

MySQL Performance Schema

Abstract

This is the MySQL Performance Schema extract from the MySQL 5.6 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.

Document generated on: 2019-01-15 (revision: 60608)

Table of Contents

Preface and Legal Notices	v
1 MySQL Performance Schema	1
2 Performance Schema Quick Start	3
3 Performance Schema Build Configuration	9
4 Performance Schema Startup Configuration	11
5 Performance Schema Runtime Configuration	15
5.1 Performance Schema Event Timing	16
5.2 Performance Schema Event Filtering	19
5.3 Event Pre-Filtering	20
5.4 Pre-Filtering by Instrument	21
5.5 Pre-Filtering by Object	23
5.6 Pre-Filtering by Thread	24
5.7 Pre-Filtering by Consumer	25
5.8 Example Consumer Configurations	27
5.9 Naming Instruments or Consumers for Filtering Operations	32
5.10 Determining What Is Instrumented	33
6 Performance Schema Queries	35
7 Performance Schema Instrument Naming Conventions	37
8 Performance Schema Status Monitoring	41
9 Performance Schema General Table Characteristics	45
10 Performance Schema Table Descriptions	47
10.1 Performance Schema Table Index	48
10.2 Performance Schema Setup Tables	49
10.2.1 The setup_actors Table	50
10.2.2 The setup_consumers Table	50
10.2.3 The setup_instruments Table	51
10.2.4 The setup_objects Table	52
10.2.5 The setup_timers Table	53
10.3 Performance Schema Instance Tables	54
10.3.1 The cond_instances Table	54
10.3.2 The file_instances Table	55
10.3.3 The mutex_instances Table	55
10.3.4 The rwlock_instances Table	56
10.3.5 The socket_instances Table	57
10.4 Performance Schema Wait Event Tables	59
10.4.1 The events_waits_current Table	61
10.4.2 The events_waits_history Table	63
10.4.3 The events_waits_history_long Table	63
10.5 Performance Schema Stage Event Tables	64
10.5.1 The events_stages_current Table	66
10.5.2 The events_stages_history Table	67
10.5.3 The events_stages_history_long Table	67
10.6 Performance Schema Statement Event Tables	68
10.6.1 The events_statements_current Table	71
10.6.2 The events_statements_history Table	75
10.6.3 The events_statements_history_long Table	75
10.7 Performance Schema Connection Tables	75
10.7.1 The accounts Table	77
10.7.2 The hosts Table	78
10.7.3 The users Table	78
10.8 Performance Schema Connection Attribute Tables	78
10.8.1 The session_account_connect_attrs Table	80
10.8.2 The session_connect_attrs Table	81
10.9 Performance Schema Summary Tables	81
10.9.1 Wait Event Summary Tables	82

10.9.2 Stage Summary Tables	84
10.9.3 Statement Summary Tables	85
10.9.4 Object Wait Summary Table	87
10.9.5 File I/O Summary Tables	88
10.9.6 Table I/O and Lock Wait Summary Tables	89
10.9.7 Socket Summary Tables	92
10.10 Performance Schema Miscellaneous Tables	93
10.10.1 The host_cache Table	93
10.10.2 The performance_timers Table	96
10.10.3 The threads Table	97
11 Performance Schema and Plugins	101
12 Performance Schema System Variables	103
13 Performance Schema Status Variables	117
14 Using the Performance Schema to Diagnose Problems	121
14.1 Query Profiling Using Performance Schema	122

Preface and Legal Notices

This is the MySQL Performance Schema extract from the MySQL 5.6 Reference Manual.

Licensing information—MySQL 5.6. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL 5.6, see the [MySQL 5.6 Commercial Release License Information User Manual](#) for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of MySQL 5.6, see the [MySQL 5.6 Community Release License Information User Manual](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Licensing information—MySQL NDB Cluster 7.3. This product may include third-party software, used under license. If you are using a *Commercial* release of NDB Cluster 7.3, see the [MySQL NDB Cluster 7.3 Commercial Release License Information User Manual](#) for licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of NDB Cluster 7.3, see the [MySQL NDB Cluster 7.3 Community Release License Information User Manual](#) for licensing information relating to third-party software that may be included in this Community release.

Licensing information—MySQL NDB Cluster 7.4. This product may include third-party software, used under license. If you are using a *Commercial* release of NDB Cluster 7.4, see the [MySQL NDB Cluster 7.4 Commercial Release License Information User Manual](#) for licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of NDB Cluster 7.4, see the [MySQL NDB Cluster 7.4 Community Release License Information User Manual](#) for licensing information relating to third-party software that may be included in this Community release.

Legal Notices

Copyright © 1997, 2019, 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.

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

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

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit

<http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Chapter 1 MySQL Performance Schema

The MySQL Performance Schema is a feature for monitoring MySQL Server execution at a low level. The Performance Schema has these characteristics:

- The Performance Schema provides a way to inspect internal execution of the server at runtime. It is implemented using the [PERFORMANCE_SCHEMA](#) storage engine and the [performance_schema](#) database. The Performance Schema focuses primarily on performance data. This differs from [INFORMATION_SCHEMA](#), which serves for inspection of metadata.
- The Performance Schema monitors server events. An “event” is anything the server does that takes time and has been instrumented so that timing information can be collected. In general, an event could be a function call, a wait for the operating system, a stage of an SQL statement execution such as parsing or sorting, or an entire statement or group of statements. Event collection provides access to information about synchronization calls (such as for mutexes) file and table I/O, table locks, and so forth for the server and for several storage engines.
- Performance Schema events are distinct from events written to the server's binary log (which describe data modifications) and Event Scheduler events (which are a type of stored program).
- Performance Schema events are specific to a given instance of the MySQL Server. Performance Schema tables are considered local to the server, and changes to them are not replicated or written to the binary log.
- Current events are available, as well as event histories and summaries. This enables you to determine how many times instrumented activities were performed and how much time they took. Event information is available to show the activities of specific threads, or activity associated with particular objects such as a mutex or file.
- The [PERFORMANCE_SCHEMA](#) storage engine collects event data using “instrumentation points” in server source code.
- Collected events are stored in tables in the [performance_schema](#) database. These tables can be queried using [SELECT](#) statements like other tables.
- Performance Schema configuration can be modified dynamically by updating tables in the [performance_schema](#) database through SQL statements. Configuration changes affect data collection immediately.
- Tables in the Performance Schema are in-memory tables that use no persistent on-disk storage. The contents are repopulated beginning at server startup and discarded at server shutdown.
- Monitoring is available on all platforms supported by MySQL.

Some limitations might apply: The types of timers might vary per platform. Instruments that apply to storage engines might not be implemented for all storage engines. Instrumentation of each third-party engine is the responsibility of the engine maintainer. See also [Restrictions on Performance Schema](#).

- Data collection is implemented by modifying the server source code to add instrumentation. There are no separate threads associated with the Performance Schema, unlike other features such as replication or the Event Scheduler.

The Performance Schema is intended to provide access to useful information about server execution while having minimal impact on server performance. The implementation follows these design goals:

- Activating the Performance Schema causes no changes in server behavior. For example, it does not cause thread scheduling to change, and it does not cause query execution plans (as shown by [EXPLAIN](#)) to change.

-
- No memory allocation is done beyond that which occurs during server startup. By using early allocation of structures with a fixed size, it is never necessary to resize or reallocate them, which is critical for achieving good runtime performance.
 - 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. In MySQL 5.7, the `sys` schema is installed by default. For MySQL 5.6, you can obtain it from the schema development website at <https://github.com/mysql/mysql-sys>. For usage instructions, see [MySQL sys Schema](#).

Chapter 2 Performance Schema Quick Start

This section briefly introduces the Performance Schema with examples that show how to use it. For additional examples, see [Chapter 14, Using the Performance Schema to Diagnose Problems](#).

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> mysql --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 Build 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;
```

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 |
| mutex_instances |
| objects_summary_global_by_type |
| performance_timers |
| 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_io_waits_summary_by_index_usage |
| table_io_waits_summary_by_table |
| table_lock_waits_summary_by_table |
| threads |
| users |
+-----+
```

```
mysql> SHOW TABLES FROM performance_schema;
```

```
+-----+
| Tables_in_performance_schema |
+-----+
| accounts |
| cond_instances |
| events_stages_current |
| events_stages_history |
| events_stages_history_long |
| ... |
```

The number of Performance Schema tables increases over time as implementation of additional instrumentation proceeds.

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

To see the structure of individual tables, use `SHOW CREATE TABLE`:

```
mysql> SHOW CREATE TABLE performance_schema.setup_consumers\G
***** 1. row *****
      Table: setup_consumers
Create Table: CREATE TABLE `setup_consumers` (
  `NAME` varchar(64) NOT NULL,
  `ENABLED` enum('YES','NO') NOT NULL
) 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 10, Performance Schema Table Descriptions](#).

Initially, not all instruments and consumers are enabled, so the performance schema does not collect all events. To turn all of these on and enable event timing, execute two statements (the row counts may differ depending on MySQL version):

```
mysql> UPDATE performance_schema.setup_instruments
      SET ENABLED = 'YES', TIMED = 'YES';
Query OK, 338 rows affected (0.12 sec)
mysql> UPDATE performance_schema.setup_consumers
      SET ENABLED = 'YES';
Query OK, 8 rows affected (0.00 sec)
```

To see what the server is doing at the moment, examine the `events_waits_current` table. It contains one row per thread showing each thread's most recent monitored event:

```
mysql> SELECT *
      FROM performance_schema.events_waits_current\G
***** 1. row *****
      THREAD_ID: 0
      EVENT_ID: 5523
      END_EVENT_ID: 5523
      EVENT_NAME: wait/synch/mutex/mysys/THR_LOCK::mutex
      SOURCE: thr_lock.c:525
      TIMER_START: 201660494489586
      TIMER_END: 201660494576112
      TIMER_WAIT: 86526
      SPINS: NULL
      OBJECT_SCHEMA: NULL
      OBJECT_NAME: NULL
      INDEX_NAME: NULL
      OBJECT_TYPE: NULL
      OBJECT_INSTANCE_BEGIN: 142270668
      NESTING_EVENT_ID: NULL
      NESTING_EVENT_TYPE: NULL
      OPERATION: lock
      NUMBER_OF_BYTES: NULL
      FLAGS: 0
      ...
```

This event indicates that thread 0 was waiting for 86,526 picoseconds to acquire a lock on `THR_LOCK::mutex`, a mutex in the `mysys` subsystem. The first few columns provide the following information:

- The ID columns indicate which thread the event comes from and the event number.
- `EVENT_NAME` indicates what was instrumented and `SOURCE` indicates which source file contains the instrumented code.
- The timer columns show when the event started and stopped and how long it took. If an event is still in progress, the `TIMER_END` and `TIMER_WAIT` values are `NULL`. Timer values are approximate and expressed in picoseconds. For information about timers and event time collection, see [Section 5.1, “Performance Schema Event Timing”](#).

The history tables contain the same kind of rows as the current-events table but have more rows and show what the server has been doing “recently” rather than “currently.” The `events_waits_history` and `events_waits_history_long` tables contain the most recent 10 events per thread and most recent 10,000 events, respectively. For example, to see information for recent events produced by thread 13, do this:

