Value	`0`
	Max Value	`1048576`

The maximum number of bytes used to store SQL statements in the `SQL_TEXT` column of the `events_statements_current`, `events_statements_history`, and `events_statements_history_long` statement event tables. Any bytes in excess of `performance_schema_max_sql_text_length` are discarded and do not appear in the `SQL_TEXT` column. Statements differing only after that many initial bytes are indistinguishable in this 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.

This variable was added in MySQL 5.7.6.

- `performance_schema_max_stage_classes`

Command-Line Format	`--performance_schema_max_stage_classes=#`	
System Variable	**Name**	`performance_schema_max_stage_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`150`

The maximum number of stage instruments.

- `performance_schema_max_statement_classes`

Command-Line Format	--performance_schema_max_statement_classes=#	
System Variable	Name	performance_schema_max_statement_classes
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	Type	integer
	Default	-1 (autosized)

The maximum number of statement instruments. 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

Introduced	5.7.2	
Command-Line Format	--performance_schema_max_statement_stack=#	
System Variable	Name	performance_schema_max_statement_stack
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	Type	integer
	Default	10

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.

This variable was added in MySQL 5.7.2.

- performance_schema_max_table_handles

Command-Line Format	--performance_schema_max_table_handles=#	
System Variable	Name	performance_schema_max_table_handles
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	Type	integer
	Default	-1 (autosized)

Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

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.

- `performance_schema_max_table_instances`

Command-Line Format	`--performance_schema_max_table_instances=#`	
System Variable	**Name**	`performance_schema_max_table_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The maximum number of instrumented table objects.

- `performance_schema_max_table_lock_stat`

Introduced	5.7.6	
Command-Line Format	`--performance_schema_max_table_lock_stat=#`	
System Variable	**Name**	`performance_schema_max_table_lock_stat`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`-1 (autosized)`

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.

This variable was added in MySQL 5.7.6.

- `performance_schema_max_thread_classes`

Command-Line Format	`--performance_schema_max_thread_classes=#`	
System Variable	**Name**	`performance_schema_max_thread_classes`
	Variable Scope	Global
	Dynamic Variable	No

Permitted Values	**Type**	integer
	Default	50

The maximum number of thread instruments.

- performance_schema_max_thread_instances

Command-Line Format	--performance_schema_max_thread_instances=#	
System Variable	**Name**	performance_schema_max_thread_instances
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	integer
	Default	-1 (autosized)
Permitted Values (>= 5.7.6)	**Type**	integer
	Default	-1 (autoscaled)

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.

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 information only about foreground threads. If not all threads are instrumented by the Performance Schema, this table will miss some rows. In this case, the Performance_schema_thread_instances_lost status variable will be greater than zero.

- performance_schema_session_connect_attrs_size

Command-Line Format	--performance_schema_session_connect_attrs_size=#	
System Variable	**Name**	performance_schema_session_connect_attrs_size
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	integer
	Default	-1 (autosized)
	Min Value	-1
	Max Value	1048576

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_warnings` system variable value is greater than zero.

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 will accept. If a client attempts to send more than 64KB of attribute data, the server rejects the connection. For more information, see Section 8.9, “Performance Schema Connection Attribute Tables”.

- `performance_schema_setup_actors_size`

Command-Line Format	`--performance_schema_setup_actors_size=#`	
System Variable	**Name**	`performance_schema_setup_actors_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`100`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The number of rows in the `setup_actors` table.

- `performance_schema_setup_objects_size`

Command-Line Format	`--performance_schema_setup_objects_size=#`	
System Variable	**Name**	`performance_schema_setup_objects_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`100`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`

The number of rows in the `setup_objects` table.

- `performance_schema_users_size`

Command-Line Format	`--performance_schema_users_size=#`	
System Variable	**Name**	`performance_schema_users_size`

	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.7.5)	**Type**	`integer`
	Default	`-1 (autosized)`
	Min Value	`-1 (autosized)`
	Max Value	`1048576`
Permitted Values (>= 5.7.6)	**Type**	`integer`
	Default	`-1 (autoscaled)`
	Min Value	`-1 (autoscaled)`
	Max 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 11 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     |
+-------------------------------------------+-------+
```

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.

  This variable was added in MySQL 5.7.6.

- `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. This variable was added in MySQL 5.7.2.

- `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.

  This variable was added in MySQL 5.7.3.

- `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.

  This variable was added in MySQL 5.7.2.

- `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.

  This variable was added in MySQL 5.7.4.

- `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.

This variable was added in MySQL 5.7.2.

- `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_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 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 8.9, “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.

  This variable was added in MySQL 5.7.6.

- `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.

For information on using these variables to check Performance Schema status, see Chapter 6, *Performance Schema Status Monitoring*.

Chapter 12 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 3.3.2, “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 will rely 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.

   At each iteration, the Performance Schema output, particularly the `events_waits_history_long` table, will contain less and less “noise” caused by nonsignificant instruments, and given that this table has a fixed size, will contain more and more data relevant to the analysis of the problem at hand.

   At each iteration, investigation should lead closer and closer to the root cause of the problem, as the “signal/noise” ratio will improve, 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 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 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 events_waits_current WHERE THREAD_ID = thread_2;
   ```

12.1 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.

As of MySQL 5.7.8, 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 will run the query. By default, `setup_actors` is configured to allow monitoring and historical event collection for all foreground threads:

   ```
   mysql> SELECT * FROM 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 will run 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 |
+--------------------------------+----------+
15 rows in set (0.00 sec)
```