```
mysql> SELECT EVENT_ID, EVENT_NAME, TIMER_WAIT
FROM performance_schema.events_waits_history
WHERE THREAD_ID = 13
ORDER BY EVENT_ID;
```

EVENT_ID	EVENT_NAME	TIMER_WAIT
86	wait/synch/mutex/mysys/THR_LOCK::mutex	686322
87	wait/synch/mutex/mysys/THR_LOCK_malloc	320535
88	wait/synch/mutex/mysys/THR_LOCK_malloc	339390
89	wait/synch/mutex/mysys/THR_LOCK_malloc	377100
90	wait/synch/mutex/sql/LOCK_plugin	614673
91	wait/synch/mutex/sql/LOCK_open	659925
92	wait/synch/mutex/sql/THD::LOCK_thd_data	494001
93	wait/synch/mutex/mysys/THR_LOCK_malloc	222489
94	wait/synch/mutex/mysys/THR_LOCK_malloc	214947
95	wait/synch/mutex/mysys/LOCK_alarm	312993

As new events are added to a history table, older events are discarded if the table is full.

Summary tables provide aggregated information for all events over time. The tables in this group summarize event data in different ways. To see which instruments have been executed the most times or have taken the most wait time, sort the `events_waits_summary_global_by_event_name` table on the `COUNT_STAR` or `SUM_TIMER_WAIT` column, which correspond to a `COUNT(*)` or `SUM(TIMER_WAIT)` value, respectively, calculated over all events:

```
mysql> SELECT EVENT_NAME, COUNT_STAR
FROM performance_schema.events_waits_summary_global_by_event_name
ORDER BY COUNT_STAR DESC LIMIT 10;
```

EVENT_NAME	COUNT_STAR
wait/synch/mutex/mysys/THR_LOCK_malloc	6419
wait/io/file/sql/FRM	452
wait/synch/mutex/sql/LOCK_plugin	337
wait/synch/mutex/mysys/THR_LOCK_open	187
wait/synch/mutex/mysys/LOCK_alarm	147
wait/synch/mutex/sql/THD::LOCK_thd_data	115
wait/io/file/myisam/kfile	102
wait/synch/mutex/sql/LOCK_global_system_variables	89
wait/synch/mutex/mysys/THR_LOCK::mutex	89
wait/synch/mutex/sql/LOCK_open	88

```
mysql> SELECT EVENT_NAME, SUM_TIMER_WAIT
FROM performance_schema.events_waits_summary_global_by_event_name
ORDER BY SUM_TIMER_WAIT DESC LIMIT 10;
```

EVENT_NAME	SUM_TIMER_WAIT
------------	----------------

wait/io/file/sql/MYSQL_LOG	1599816582
wait/synch/mutex/mysys/THR_LOCK_malloc	1530083250
wait/io/file/sql/binlog_index	1385291934
wait/io/file/sql/FRM	1292823243
wait/io/file/myisam/kfile	411193611
wait/io/file/myisam/dfile	322401645
wait/synch/mutex/mysys/LOCK_alarm	145126935
wait/io/file/sql/casetest	104324715
wait/synch/mutex/sql/LOCK_plugin	86027823
wait/io/file/sql/pid	72591750

These results show that the `THR_LOCK_malloc` mutex is “hot,” both in terms of how often it is used and amount of time that threads wait attempting to acquire it.

Note

The `THR_LOCK_malloc` mutex is used only in debug builds. In production builds it is not hot because it is nonexistent.

Instance tables document what types of objects are instrumented. An instrumented object, when used by the server, produces an event. These tables provide event names and explanatory notes or status information. For example, the `file_instances` table lists instances of instruments for file I/O operations and their associated files:

```
mysql> SELECT *
FROM performance_schema.file_instances\G
***** 1. row *****
FILE_NAME: /opt/mysql-log/60500/binlog.000007
EVENT_NAME: wait/io/file/sql/binlog
OPEN_COUNT: 0
***** 2. row *****
FILE_NAME: /opt/mysql/60500/data/mysql/tables_priv.MYI
EVENT_NAME: wait/io/file/myisam/kfile
OPEN_COUNT: 1
***** 3. row *****
FILE_NAME: /opt/mysql/60500/data/mysql/columns_priv.MYI
EVENT_NAME: wait/io/file/myisam/kfile
OPEN_COUNT: 1
...
```

Setup tables are used to configure and display monitoring characteristics. For example, `setup_instruments` lists the set of instruments for which events can be collected and shows which of them are enabled:

```
mysql> SELECT * FROM performance_schema.setup_instruments;
+-----+-----+-----+
| NAME                                | ENABLED | TIMED |
+-----+-----+-----+
...
| stage/sql/end                       | NO      | NO    |
| stage/sql/executing                 | NO      | NO    |
| stage/sql/init                      | NO      | NO    |
| stage/sql/insert                    | NO      | NO    |
...
| statement/sql/load                  | YES     | YES   |
| statement/sql/grant                 | YES     | YES   |
| statement/sql/check                 | YES     | YES   |
| statement/sql/flush                 | YES     | YES   |
...
| wait/synch/mutex/sql/LOCK_global_read_lock | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db   | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager   | YES     | YES   |
...
| wait/synch/rwlock/sql/LOCK_grant    | YES     | YES   |
| wait/synch/rwlock/sql/LOGGER::LOCK_logger | YES     | YES   |
```

wait/synch/rwlock/sql/LOCK_sys_init_connect	YES	YES	
wait/synch/rwlock/sql/LOCK_sys_init_slave	YES	YES	
...			
wait/io/file/sql/binlog	YES	YES	
wait/io/file/sql/binlog_index	YES	YES	
wait/io/file/sql/casetest	YES	YES	
wait/io/file/sql/dbopt	YES	YES	
...			

To understand how to interpret instrument names, see [Chapter 7, Performance Schema Instrument Naming Conventions](#).

To control whether events are collected for an instrument, set its `ENABLED` value to `YES` or `NO`. For example:

```
mysql> UPDATE performance_schema.setup_instruments
      SET ENABLED = 'NO'
      WHERE NAME = 'wait/synch/mutex/sql/LOCK_mysql_create_db';
```

The Performance Schema uses collected events to update tables in the `performance_schema` database, which act as “consumers” of event information. The `setup_consumers` table lists the available consumers and which are enabled:

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

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

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

There are some miscellaneous tables that do not fall into any of the previous groups. For example, `performance_timers` lists the available event timers and their characteristics. For information about timers, see [Section 5.1, “Performance Schema Event Timing”](#).

Chapter 3 Performance Schema Build Configuration

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](#).

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.

Chapter 4 Performance Schema Startup Configuration

To use the MySQL Performance Schema, it must be enabled at server startup to enable event collection to occur.

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'
```

Longer instrument name strings take precedence over shorter pattern names, regardless of order. For information about specifying patterns to select instruments, see [Section 5.9, “Naming Instruments or Consumers for Filtering Operations”](#).

An unrecognized instrument name is ignored. It is possible that a plugin installed later may create the instrument, at which time the name is recognized and configured.

To control a consumer at server startup, use an option of this form:

```
--performance-schema-consumer-consumer_name=value
```

Here, `consumer_name` is a consumer name such as `events_waits_history`, and `value` is one of these values:

- `OFF`, `FALSE`, or `0`: Do not collect events for the consumer
- `ON`, `TRUE`, or `1`: Collect events for the consumer

For example, to enable the `events_waits_history` consumer, use this option:

```
--performance-schema-consumer-events-waits-history=ON
```

The permitted consumer names can be found by examining the `setup_consumers` table. Patterns are not permitted. Consumer names in the `setup_consumers` table use underscores, but for consumers set at startup, dashes and underscores within the name are equivalent.

The Performance Schema includes several system variables that provide configuration information:

```
mysql> SHOW VARIABLES LIKE 'perf%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| performance_schema | ON |
| performance_schema_accounts_size | 100 |
| performance_schema_digests_size | 200 |
| performance_schema_events_stages_history_long_size | 10000 |
| performance_schema_events_stages_history_size | 10 |
| performance_schema_events_statements_history_long_size | 10000 |
| performance_schema_events_statements_history_size | 10 |
| performance_schema_events_waits_history_long_size | 10000 |
| performance_schema_events_waits_history_size | 10 |
| performance_schema_hosts_size | 100 |
| performance_schema_max_cond_classes | 80 |
| performance_schema_max_cond_instances | 1000 |
| ...
```

The `performance_schema` variable is `ON` or `OFF` to indicate whether the Performance Schema is enabled or disabled. The other variables indicate table sizes (number of rows) or memory allocation values.

Note

With the Performance Schema enabled, the number of Performance Schema instances affects the server memory footprint, perhaps to a large extent. 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.

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. To see which parameters are autosized under this policy, use `mysqld --verbose --help` and look for those with a default value of `-1`, or see [Chapter 12, Performance Schema System Variables](#).

For each autosized parameter that is not set at server startup, the Performance Schema determines how to set its value based on the value of the following system values, which are considered as “hints” about how you have configured your MySQL server:

```
max_connections
open_files_limit
table_definition_cache
```

`table_open_cache`

To override autosizing 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.

If the Performance Schema is disabled, its autosized parameters remain set to -1 and `SHOW VARIABLES` displays -1.

Chapter 5 Performance Schema Runtime Configuration

Table of Contents

5.1 Performance Schema Event Timing	16
5.2 Performance Schema Event Filtering	19
5.3 Event Pre-Filtering	20
5.4 Pre-Filtering by Instrument	21
5.5 Pre-Filtering by Object	23
5.6 Pre-Filtering by Thread	24
5.7 Pre-Filtering by Consumer	25
5.8 Example Consumer Configurations	27
5.9 Naming Instruments or Consumers for Filtering Operations	32
5.10 Determining What Is Instrumented	33

Specific Performance Schema features can be enabled at runtime to control which types of event collection occur.

Performance Schema setup tables contain information about monitoring configuration:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'performance_schema'
AND TABLE_NAME LIKE 'setup%';
+-----+
| TABLE_NAME |
+-----+
| setup_actors |
| setup_consumers |
| setup_instruments |
| setup_objects |
| setup_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 10.2, “Performance Schema Setup Tables”](#).

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

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

```

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

```

For discussion of timers, see [Section 5.1, “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 5.2, “Performance Schema Event Filtering”](#), discusses how you can modify these tables to affect event collection.

If there are Performance Schema configuration changes that must be made at runtime using SQL statements and you would like these changes to take effect each time the server starts, put the statements in a file and start the server with the `--init-file=filename` 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.

5.1 Performance Schema Event Timing

Events are collected by means of instrumentation added to the server source code. Instruments time events, which is how the Performance Schema provides an idea of how long events take. It is also possible to configure instruments not to collect timing information. This section discusses the available timers and their characteristics, and how timing values are represented in events.

Performance Schema Timers

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_schema.performance_timers;
+-----+-----+-----+-----+
| TIMER_NAME | TIMER_FREQUENCY | TIMER_RESOLUTION | TIMER_OVERHEAD |
+-----+-----+-----+-----+
| CYCLE      | 2389029850     | 1                | 72             |
| NANOSECOND | 1000000000     | 1                | 112            |
| MICROSECOND | 1000000        | 1                | 136            |
| MILLISECOND | 1036           | 1                | 168            |
| TICK       | 105            | 1                | 2416           |
+-----+-----+-----+-----+

```

If the values associated with a given timer name are `NULL`, that timer is not supported on your platform. The rows that do not contain `NULL` indicate which timers you can use in `setup_timers`.

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 performance_schema.setup_timers;
+-----+-----+
| NAME      | TIMER_NAME |
+-----+-----+
| idle      | MICROSECOND |
| wait      | CYCLE      |
| stage     | NANOSECOND |
| statement | NANOSECOND |
+-----+-----+

mysql> UPDATE performance_schema.setup_timers
      SET TIMER_NAME = 'MICROSECOND'
      WHERE NAME = 'idle';

mysql> SELECT * FROM performance_schema.setup_timers;
+-----+-----+
| NAME      | TIMER_NAME |
+-----+-----+
| idle      | MICROSECOND |
| wait      | CYCLE      |
| stage     | NANOSECOND |
| statement | 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, macOS, Linux, Solaris, and other Unix flavors), PowerPC, and IA-64.

Performance Schema Timer Representation in Events

Rows in Performance Schema tables that store current events and historical events have three columns to represent timing information: `TIMER_START` and `TIMER_END` indicate when an event started and finished, and `TIMER_WAIT` indicates event duration.

The `setup_instruments` table has an `ENABLED` column to indicate the instruments for which to collect events. The table also has a `TIMED` column to indicate which instruments are timed. If an instrument is not enabled, it produces no events. If an enabled instrument is not timed, events produced by the instrument have `NULL` for the `TIMER_START`, `TIMER_END`, and `TIMER_WAIT` timer values. This in turn causes those values to be ignored when calculating aggregate time values in summary tables (sum, minimum, maximum, and average).

Internally, times within events are stored in units given by the timer in effect when event timing begins. For display when events are retrieved from Performance Schema tables, times are shown in picoseconds (trillionths of a second) to normalize them to a standard unit, regardless of which timer is selected.

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 5.6.26, while a wait, stage, or statement 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
```

As of MySQL 5.6.26, current-event timing provides more information. To make it possible to determine how long a not-yet-completed event has been running, the timer columns are set as follows:

- `TIMER_START` is populated.
- `TIMER_END` is populated with the current timer value.
- `TIMER_WAIT` is populated with the time elapsed so far (`TIMER_END - TIMER_START`).

Events that have not yet completed have an `END_EVENT_ID` value of `NULL`. To assess time elapsed so far for an event, use the `TIMER_WAIT` column. Therefore, to identify events that have not yet completed and have taken longer than `N` picoseconds thus far, monitoring applications can use this expression in queries:

```
WHERE END_EVENT_ID IS NULL AND TIMER_WAIT > N
```

Event identification as just described assumes that the corresponding instruments have `ENABLED` and `TIMED` set to `YES` and that the relevant consumers are enabled.

5.2 Performance Schema Event Filtering

Events are processed in a producer/consumer fashion:

- Instrumented code is the source for events and produces events to be collected. The `setup_instruments` table lists the instruments for which events can be collected, whether they are enabled, and (for enabled instruments) whether to collect timing information:

```
mysql> SELECT * FROM performance_schema.setup_instruments;
+-----+-----+-----+
| NAME                                     | ENABLED | TIMED |
+-----+-----+-----+
...
| wait/synch/mutex/sql/LOCK_global_read_lock | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db         | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager        | YES     | YES   |
...

```

The `setup_instruments` table provides the most basic form of control over event production. To further refine event production based on the type of object or thread being monitored, other tables may be used as described in [Section 5.3, “Event Pre-Filtering”](#).

- Performance Schema tables are the destinations for events and consume events. The `setup_consumers` table lists the types of consumers to which event information can be sent and whether they are enabled:

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

```

events_statements_current	YES
events_statements_history	NO
events_statements_history_long	NO
events_waits_current	NO
events_waits_history	NO
events_waits_history_long	NO
global_instrumentation	YES
thread_instrumentation	YES
statements_digest	YES

Filtering can be done at different stages of performance monitoring:

- **Pre-filtering.** This is done by modifying Performance Schema configuration so that only certain types of events are collected from producers, and collected events update only certain consumers. To do this, enable or disable instruments or consumers. Pre-filtering is done by the Performance Schema and has a global effect that applies to all users.

Reasons to use pre-filtering:

- To reduce overhead. Performance Schema overhead should be minimal even with all instruments enabled, but perhaps you want to reduce it further. Or you do not care about timing events and want to disable the timing code to eliminate timing overhead.
- To avoid filling the current-events or history tables with events in which you have no interest. Pre-filtering leaves more “room” in these tables for instances of rows for enabled instrument types. If you enable only file instruments with pre-filtering, no rows are collected for nonfile instruments. With post-filtering, nonfile events are collected, leaving fewer rows for file events.
- To avoid maintaining some kinds of event tables. If you disable a consumer, the server does not spend time maintaining destinations for that consumer. For example, if you do not care about event histories, you can disable the history table consumers to improve performance.
- **Post-filtering.** This involves the use of `WHERE` clauses in queries that select information from Performance Schema tables, to specify which of the available events you want to see. Post-filtering is performed on a per-user basis because individual users select which of the available events are of interest.

Reasons to use post-filtering:

- To avoid making decisions for individual users about which event information is of interest.
- To use the Performance Schema to investigate a performance issue when the restrictions to impose using pre-filtering are not known in advance.

The following sections provide more detail about pre-filtering and provide guidelines for naming instruments or consumers in filtering operations. For information about writing queries to retrieve information (post-filtering), see [Chapter 6, Performance Schema Queries](#).

5.3 Event Pre-Filtering

Pre-filtering is done by the Performance Schema and has a global effect that applies to all users. Pre-filtering can be applied to either the producer or consumer stage of event processing:

- To configure pre-filtering at the producer stage, several tables can be used:
 - `setup_instruments` indicates which instruments are available. An instrument disabled in this table produces no events regardless of the contents of the other production-related setup tables. An instrument enabled in this table is permitted to produce events, subject to the contents of the other tables.
 - `setup_objects` controls whether the Performance Schema monitors particular table 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.
- Modifications to the `setup_actors` table affect only foreground threads created subsequent to the modification, not existing threads.

When you change the monitoring configuration, the Performance Schema does not flush the history tables. Events already collected remain in the current-events and history tables until displaced by newer events. If you disable instruments, you might need to wait a while before events for them are displaced by newer events of interest. Alternatively, use `TRUNCATE TABLE` to empty the history tables.

After making instrumentation changes, you might want to truncate the summary tables. Generally, the effect is to reset the summary columns to 0 or `NULL`, not to remove rows. This enables you to clear collected values and restart aggregation. That might be useful, for example, after you have made a runtime configuration change. Exceptions to this truncation behavior are noted in individual summary table sections.

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

5.4 Pre-Filtering by Instrument

The `setup_instruments` table lists the available instruments:

```
mysql> SELECT * FROM performance_schema.setup_instruments;
```

NAME	ENABLED	TIMED
...		
stage/sql/end	NO	NO
stage/sql/executing	NO	NO
stage/sql/init	NO	NO
stage/sql/insert	NO	NO
...		
statement/sql/load	YES	YES
statement/sql/grant	YES	YES
statement/sql/check	YES	YES
statement/sql/flush	YES	YES
...		
wait/synch/mutex/sql/LOCK_global_read_lock	YES	YES
wait/synch/mutex/sql/LOCK_global_system_variables	YES	YES
wait/synch/mutex/sql/LOCK_lock_db	YES	YES
wait/synch/mutex/sql/LOCK_manager	YES	YES
...		
wait/synch/rwlock/sql/LOCK_grant	YES	YES
wait/synch/rwlock/sql/LOGGER::LOCK_logger	YES	YES
wait/synch/rwlock/sql/LOCK_sys_init_connect	YES	YES
wait/synch/rwlock/sql/LOCK_sys_init_slave	YES	YES
...		
wait/io/file/sql/binlog	YES	YES
wait/io/file/sql/binlog_index	YES	YES
wait/io/file/sql/casetest	YES	YES
wait/io/file/sql/dbopt	YES	YES

...

To control whether an instrument is enabled, set its `ENABLED` column to `YES` or `NO`. To configure whether to collect timing information for an enabled instrument, set its `TIMED` value to `YES` or `NO`. Setting the `TIMED` column affects Performance Schema table contents as described in [Section 5.1, “Performance Schema Event Timing”](#).

Modifications to most `setup_instruments` rows affect monitoring immediately. For some instruments, modifications are effective only at server startup; changing them at runtime has no effect. This affects primarily mutexes, conditions, and rwlocks in the server, although there may be other instruments for which this is true.

The `setup_instruments` table provides the most basic form of control over event production. To further refine event production based on the type of object or thread being monitored, other tables may be used as described in [Section 5.3, “Event Pre-Filtering”](#).

The following examples demonstrate possible operations on the `setup_instruments` table. These changes, like other pre-filtering operations, affect all users. Some of these queries use the `LIKE` operator and a pattern match instrument names. For additional information about specifying patterns to select instruments, see [Section 5.9, “Naming Instruments or Consumers for Filtering Operations”](#).

- Disable all instruments:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO';
```

Now no events will be collected.

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

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO'
WHERE NAME LIKE 'wait/io/file/%';
```

- Disable only file instruments, enable all other instruments:

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

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

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

- Disable a specific instrument:

```
UPDATE performance_schema.setup_instruments
SET ENABLED = 'NO'
WHERE NAME = 'wait/synch/mutex/mysys/TMPDIR_mutex';
```

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

```
UPDATE performance_schema.setup_instruments
SET ENABLED = IF(ENABLED = 'YES', 'NO', 'YES')
WHERE NAME = 'wait/synch/mutex/mysys/TMPDIR_mutex';
```

- Disable timing for all events:

```
UPDATE performance_schema.setup_instruments
SET TIMED = 'NO';
```

5.5 Pre-Filtering by Object

The `setup_objects` table controls whether the Performance Schema monitors particular table objects. The initial `setup_objects` contents look like this:

```
mysql> SELECT * FROM performance_schema.setup_objects;
```

OBJECT_TYPE	OBJECT_SCHEMA	OBJECT_NAME	ENABLED	TIMED
TABLE	mysql	%	NO	NO
TABLE	performance_schema	%	NO	NO
TABLE	information_schema	%	NO	NO
TABLE	%	%	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 table name, or `'%'` to match any name.

The `ENABLED` column indicates whether matching objects are monitored, and `TIMED` indicates whether to collect timing information. Setting the `TIMED` column affects Performance Schema table contents as described in [Section 5.1, “Performance Schema Event Timing”](#).

The effect of the default object configuration is to instrument all 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 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`.

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 a table-related instrument in `setup_instruments` has an `ENABLED` value of `NO`, events for the object are not monitored. If the `ENABLED` value is `YES`, event monitoring occurs according to the `ENABLED` value in the relevant `setup_objects` row:

- `db1.t1` events are monitored
- `db1.t2` events are not monitored
- `db2.t3` events are monitored
- `db3.t4` events are not monitored
- `db4.t5` events are monitored

Similar logic applies for combining the `TIMED` columns from the `setup_instruments` and `setup_objects` tables to determine whether to collect event timing information.

If a persistent table and a temporary table have the same name, matching against `setup_objects` rows occurs the same way for both. It is not possible to enable monitoring for one table but not the other. However, each table is instrumented separately.

5.6 Pre-Filtering by Thread

The `threads` table contains a row for each server thread. Each row contains information about a thread and indicates whether monitoring is enabled for it. For the Performance Schema to monitor a thread, these things must be true:

- The `thread_instrumentation` consumer in the `setup_consumers` table must be `YES`.
- The `threads.INSTRUMENTED` column must be `YES`.
- Monitoring occurs only for those thread events produced from instruments that are enabled in the `setup_instruments` table.

The `INSTRUMENTED` column in the `threads` table indicates the monitoring state for each thread. For foreground threads (resulting from client connections), the initial `INSTRUMENTED` value is determined by whether the user account associated with the thread matches any row in the `setup_actors` table.

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

The initial `setup_actors` contents look like this:

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

The `HOST` and `USER` columns should contain a literal host or user name, or `'%'` to match any name.

The Performance Schema uses the `HOST` and `USER` columns to match each new foreground thread. (`ROLE` is unused.) The `INSTRUMENTED` value for the thread becomes `YES` if any row matches, `NO`

otherwise. This enables instrumenting to be applied selectively per host, user, or combination of host and user.

By default, monitoring is 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 delete this row because it matches any connection.

Suppose that you modify `setup_actors` as follows:

```
TRUNCATE TABLE performance_schema.setup_actors;
```

Now `setup_actors` is empty and there are no rows that could match incoming connections. Consequently, the Performance Schema sets the `INSTRUMENTED` column to `NO` for all new foreground threads.

Suppose that you further modify `setup_actors`:

```
INSERT INTO performance_schema.setup_actors
(HOST,USER,ROLE) VALUES('localhost','joe','%');
INSERT INTO performance_schema.setup_actors
(HOST,USER,ROLE) VALUES('%','sam','%');
```

Now the Performance Schema determines how to set the `INSTRUMENTED` value for new connection threads as follows:

- If `joe` connects from the local host, the connection matches the first inserted row.
- If `joe` connects from any other host, there is no match.
- If `sam` connects from any host, the connection matches the second inserted row.
- For any other connection, there is no match.

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` column of `threads` table rows.

5.7 Pre-Filtering by Consumer

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

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

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

Modifications to the `setup_consumers` table affect monitoring immediately.

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

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

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

- Destinations associated with a consumer receive no events unless the Performance Schema checks the consumer and the consumer is enabled.
- A consumer is checked only if all consumers it depends on (if any) are enabled.
- If a consumer is not checked, or is checked but is disabled, other consumers that depend on it are not checked.
- Dependent consumers may have their own dependent consumers.
- If an event would not be sent to any destination, the Performance Schema does not produce it.

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

- [Global and Thread Consumers](#)
- [Wait Event Consumers](#)
- [Stage Event Consumers](#)
- [Statement Event Consumers](#)
- [Statement Digest Consumer](#)

Global and Thread Consumers

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

Wait Event Consumers

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

- `events_waits_current`, if `NO`, disables collection of individual wait events in the `events_waits_current` table. If `YES`, it enables wait event collection and the Performance Schema checks the `events_waits_history` and `events_waits_history_long` consumers.

- `events_waits_history` is not checked if `event_waits_current` is `NO`. Otherwise, an `events_waits_history` value of `NO` or `YES` disables or enables collection of wait events in the `events_waits_history` table.
- `events_waits_history_long` is not checked if `event_waits_current` is `NO`. Otherwise, an `events_waits_history_long` value of `NO` or `YES` disables or enables collection of wait events in the `events_waits_history_long` table.

Stage Event Consumers

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

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

Statement Event Consumers

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

- `events_statements_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.

Statement Digest Consumer

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

For more information about statement digesting, see [Performance Schema Statement Digests](#).

5.8 Example Consumer Configurations

The consumer settings in the `setup_consumers` table form a hierarchy from higher levels to lower. The following discussion describes how consumers work, showing specific configurations and their effects as consumer settings are enabled progressively from high to low. The consumer values shown

are representative. The general principles described here apply to other consumer values that may be available.

The configuration descriptions occur in order of increasing functionality and overhead. If you do not need the information provided by enabling lower-level settings, disable them 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
statements_digest
```

Note

In the consumer hierarchy, the consumers for waits, stages, and statements 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.

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

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

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

- [No Instrumentation](#)
- [Global Instrumentation Only](#)
- [Global and Thread Instrumentation Only](#)
- [Global, Thread, and Current-Event Instrumentation](#)
- [Global, Thread, Current-Event, and Event-History instrumentation](#)

No Instrumentation

Server configuration state:

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

In this configuration, nothing is instrumented.

Setup elements checked:

- Table `setup_consumers`, consumer `global_instrumentation`

Output tables maintained:

- None

Global Instrumentation Only

Server configuration state:

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

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

Additional setup elements checked, relative to the preceding configuration:

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

Global and Thread Instrumentation Only

Server configuration state:

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

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`
- 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`; and `yyy` is `thread`, `user`, `host`, `account`

Global, Thread, and Current-Event Instrumentation

Server configuration state:

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

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`
- Consumers `events_xxx_history_long`, where `xxx` is `waits`, `stages`, `statements`

Additional output tables maintained, relative to the preceding configuration:

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

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

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

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

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

Event-history tables maintained for this configuration:

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

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

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

Event-history tables maintained for this configuration:

- `events_XXX_history_long`, where `XXX` is `waits`, `stages`, `statements`

This configuration collects event history per thread and globally:

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

Event-history tables maintained for this configuration:

- `events_XXX_history`, where `XXX` is `waits`, `stages`, `statements`
- `events_XXX_history_long`, where `XXX` is `waits`, `stages`, `statements`

5.9 Naming Instruments or Consumers for Filtering Operations

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

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

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

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

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

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

A pattern of `%/file/%` 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:

```
SELECT NAME FROM performance_schema.setup_instruments
```

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

For information about the types of names that are supported, see [Chapter 7, Performance Schema Instrument Naming Conventions](#).

5.10 Determining What Is Instrumented

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

```
mysql> SELECT * FROM performance_schema.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.

Chapter 6 Performance Schema Queries

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

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

```
mysql> SELECT THREAD_ID, NUMBER_OF_BYTES
        FROM performance_schema.events_waits_history
        WHERE EVENT_NAME LIKE 'wait/io/file/%'
        AND NUMBER_OF_BYTES IS NOT NULL;
```

THREAD_ID	NUMBER_OF_BYTES
11	66
11	47
11	139
5	24
5	834

Chapter 7 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 Instrument Components](#)
- [Stage Instrument Components](#)
- [Statement Instrument Components](#)
- [Wait Instrument Components](#)

Top-Level Instrument Components

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

- `statement`: An instrumented statement event.
- `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 10.3.5, “The `socket_instances` Table”](#).

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 10.6, “Performance Schema Statement Event Tables”](#).
- `statement/com`: An instrumented command operation. These have names corresponding to `COM_xxx` operations (see the `mysql_com.h` header file and `sql/sql_parse.cc`). For example, the `statement/com/Connect` and `statement/com/Init DB` instruments correspond to the `COM_CONNECT` and `COM_INIT_DB` commands.
- `statement/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/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.

Chapter 8 Performance Schema Status Monitoring

There are several status variables associated with the Performance Schema:

```
mysql> SHOW STATUS LIKE 'perf%';
```

Variable_name	Value
Performance_schema_accounts_lost	0
Performance_schema_cond_classes_lost	0
Performance_schema_cond_instances_lost	0
Performance_schema_digest_lost	0
Performance_schema_file_classes_lost	0
Performance_schema_file_handles_lost	0
Performance_schema_file_instances_lost	0
Performance_schema_hosts_lost	0
Performance_schema_locker_lost	0
Performance_schema_mutex_classes_lost	0
Performance_schema_mutex_instances_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.row_size
  Status: 76
***** 4. row *****
  Type: performance_schema
  Name: events_waits_history.row_count
  Status: 10000
***** 5. row *****
  Type: performance_schema
  Name: events_waits_history.memory
  Status: 760000
...
***** 57. row *****
  Type: performa
```

```
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 9 Performance Schema General Table Characteristics

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

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

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

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

Summary tables can be truncated with `TRUNCATE TABLE`. Generally, the effect is to reset the summary columns to 0 or `NULL`, not to remove rows. This enables you to clear collected values and restart aggregation. That might be useful, for example, after you have made a runtime configuration change. Exceptions to this truncation behavior are noted in individual summary table sections.

Privileges are as for other databases and tables:

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

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

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

Instead, grant exactly the desired privileges:

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

Chapter 10 Performance Schema Table Descriptions

Table of Contents

10.1 Performance Schema Table Index	48
10.2 Performance Schema Setup Tables	49
10.2.1 The setup_actors Table	50
10.2.2 The setup_consumers Table	50
10.2.3 The setup_instruments Table	51
10.2.4 The setup_objects Table	52
10.2.5 The setup_timers Table	53
10.3 Performance Schema Instance Tables	54
10.3.1 The cond_instances Table	54
10.3.2 The file_instances Table	55
10.3.3 The mutex_instances Table	55
10.3.4 The rwlock_instances Table	56
10.3.5 The socket_instances Table	57
10.4 Performance Schema Wait Event Tables	59
10.4.1 The events_waits_current Table	61
10.4.2 The events_waits_history Table	63
10.4.3 The events_waits_history_long Table	63
10.5 Performance Schema Stage Event Tables	64
10.5.1 The events_stages_current Table	66
10.5.2 The events_stages_history Table	67
10.5.3 The events_stages_history_long Table	67
10.6 Performance Schema Statement Event Tables	68
10.6.1 The events_statements_current Table	71
10.6.2 The events_statements_history Table	75
10.6.3 The events_statements_history_long Table	75
10.7 Performance Schema Connection Tables	75
10.7.1 The accounts Table	77
10.7.2 The hosts Table	78
10.7.3 The users Table	78
10.8 Performance Schema Connection Attribute Tables	78
10.8.1 The session_account_connect_attrs Table	80
10.8.2 The session_connect_attrs Table	81
10.9 Performance Schema Summary Tables	81
10.9.1 Wait Event Summary Tables	82
10.9.2 Stage Summary Tables	84
10.9.3 Statement Summary Tables	85
10.9.4 Object Wait Summary Table	87
10.9.5 File I/O Summary Tables	88
10.9.6 Table I/O and Lock Wait Summary Tables	89
10.9.7 Socket Summary Tables	92
10.10 Performance Schema Miscellaneous Tables	93
10.10.1 The host_cache Table	93
10.10.2 The performance_timers Table	96
10.10.3 The threads Table	97

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, and [events_statements_current](#) for statement 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 and statement histories.

To change the sizes of the history tables, set the appropriate system variables at server startup. For example, to set the sizes of the wait event history tables, set `performance_schema_events_waits_history_size` and `performance_schema_events_waits_history_long_size`.

- Summary tables. These tables contain information aggregated over groups of events, including those that have been discarded from the history tables.
- Instance tables. These tables document what types of objects are instrumented. An instrumented object, when used by the server, produces an event. These tables provide event names and explanatory notes or status information.
- Miscellaneous tables. These do not fall into any of the other table groups.

10.1 Performance Schema Table Index

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

Table 10.1 Performance Schema Tables

Table Name	Description
<code>accounts</code>	Connection statistics per client account
<code>cond_instances</code>	synchronization object instances
<code>events_stages_current</code>	Current stage events
<code>events_stages_history</code>	Most recent stage events per thread
<code>events_stages_history_long</code>	Most recent stage events overall
<code>events_stages_summary_by_account_by_event_name</code>	Stage events per account and event name
<code>events_stages_summary_by_host_by_event_name</code>	Stage events per host name and event name
<code>events_stages_summary_by_thread_by_event_name</code>	Stage waits per thread and event name
<code>events_stages_summary_by_user_by_event_name</code>	Stage events per user name and event name
<code>events_stages_summary_global_by_event_name</code>	Stage waits per event name
<code>events_statements_current</code>	Current statement events
<code>events_statements_history</code>	Most recent statement events per thread
<code>events_statements_history_long</code>	Most recent statement events overall
<code>events_statements_summary_by_account_by_event_name</code>	Statement events per account and event name
<code>events_statements_summary_by_digest</code>	Statement events per schema and digest value
<code>events_statements_summary_by_host_by_event_name</code>	Statement events per host name and event name
<code>events_statements_summary_by_thread_by_event_name</code>	Statement events per thread and event name
<code>events_statements_summary_by_user_by_event_name</code>	Statement events per user name and event name
<code>events_statements_summary_global_by_event_name</code>	Statement events per event name
<code>events_waits_current</code>	Current wait events
<code>events_waits_history</code>	Most recent wait events per thread

Table Name	Description
events_waits_history_long	Most recent wait events overall
events_waits_summary_by_account_by_event_name	Wait events per account and event name
events_waits_summary_by_host_by_event_name	Wait events per host name and event name
events_waits_summary_by_instance	Wait events per instance
events_waits_summary_by_thread_by_event_name	Wait events per thread and event name
events_waits_summary_by_user_by_event_name	Wait events per user name and event name
events_waits_summary_global_by_event_name	Wait events per event name
file_instances	File instances
file_summary_by_event_name	File events per event name
file_summary_by_instance	File events per file instance
host_cache	Information from the internal host cache
hosts	Connection statistics per client host name
mutex_instances	Mutex synchronization object instances
objects_summary_global_by_type	Object summaries
performance_timers	Which event timers are available
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
setup_actors	How to initialize monitoring for new foreground threads
setup_consumers	Consumers for which event information can be stored
setup_instruments	Classes of instrumented objects for which events can be collected
setup_objects	Which objects should be monitored
setup_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
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

10.2 Performance Schema Setup Tables

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

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

These setup tables are available:

- `setup_actors`: How to initialize monitoring for new foreground threads
- `setup_consumers`: The destinations to which event information can be sent and stored
- `setup_instruments`: The classes of instrumented objects for which events can be collected
- `setup_objects`: Which objects should be monitored
- `setup_timers`: The current event timer

10.2.1 The setup_actors Table

The `setup_actors` table contains information that determines whether to enable monitoring 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` column value is used to set the `INSTRUMENTED` column of the `threads` table row for the thread. This enables instrumenting to be applied selectively per host, user, or account (user and host combination). If there is no match, the `INSTRUMENTED` column for the thread is set to `NO`.

For background threads, there is no associated user. `INSTRUMENTED` is `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 is enabled by default for all foreground threads:

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

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

Modifications to the `setup_actors` table affect only foreground threads created subsequent to the modification, not existing threads. To affect existing threads, modify the `INSTRUMENTED` column 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.

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

10.2.2 The setup_consumers Table

The `setup_consumers` table lists the types of consumers for which event information can be stored and which are enabled:

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

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

Modifications to the `setup_consumers` table affect monitoring immediately.

The `setup_consumers` table has these columns:

- `NAME`

The consumer name.

- `ENABLED`

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

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

10.2.3 The setup_instruments Table

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

```
mysql> SELECT * FROM performance_schema.setup_instruments;
+-----+-----+-----+
| NAME                                | ENABLED | TIMED |
+-----+-----+-----+
| ...
| stage/sql/end                       | NO      | NO    |
| stage/sql/executing                 | NO      | NO    |
| stage/sql/init                      | NO      | NO    |
| stage/sql/insert                    | NO      | NO    |
| ...
| statement/sql/load                  | YES     | YES   |
| statement/sql/grant                 | YES     | YES   |
| statement/sql/check                 | YES     | YES   |
| statement/sql/flush                 | YES     | YES   |
| ...
| wait/synch/mutex/sql/LOCK_global_read_lock | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db   | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager   | YES     | YES   |
+-----+-----+-----+
```

```

...
| wait/synch/rwlock/sql/LOCK_grant          | YES | YES |
| wait/synch/rwlock/sql/LOGGER::LOCK_logger | YES | YES |
| wait/synch/rwlock/sql/LOCK_sys_init_connect | YES | YES |
| wait/synch/rwlock/sql/LOCK_sys_init_slave  | YES | YES |
...
| wait/io/file/sql/binlog                   | YES | YES |
| wait/io/file/sql/binlog_index             | YES | YES |
| wait/io/file/sql/casetest                 | YES | YES |
| wait/io/file/sql/dbopt                    | YES | YES |
...

```

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

Modifications to most `setup_instruments` rows affect monitoring immediately. For some instruments, modifications are effective only at server startup; changing them at runtime has no effect. This affects primarily mutexes, conditions, and rwlocks in the server, although there may be other instruments for which this is true.

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

The `setup_instruments` table has these columns:

- NAME**

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

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

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

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.

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

10.2.4 The setup_objects Table

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

The initial `setup_objects` contents look like this:

```

mysql> SELECT * FROM performance_schema.setup_objects;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

The setup_timers Table

OBJECT_TYPE	OBJECT_SCHEMA	OBJECT_NAME	ENABLED	TIMED
TABLE	mysql	%	NO	NO
TABLE	performance_schema	%	NO	NO
TABLE	information_schema	%	NO	NO
TABLE	%	%	YES	YES

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

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

The effect of the default object configuration is to instrument all tables except those in the `mysql`, `INFORMATION_SCHEMA`, and `performance_schema` databases. (Tables in the `INFORMATION_SCHEMA` database are not instrumented regardless of the contents of `setup_objects`; the row for `information_schema.%` simply makes this default explicit.)

When the Performance Schema checks for a match in `setup_objects`, it tries to find more specific matches first. For example, with a table `db1.t1`, it looks for a match for `'db1'` and `'t1'`, then for `'db1'` and `'%'`, then for `'%'` and `'%'`. The order in which matching occurs matters because different matching `setup_objects` rows can have different `ENABLED` and `TIMED` values.

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

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

The `setup_objects` table has these columns:

- `OBJECT_TYPE`

The type of object to instrument. This is always `'TABLE'` (base 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. The value is `YES` or `NO`. This column can be modified.

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

10.2.5 The setup_timers Table

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

```
mysql> SELECT * FROM performance_schema.setup_timers;
+-----+-----+
| NAME      | TIMER_NAME |
+-----+-----+
| idle      | MICROSECOND |
| wait      | CYCLE       |
| stage     | NANOSECOND  |
| statement | 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 5.1, “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.

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

10.3 Performance Schema Instance Tables

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

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

These tables list instrumented synchronization objects, files, and connections. There are three types of synchronization objects: `cond`, `mutex`, and `rwlock`. Each instance table has an `EVENT_NAME` or `NAME` column to indicate the instrument associated with each row. Instrument names may have multiple parts and form a hierarchy, as discussed in [Chapter 7, Performance Schema Instrument Naming Conventions](#).

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

10.3.1 The `cond_instances` Table

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

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

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

10.3.2 The file_instances Table

The `file_instances` table lists all the files seen by the Performance Schema when executing file I/O instrumentation. If a file on disk has never been opened, it 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`.

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

10.3.3 The mutex_instances Table

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

When two threads executing in the server (for example, two user sessions executing a query simultaneously) do need to access the same resource (a file, a buffer, or some piece of data), these two threads 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`.

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

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

- The `setup_instruments` table lists the name of the instrumentation point, with the prefix `wait/synch/mutex/`.
- When some code creates a mutex, a row is added to the `mutex_instances` table. The `OBJECT_INSTANCE_BEGIN` column is a property that uniquely identifies the mutex.
- When a thread attempts to lock a mutex, the `events_waits_current` table shows a row for that thread, indicating that it is waiting on a mutex (in the `EVENT_NAME` column), and indicating which mutex is waited on (in the `OBJECT_INSTANCE_BEGIN` column).
- When a thread succeeds in locking a mutex:
 - `events_waits_current` shows that the wait on the mutex is completed (in the `TIMER_END` and `TIMER_WAIT` columns)
 - The completed wait event is added to the `events_waits_history` and `events_waits_history_long` tables
 - `mutex_instances` shows that the mutex is now owned by the thread (in the `THREAD_ID` column).
- When a thread unlocks a mutex, `mutex_instances` shows that the mutex now has no owner (the `THREAD_ID` column is `NULL`).
- When a mutex object is destroyed, the corresponding row is removed from `mutex_instances`.

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

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

10.3.4 The rwlock_instances Table

The `rwlock_instances` table lists all the `rwlock` instances (read write locks) 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) or exclusive (only one thread can have a write lock at a given time).

Depending on how many threads are requesting a lock, and the nature of the locks requested, access can be either granted in shared mode, granted in 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.

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

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

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

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

10.3.5 The `socket_instances` Table

The `socket_instances` table provides a real-time snapshot of the active connections to the MySQL server. The table contains one row per TCP/IP or Unix socket file connection. Information available in this table provides a real-time snapshot of the active connections to the server. (Additional information is available in socket summary tables, including network activity such as socket operations and number of bytes transmitted and received; see [Section 10.9.7, “Socket Summary Tables”](#)).

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

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

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

The `socket_instances` table has these columns:

- `EVENT_NAME`

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

- `OBJECT_INSTANCE_BEGIN`

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

- `THREAD_ID`

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

- `SOCKET_ID`

The internal file handle assigned to the socket.

- `IP`

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

- `PORT`

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

- `STATE`

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

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

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

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

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

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

10.4 Performance Schema Wait Event Tables

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

These tables store wait events:

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

The following sections describe the wait event tables. There are also summary tables that aggregate information about wait events; see [Section 10.9.1, “Wait Event Summary Tables”](#).

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

Configuring Wait Event Collection

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

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

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

```
mysql> SELECT * FROM performance_schema.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 performance_schema.setup_instruments
      WHERE NAME LIKE 'wait/io/socket/%';
+-----+-----+-----+
| NAME                                     | ENABLED | TIMED |
+-----+-----+-----+
| wait/io/socket/sql/server_tcpip_socket  | NO      | NO    |
| wait/io/socket/sql/server_unix_socket   | NO      | NO    |
| wait/io/socket/sql/client_connection    | NO      | NO    |
+-----+-----+-----+
```

The wait consumers are disabled by default:

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

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

- Enable:

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

- Disable:

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

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

- Enable:

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

- Disable:

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

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

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 performance_schema.setup_timers
      WHERE NAME = 'wait';
+-----+-----+
| NAME | TIMER_NAME |
+-----+-----+
| wait | CYCLE      |
+-----+-----+
```

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

```
UPDATE performance_schema.setup_timers
SET TIMER_NAME = 'NANOSECOND'
WHERE NAME = 'wait';
```

For additional information about configuring event collection, see [Chapter 4, Performance Schema Startup Configuration](#), and [Chapter 5, Performance Schema Runtime Configuration](#).

10.4.1 The events_waits_current Table

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

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

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

For information about configuring whether to collect wait events, see [Section 10.4, "Performance Schema Wait Event Tables"](#).

The `events_waits_current` table has these columns:

- `THREAD_ID`, `EVENT_ID`

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

- `END_EVENT_ID`

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

- `EVENT_NAME`

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

- `SOURCE`

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

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

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

If an event has not finished, `TIMER_END` and `TIMER_WAIT` are `NULL` before MySQL 5.6.26. As of 5.6.26, `TIMER_END` is the current timer value and `TIMER_WAIT` is the time elapsed so far (`TIMER_END - TIMER_START`).

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

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

- `SPINS`

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

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

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

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

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

For a file I/O object:

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

For a socket object:

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

For a table I/O object:

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

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

- `INDEX_NAME`

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

- `NESTING_EVENT_ID`

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

- `NESTING_EVENT_TYPE`

The nesting event type. The value is `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`.

- `FLAGS`

Reserved for future use.

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

10.4.2 The events_waits_history Table

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

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

The `events_waits_history` table has the same columns as `events_waits_current`. See [Section 10.4.1, “The events_waits_current Table”](#).

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

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

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

10.4.3 The events_waits_history_long Table

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

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

The `events_waits_history_long` table has the same columns as `events_waits_current`. See [Section 10.4.1, “The events_waits_current Table”](#).

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

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

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

10.5 Performance Schema Stage Event Tables

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

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

These tables store stage events:

- `events_stages_current`: The current stage event for each thread.
- `events_stages_history`: The most recent stage events that have ended per thread.
- `events_stages_history_long`: The most recent stage events that have ended globally (across all threads).

The following sections describe the stage event tables. There are also summary tables that aggregate information about stage events; see [Section 10.9.2, “Stage Summary Tables”](#).

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

Configuring Stage Event Collection

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

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

The stage instruments are disabled by default. For example:

```
mysql> SELECT *
      FROM performance_schema.setup_instruments
      WHERE NAME RLIKE 'stage/sql/[a-c]';
```

NAME	ENABLED	TIMED
stage/sql/After create	NO	NO
stage/sql/allocating local table	NO	NO
stage/sql/altering table	NO	NO
stage/sql/committing alter table to storage engine	NO	NO
stage/sql/Changing master	NO	NO
stage/sql/Checking master version	NO	NO
stage/sql/checking permissions	NO	NO
stage/sql/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 delayed handler	NO	NO
stage/sql/Creating sort index	NO	NO
stage/sql/creating table	NO	NO
stage/sql/Creating tmp table	NO	NO

The stage consumers are disabled by default:

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

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

- Enable:

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

- Disable:

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

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

- Enable:

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

- Disable:

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

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

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 performance_schema.setup_timers
      WHERE NAME = 'stage';
+-----+-----+
| NAME | TIMER_NAME |
+-----+-----+
```

```
| stage | NANOSECOND |
+-----+-----+
```

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

```
UPDATE performance_schema.setup_timers
SET TIMER_NAME = 'MICROSECOND'
WHERE NAME = 'stage';
```

For additional information about configuring event collection, see [Chapter 4, Performance Schema Startup Configuration](#), and [Chapter 5, Performance Schema Runtime Configuration](#).

10.5.1 The events_stages_current Table

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

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

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

For information about configuring whether to collect stage events, see [Section 10.5, "Performance Schema Stage Event Tables"](#).

The `events_stages_current` table has these columns:

- `THREAD_ID`, `EVENT_ID`

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

- `END_EVENT_ID`

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

- `EVENT_NAME`

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

- `SOURCE`

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

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

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

If an event has not finished, `TIMER_END` and `TIMER_WAIT` are `NULL` before MySQL 5.6.26. As of 5.6.26, `TIMER_END` is the current timer value and `TIMER_WAIT` is the time elapsed so far (`TIMER_END - TIMER_START`).

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

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

- `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 `STATEMENT`, `STAGE`, or `WAIT`.

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

10.5.2 The events_stages_history Table

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

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

The `events_stages_history` table has the same columns as `events_stages_current`. See [Section 10.5.1, “The events_stages_current Table”](#).

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

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

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

10.5.3 The events_stages_history_long Table

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

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

The `events_stages_history_long` table has the same columns as `events_stages_current`. See [Section 10.5.1, “The events_stages_current Table”](#).

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

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

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

10.6 Performance Schema Statement Event Tables

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

These tables store statement events:

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

The following sections describe the statement event tables. There are also summary tables that aggregate information about statement events; see [Section 10.9.3, “Statement Summary Tables”](#).

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

Configuring Statement Event Collation

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

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

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

```
mysql> SELECT *
      FROM performance_schema.setup_instruments
      WHERE NAME LIKE 'statement/%';
```

NAME	ENABLED	TIMED
statement/sql/select	YES	YES
statement/sql/create_table	YES	YES
statement/sql/create_index	YES	YES
...		
statement/sp/stmt	YES	YES
statement/sp/set	YES	YES
statement/sp/set_trigger_field	YES	YES
statement/scheduler/event	YES	YES
statement/com/Sleep	YES	YES
statement/com/Quit	YES	YES
statement/com/Init DB	YES	YES
...		
statement/abstract/Query	YES	YES
statement/abstract/new_packet	YES	YES
statement/abstract/relay_log	YES	YES

```
mysql> SELECT *
      FROM performance_schema.setup_consumers
      WHERE NAME LIKE '%statements%';
```

NAME	ENABLED
events_statements_current	YES
events_statements_history	NO
events_statements_history_long	NO
statements_digest	YES

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

- Enable:

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

- Disable:

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

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

- Enable:

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

- Disable:

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

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

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 performance_schema.setup_timers
WHERE NAME = 'statement';
+-----+-----+
| NAME      | TIMER_NAME |
+-----+-----+
| statement | NANOSECOND |
+-----+-----+
```

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

```
UPDATE performance_schema.setup_timers
SET TIMER_NAME = 'MICROSECOND'
WHERE NAME = 'statement';
```

For additional information about configuring event collection, see [Chapter 4, Performance Schema Startup Configuration](#), and [Chapter 5, Performance Schema Runtime Configuration](#).

Statement Monitoring

Statement monitoring begins from the moment the server sees that activity is requested on a thread, to the moment when all activity has ceased. Typically, this means from the time the server gets the first packet from the client to the time the server has finished sending the response. Monitoring occurs only for top-level statements. Statements within stored programs and subqueries are not seen separately.

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

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

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

The abstract instrument names in the preceding discussion are as of MySQL 5.6.15. In earlier 5.6 versions, there was some renaming before those names were settled on:

- `statement/abstract/new_packet` was `statement/com/` in MySQL 5.6.14, `statement/com/new_packet` in MySQL 5.6.13, and `statement/com/` before that.
- `statement/abstract/Query` was `statement/com/Query` before MySQL 5.6.15.
- `statement/abstract/relay_log` was `statement/rpl/relay_log` from MySQL 5.6.13 to 5.6.14 and did not exist before that.

10.6.1 The `events_statements_current` Table

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

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

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

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

The `events_statements_current` table has these columns:

- `THREAD_ID`, `EVENT_ID`

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

- `END_EVENT_ID`

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

- `EVENT_NAME`

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

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

- `SOURCE`

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

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

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

If an event has not finished, `TIMER_END` and `TIMER_WAIT` are `NULL` before MySQL 5.6.26. As of 5.6.26, `TIMER_END` is the current timer value and `TIMER_WAIT` is the time elapsed so far (`TIMER_END - TIMER_START`).

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

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

- `LOCK_TIME`

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

- `SQL_TEXT`

The text of the SQL statement. For a command not associated with an SQL statement, the value is `NULL`. The maximum space available for statement display is 1024 bytes.

- `DIGEST`

The statement digest MD5 value as a string of 32 hexadecimal characters, or `NULL` if the `statements_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 `statements_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 per session for digest value storage. However, the display length of statement digests may be longer than the available buffer size due to encoding of statement 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.

`performance_schema_max_digest_length` was added in MySQL 5.6.26. In MySQL 5.6.24 and 5.6.25, `performance_schema_max_digest_length` is not available and the `max_digest_length` value determines the maximum number of bytes available for digest value storage. Before MySQL 5.6.24, neither `max_digest_length` nor `performance_schema_max_digest_length` are available and a fixed maximum of 1024 bytes is available for digest value storage.

- `CURRENT_SCHEMA`

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

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

Reserved. Always `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 if 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`

Reserved. Always `NULL`.

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

10.6.2 The events_statements_history Table

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

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

The `events_statements_history` table has the same columns as `events_statements_current`. See [Section 10.6.1, “The events_statements_current Table”](#).

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

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

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

10.6.3 The events_statements_history_long Table

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

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

The `events_statements_history_long` table has the same columns as `events_statements_current`. See [Section 10.6.1, “The events_statements_current Table”](#).

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

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

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

10.7 Performance Schema Connection Tables

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

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

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

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

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

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

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

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

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

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

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

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

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

TABLE_NAME
events_stages_summary_by_account_by_event_name
events_stages_summary_by_host_by_event_name
events_stages_summary_by_user_by_event_name
events_statements_summary_by_account_by_event_name
events_statements_summary_by_host_by_event_name
events_statements_summary_by_user_by_event_name
events_waits_summary_by_account_by_event_name
events_waits_summary_by_host_by_event_name

```
| events_waits_summary_by_user_by_event_name |
+-----+-----+
```

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

- Wait event summaries: [Section 10.9.1, “Wait Event Summary Tables”](#)
- Stage event summaries: [Section 10.9.2, “Stage Summary Tables”](#)
- Statement event summaries: [Section 10.9.3, “Statement Summary Tables”](#)

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

Table 10.2 Implicit Effects of Connection Table Truncation

Truncated Connection Table	Implicitly Truncated Summary Tables
<code>accounts</code>	Tables with names containing <code>_summary_by_account</code> , <code>_summary_by_thread</code>
<code>hosts</code>	Tables with names containing <code>_summary_by_account</code> , <code>_summary_by_host</code> , <code>_summary_by_thread</code>
<code>users</code>	Tables with names containing <code>_summary_by_account</code> , <code>_summary_by_user</code> , <code>_summary_by_thread</code>

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

10.7.1 The accounts Table

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

The `accounts` table has the following columns. For a description of how the Performance Schema maintains rows in this table, including the effect of `TRUNCATE TABLE`, see [Section 10.7, “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.

10.7.2 The hosts Table

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

The `hosts` table has the following columns. For a description of how the Performance Schema maintains rows in this table, including the effect of `TRUNCATE TABLE`, see [Section 10.7, “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.

10.7.3 The users Table

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

The `users` table has the following columns. For a description of how the Performance Schema maintains rows in this table, including the effect of `TRUNCATE TABLE`, see [Section 10.7, “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.

10.8 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 clients define additional attributes:

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

- `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
```

10.8.1 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.

The `session_account_connect_attrs` table contains connection attributes only for sessions open for your own account. To see connection attributes for all sessions, look in the `session_connect_attrs` table. For descriptions of common attributes, see [Section 10.8, “Performance Schema Connection Attribute Tables”](#).

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.

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

10.8.2 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 10.8, “Performance Schema Connection Attribute Tables”](#).

The `session_connect_attrs` table contains connection attributes for all sessions. To see connection attributes only for sessions open 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.

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

10.9 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.

Wait Event Summaries

- `events_waits_summary_by_account_by_event_name`: Wait events per account and event name
- `events_waits_summary_by_host_by_event_name`: Wait events per host name and event name
- `events_waits_summary_by_instance`: Wait events per instance
- `events_waits_summary_by_thread_by_event_name`: Wait events per thread and event name
- `events_waits_summary_by_user_by_event_name`: Wait events per user name and event name
- `events_waits_summary_global_by_event_name`: Wait events per event name

Stage Summaries

- `events_stages_summary_by_account_by_event_name`: Stage events per account and event name
- `events_stages_summary_by_host_by_event_name`: Stage events per host name and event name
- `events_stages_summary_by_thread_by_event_name`: Stage waits per thread and event name

- [events_stages_summary_by_user_by_event_name](#): Stage events per user name and event name
- [events_stages_summary_global_by_event_name](#): Stage waits per event name

Statement Summaries

- [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_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

Object Wait Summaries

- [objects_summary_global_by_type](#): Object summaries

File I/O Summaries

- [file_summary_by_event_name](#): File events per event name
- [file_summary_by_instance](#): File events per file instance

Table I/O and Lock Wait Summaries

- [table_io_waits_summary_by_index_usage](#): Table I/O waits per index
- [table_io_waits_summary_by_table](#): Table I/O waits per table
- [table_lock_waits_summary_by_table](#): Table lock waits per table

Socket Summaries

- [socket_summary_by_instance](#): Socket waits and I/O per instance
- [socket_summary_by_event_name](#): Socket waits and I/O per event name

Each summary table has grouping columns that determine how to group the data to be aggregated, and summary columns that contain the aggregated values. Tables that summarize events in similar ways often have similar sets of summary columns and differ only in the grouping columns used to determine how events are aggregated.

Summary tables can be truncated with `TRUNCATE TABLE`. Generally, the effect is to reset the summary columns to 0 or `NULL`, not to remove rows. This enables you to clear collected values and restart aggregation. That might be useful, for example, after you have made a runtime configuration change. Exceptions to this truncation behavior are noted in individual summary table sections.

10.9.1 Wait Event Summary Tables

The Performance Schema maintains tables for collecting current and recent wait events, and aggregates that information in summary tables. [Section 10.4, “Performance Schema Wait Event Tables”](#) describes the events on which wait summaries are based. See that discussion for information

about the content of wait events, the current and recent wait event tables, and how to control wait event collection, which is disabled by default.

Example wait event summary information:

```
mysql> SELECT *
      FROM performance_schema.events_waits_summary_global_by_event_name\G
...
***** 6. row *****
      EVENT_NAME: wait/synch/mutex/sql/BINARY_LOG::LOCK_index
      COUNT_STAR: 8
      SUM_TIMER_WAIT: 2119302
      MIN_TIMER_WAIT: 196092
      AVG_TIMER_WAIT: 264912
      MAX_TIMER_WAIT: 569421
...
***** 9. row *****
      EVENT_NAME: wait/synch/mutex/sql/hash_filo::lock
      COUNT_STAR: 69
      SUM_TIMER_WAIT: 16848828
      MIN_TIMER_WAIT: 0
      AVG_TIMER_WAIT: 244185
      MAX_TIMER_WAIT: 735345
...
```

Each wait event summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `events_waits_summary_by_account_by_event_name` has `EVENT_NAME`, `USER`, and `HOST` columns. Each row summarizes events for a given account (user and host combination) and event name.
- `events_waits_summary_by_host_by_event_name` has `EVENT_NAME` and `HOST` columns. Each row summarizes events for a given host and event name.
- `events_waits_summary_by_instance` has `EVENT_NAME` and `OBJECT_INSTANCE_BEGIN` columns. Each row summarizes events for a given event name and object. If an instrument is used to create multiple instances, each instance has a unique `OBJECT_INSTANCE_BEGIN` value and is summarized separately in this table.
- `events_waits_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `events_waits_summary_by_user_by_event_name` has `EVENT_NAME` and `USER` columns. Each row summarizes events for a given user and event name.
- `events_waits_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name. An instrument might be used to create multiple instances of the instrumented object. For example, if there is an instrument for a mutex that is created for each connection, there are as many instances as there are connections. The summary row for the instrument summarizes over all these instances.

Each wait event summary table has these summary columns containing aggregated values:

- `COUNT_STAR`
The number of summarized events. This value includes all events, whether timed or nontimed.
- `SUM_TIMER_WAIT`
The total wait time of the summarized timed events. This value is calculated only for timed events because nontimed events have a wait time of `NULL`. The same is true for the other `xxx_TIMER_WAIT` values.

- [MIN_TIMER_WAIT](#)

The minimum wait time of the summarized timed events.

- [AVG_TIMER_WAIT](#)

The average wait time of the summarized timed events.

- [MAX_TIMER_WAIT](#)

The maximum wait time of the summarized timed events.

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

- For summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each wait summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of [events_waits_summary_global_by_event_name](#). For details, see [Section 10.7, “Performance Schema Connection Tables”](#).

10.9.2 Stage Summary Tables

The Performance Schema maintains tables for collecting current and recent stage events, and aggregates that information in summary tables. [Section 10.5, “Performance Schema Stage Event Tables”](#) describes the events on which stage summaries are based. See that discussion for information about the content of stage events, the current and historical stage event tables, and how to control stage event collection, which is disabled by default.

Example stage event summary information:

```
mysql> SELECT *
      FROM performance_schema.events_stages_summary_global_by_event_name\G
...
***** 5. row *****
EVENT_NAME: stage/sql/checking permissions
COUNT_STAR: 57
SUM_TIMER_WAIT: 26501888880
MIN_TIMER_WAIT: 7317456
AVG_TIMER_WAIT: 464945295
MAX_TIMER_WAIT: 12858936792
...
***** 9. row *****
EVENT_NAME: stage/sql/closing tables
COUNT_STAR: 37
SUM_TIMER_WAIT: 662606568
MIN_TIMER_WAIT: 1593864
AVG_TIMER_WAIT: 17907891
MAX_TIMER_WAIT: 437977248
...
```

Each stage summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the [setup_instruments](#) table:

- [events_stages_summary_by_account_by_event_name](#) has `EVENT_NAME`, `USER`, and `HOST` columns. Each row summarizes events for a given account (user and host combination) and event name.
- [events_stages_summary_by_host_by_event_name](#) has `EVENT_NAME` and `HOST` columns. Each row summarizes events for a given host and event name.

- `events_stages_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `events_stages_summary_by_user_by_event_name` has `EVENT_NAME` and `USER` columns. Each row summarizes events for a given user and event name.
- `events_stages_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

Each stage summary table has these summary columns containing aggregated values: `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, and `MAX_TIMER_WAIT`. These columns are analogous to the columns of the same names in the wait event summary tables (see [Section 10.9.1, “Wait Event Summary Tables”](#)), except that the stage summary tables aggregate events from `events_stages_current` rather than `events_waits_current`.

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

- For summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each stage summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_stages_summary_global_by_event_name`. For details, see [Section 10.7, “Performance Schema Connection Tables”](#).

10.9.3 Statement Summary Tables

The Performance Schema maintains tables for collecting current and recent statement events, and aggregates that information in summary tables. [Section 10.6, “Performance Schema Statement Event Tables”](#) describes the events on which statement summaries are based. See that discussion for information about the content of statement events, the current and historical statement event tables, and how to control statement event collection, which is partially disabled by default.

Example statement event summary information:

```
mysql> SELECT *
      FROM performance_schema.events_statements_summary_global_by_event_name\G
***** 1. row *****
      EVENT_NAME: statement/sql/select
      COUNT_STAR: 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
...
```

Each statement summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `events_statements_summary_by_account_by_event_name` has `EVENT_NAME`, `USER`, and `HOST` columns. Each row summarizes events for a given account (user and host combination) and event name.
- `events_statements_summary_by_digest` has `SCHEMA_NAME` and `DIGEST` columns. Each row summarizes events per schema and digest value. (The `DIGEST_TEXT` column contains the corresponding normalized statement digest text, but is neither a grouping nor a summary column.)

The maximum number of rows in the table is autosized at server startup. To set this maximum explicitly, set the `performance_schema_digests_size` system variable at server startup.

- `events_statements_summary_by_host_by_event_name` has `EVENT_NAME` and `HOST` columns. Each row summarizes events for a given host and event name.
- `events_statements_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.
- `events_statements_summary_by_user_by_event_name` has `EVENT_NAME` and `USER` columns. Each row summarizes events for a given user and event name.
- `events_statements_summary_global_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.

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 wait event summary tables (see [Section 10.9.1, “Wait Event Summary Tables”](#)), except that the statement summary tables aggregate events from `events_statements_current` rather than `events_waits_current`.

- `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`, `LAST_SEEN`

Timestamps indicating when statements with the given digest value were first seen and most recently seen.

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

- For `events_statements_summary_by_digest`, it removes the rows.
- For other summary tables not aggregated by account, host, or user, truncation resets the summary columns to zero rather than removing rows.
- For other summary tables aggregated by account, host, or user, truncation removes rows for accounts, hosts, or users with no connections, and resets the summary columns to zero for the remaining rows.

In addition, each statement summary table that is aggregated by account, host, user, or thread is implicitly truncated by truncation of the connection table on which it depends, or truncation of `events_statements_summary_global_by_event_name`. For details, see [Section 10.7, “Performance Schema Connection Tables”](#).

Statement Digest Aggregation Rules

If the `statements_digest` consumer is enabled, aggregation into `events_statements_summary_by_digest` occurs as follows when a statement completes. Aggregation is based on the `DIGEST` value computed for the statement.

- If a `events_statements_summary_by_digest` row already exists with the digest value for the statement that just completed, statistics for the statement are aggregated to that row. The `LAST_SEEN` column is updated to the current time.
- If no row has the digest value for the statement that just completed, and the table is not full, a new row is created for the statement. The `FIRST_SEEN` and `LAST_SEEN` columns are initialized with the current time.
- If no row has the statement digest value for the statement that just completed, and the table is full, the statistics for the statement that just completed are added to a special “catch-all” row with `DIGEST = NULL`, which is created if necessary. If the row is created, the `FIRST_SEEN` and `LAST_SEEN` columns are initialized with the current time. Otherwise, the `LAST_SEEN` column is updated with the current time.

The row with `DIGEST = NULL` is maintained because Performance Schema tables have a maximum size due to memory constraints. The `DIGEST = NULL` row permits digests that do not match other rows to be counted even if the summary table is full, using a common “other” bucket. This row helps you estimate whether the digest summary is representative:

- A `DIGEST = NULL` row that has a `COUNT_STAR` value that represents 5% of all digests shows that the digest summary table is very representative; the other rows cover 95% of the statements seen.
- A `DIGEST = NULL` row that has a `COUNT_STAR` value that represents 50% of all digests shows that the digest summary table is not very representative; the other rows cover only half the statements seen. Most likely the DBA should increase the maximum table size so that more of the rows counted in the `DIGEST = NULL` row would be counted using more specific rows instead. To do this, set the `performance_schema_digests_size` system variable to a larger value at server startup. The default size is 200.

10.9.4 Object Wait Summary Table

The Performance Schema maintains the `objects_summary_global_by_type` table for aggregating object wait events.

Example object wait event summary information:

```
mysql> SELECT * FROM performance_schema.objects_summary_global_by_type\G
...
***** 3. row *****
  OBJECT_TYPE: TABLE
  OBJECT_SCHEMA: test
  OBJECT_NAME: t
  COUNT_STAR: 3
  SUM_TIMER_WAIT: 263126976
  MIN_TIMER_WAIT: 1522272
  AVG_TIMER_WAIT: 87708678
  MAX_TIMER_WAIT: 258428280
  ...
***** 10. row *****
  OBJECT_TYPE: TABLE
  OBJECT_SCHEMA: mysql
```

```

OBJECT_NAME: user
COUNT_STAR: 14
SUM_TIMER_WAIT: 365567592
MIN_TIMER_WAIT: 1141704
AVG_TIMER_WAIT: 26111769
MAX_TIMER_WAIT: 334783032
...

```

The `objects_summary_global_by_type` table has these grouping columns to indicate how the table aggregates events: `OBJECT_TYPE`, `OBJECT_SCHEMA`, and `OBJECT_NAME`. Each row summarizes events for the given object.

`objects_summary_global_by_type` has the same summary columns as the `events_waits_summary_by_xxx` tables. See [Section 10.9.1, “Wait Event Summary Tables”](#).

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

10.9.5 File I/O Summary Tables

The Performance Schema maintains file I/O summary tables that aggregate information about I/O operations.

Example file I/O event summary information:

```

mysql> SELECT * FROM performance_schema.file_summary_by_event_name\G
...
***** 2. row *****
          EVENT_NAME: wait/io/file/sql/binlog
          COUNT_STAR: 31
          SUM_TIMER_WAIT: 8243784888
          MIN_TIMER_WAIT: 0
          AVG_TIMER_WAIT: 265928484
          MAX_TIMER_WAIT: 6490658832
...
mysql> SELECT * FROM performance_schema.file_summary_by_instance\G
...
***** 2. row *****
          FILE_NAME: /var/mysql/share/english/errmsg.sys
          EVENT_NAME: wait/io/file/sql/ERRMSG
          EVENT_NAME: wait/io/file/sql/ERRMSG
          OBJECT_INSTANCE_BEGIN: 4686193384
          COUNT_STAR: 5
          SUM_TIMER_WAIT: 13990154448
          MIN_TIMER_WAIT: 26349624
          AVG_TIMER_WAIT: 2798030607
          MAX_TIMER_WAIT: 8150662536
...

```

Each file I/O summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `file_summary_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.
- `file_summary_by_instance` has `FILE_NAME`, `EVENT_NAME`, and `OBJECT_INSTANCE_BEGIN` columns. Each row summarizes events for a given file and event name.

Each file I/O summary table has the following summary columns containing aggregated values. Some columns are more general and have values that are the same as the sum of the values of more fine-grained columns. In this way, aggregations at higher levels are available directly without the need for user-defined views that sum lower-level columns.

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

These columns aggregate all I/O operations.

- `COUNT_READ`, `SUM_TIMER_READ`, `MIN_TIMER_READ`, `AVG_TIMER_READ`, `MAX_TIMER_READ`, `SUM_NUMBER_OF_BYTES_READ`

These columns aggregate all read operations, including `FGETS`, `FGETC`, `FREAD`, and `READ`.

- `COUNT_WRITE`, `SUM_TIMER_WRITE`, `MIN_TIMER_WRITE`, `AVG_TIMER_WRITE`, `MAX_TIMER_WRITE`, `SUM_NUMBER_OF_BYTES_WRITE`

These columns aggregate all write operations, including `FPUTS`, `FPUTC`, `FPRINTF`, `VFPRINTF`, `FWRITE`, and `PWRITE`.

- `COUNT_MISC`, `SUM_TIMER_MISC`, `MIN_TIMER_MISC`, `AVG_TIMER_MISC`, `MAX_TIMER_MISC`

These columns aggregate all other I/O operations, including `CREATE`, `DELETE`, `OPEN`, `CLOSE`, `STREAM_OPEN`, `STREAM_CLOSE`, `SEEK`, `TELL`, `FLUSH`, `STAT`, `FSTAT`, `CHSIZE`, `RENAME`, and `SYNC`. There are no byte counts for these operations.

`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.

10.9.6 Table I/O and Lock Wait Summary Tables

The following sections describe the table I/O and lock wait summary tables:

- `table_io_waits_summary_by_index_usage`: Table I/O waits per index
- `table_io_waits_summary_by_table`: Table I/O waits per table
- `table_lock_waits_summary_by_table`: Table lock waits per table

10.9.6.1 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.

10.9.6.2 The `table_io_waits_summary_by_index_usage` Table

The `table_io_waits_summary_by_index_usage` table aggregates all table index I/O wait events, as generated by the `wait/io/table/sql/handler` instrument. The grouping is by table index.

The columns of `table_io_waits_summary_by_index_usage` are nearly identical to `table_io_waits_summary_by_table`. The only difference is the additional group column, `INDEX_NAME`, which corresponds to the name of the index that was used when the table I/O wait event was recorded:

- A value of `PRIMARY` indicates that table I/O used the primary index.
- A value of `NULL` means that table I/O used no index.
- Inserts are counted against `INDEX_NAME = NULL`.

`TRUNCATE TABLE` is permitted for table I/O summary tables. It resets the summary columns to zero rather than removing rows. This table is also truncated by truncation of the `table_io_waits_summary_by_table` table. A DDL operation that changes the index structure of a table may cause the per-index statistics to be reset.

10.9.6.3 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_DELAYED`, `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_DELAYED`, `SUM_TIMER_WRITE_DELAYED`, `MIN_TIMER_WRITE_DELAYED`, `AVG_TIMER_WRITE_DELAYED`, `MAX_TIMER_WRITE_DELAYED`

These columns aggregate internal write locks.

`DELAYED` inserts are deprecated, so these columns will be removed in a future release.

- `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.

10.9.7 Socket Summary Tables

The Performance Schema maintains socket summary tables, which aggregate timer and byte count information for socket operations:

- `socket_summary_by_event_name`: Aggregate timer and byte count statistics generated by the `wait/io/socket/*` instruments for all socket I/O operations, per socket instrument.
- `socket_summary_by_instance`: Aggregate timer and byte count statistics generated by the `wait/io/socket/*` instruments for all socket I/O operations, per socket instance. When a connection terminates, the row in `socket_summary_by_instance` corresponding to it is deleted.

The socket summary tables do not aggregate waits generated by `idle` events while sockets are waiting for the next request from the client. For `idle` event aggregations, use the wait-event summary tables; see [Section 10.9.1, “Wait Event Summary Tables”](#).

Each socket summary table has one or more grouping columns to indicate how the table aggregates events. Event names refer to names of event instruments in the `setup_instruments` table:

- `socket_summary_by_event_name` has an `EVENT_NAME` column. Each row summarizes events for a given event name.
- `socket_summary_by_instance` has an `OBJECT_INSTANCE_BEGIN` column. Each row summarizes events for a given object.

Each socket summary table has these summary columns containing aggregated values:

- `COUNT_STAR`, `SUM_TIMER_WAIT`, `MIN_TIMER_WAIT`, `AVG_TIMER_WAIT`, `MAX_TIMER_WAIT`

These columns aggregate all operations.

- `COUNT_READ`, `SUM_TIMER_READ`, `MIN_TIMER_READ`, `AVG_TIMER_READ`, `MAX_TIMER_READ`, `SUM_NUMBER_OF_BYTES_READ`

These columns aggregate all receive operations (`RECV`, `RECVFROM`, and `RECVMSG`).

- `COUNT_WRITE`, `SUM_TIMER_WRITE`, `MIN_TIMER_WRITE`, `AVG_TIMER_WRITE`, `MAX_TIMER_WRITE`, `SUM_NUMBER_OF_BYTES_WRITE`

These columns aggregate all send operations (`SEND`, `SENDTO`, and `SENDMSG`).

- `COUNT_MISC`, `SUM_TIMER_MISC`, `MIN_TIMER_MISC`, `AVG_TIMER_MISC`, `MAX_TIMER_MISC`

These columns aggregate all other socket operations, such as `CONNECT`, `LISTEN`, `ACCEPT`, `CLOSE`, and `SHUTDOWN`. There are no byte counts for these operations.

The `socket_summary_by_instance` table also has an `EVENT_NAME` column that indicates the class of the socket: `client_connection`, `server_tcpip_socket`, `server_unix_socket`. This column can be grouped on to isolate, for example, client activity from that of the server listening sockets.

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

10.10 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

10.10.1 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 Domain Name System (DNS) lookups. The `host_cache_size` system variable controls the size of the host cache, as well as the size of the `host_cache` table that exposes the cache contents. For operational and configuration information about the host cache, see [DNS Lookup Optimization and the Host Cache](#).

Because the `host_cache` table exposes the contents of the host cache, it can be examined using `SELECT` statements. This may help you diagnose the causes of connection problems. The Performance Schema must be enabled or this table is empty.

The `host_cache` table has these columns:

- `IP`

The IP address of the client that connected to the server, expressed as a string.

- `HOST`

The resolved DNS host name for that client IP, or `NULL` if the name is unknown.

- `HOST_VALIDATED`

Whether the IP-to-host name-to-IP DNS resolution was performed successfully for the client IP. If `HOST_VALIDATED` is `YES`, the `HOST` column is used as the host name corresponding to the IP so that additional calls to DNS can be avoided. While `HOST_VALIDATED` is `NO`, DNS resolution is attempted for each connection attempt, until it eventually completes with either a valid result or a permanent error. This information enables the server to avoid caching bad or missing host names during temporary DNS failures, which would negatively affect clients forever.

- `SUM_CONNECT_ERRORS`

The number of connection errors that are deemed “blocking” (assessed against the `max_connect_errors` system variable). Only protocol handshake errors are counted, and only for hosts that passed validation (`HOST_VALIDATED = YES`).

Once `SUM_CONNECT_ERRORS` for a given host reaches the value of `max_connect_errors`, new connections from that host are blocked. The `SUM_CONNECT_ERRORS` value can exceed the `max_connect_errors` value because multiple connection attempts from a host can occur simultaneously while the host is not blocked. Any or all of them can fail, independently incrementing `SUM_CONNECT_ERRORS`, possibly beyond the value of `max_connect_errors`.

Suppose that `max_connect_errors` is 200 and `SUM_CONNECT_ERRORS` for a given host is 199. If 10 clients attempt to connect from that host simultaneously, none of them are blocked because `SUM_CONNECT_ERRORS` has not reached 200. If blocking errors occur for five of the clients, `SUM_CONNECT_ERRORS` is increased by one for each client, for a resulting `SUM_CONNECT_ERRORS` value of 204. The other five clients succeed and are not blocked because the value of `SUM_CONNECT_ERRORS` when their connection attempts began had not reached 200. New connections from the host that begin after `SUM_CONNECT_ERRORS` reaches 200 are blocked.

- `COUNT_HOST_BLOCKED_ERRORS`

The number of connections that were blocked because `SUM_CONNECT_ERRORS` exceeded the value of the `max_connect_errors` system variable.

- `COUNT_NAMEINFO_TRANSIENT_ERRORS`

The number of transient errors during IP-to-host name DNS resolution.

- `COUNT_NAMEINFO_PERMANENT_ERRORS`

The number of permanent errors during IP-to-host name DNS resolution.

- `COUNT_FORMAT_ERRORS`

The number of host name format errors. MySQL does not perform matching of `Host` column values in the `mysql.user` system table against host names for which one or more of the initial components of the name are entirely numeric, such as `1.2.example.com`. The client IP address is used instead. For the rationale why this type of matching does not occur, see [Specifying Account Names](#).

- [COUNT_ADDRINFO_TRANSIENT_ERRORS](#)
The number of transient errors during host name-to-IP reverse DNS resolution.
- [COUNT_ADDRINFO_PERMANENT_ERRORS](#)
The number of permanent errors during host name-to-IP reverse DNS resolution.
- [COUNT_FCRDNS_ERRORS](#)
The number of forward-confirmed reverse DNS errors. These errors occur when IP-to-host name-to-IP DNS resolution produces an IP address that does not match the client originating IP address.
- [COUNT_HOST_ACL_ERRORS](#)
The number of errors that occur because no users are permitted to connect from the client host. In such cases, the server returns [ER_HOST_NOT_PRIVILEGED](#) and does not even ask for a user name or password.
- [COUNT_NO_AUTH_PLUGIN_ERRORS](#)
The number of errors due to requests for an unavailable authentication plugin. A plugin can be unavailable if, for example, it was never loaded or a load attempt failed.
- [COUNT_AUTH_PLUGIN_ERRORS](#)
The number of errors reported by authentication plugins.

An authentication plugin can report different error codes to indicate the root cause of a failure. Depending on the type of error, one of these columns is incremented: [COUNT_AUTHENTICATION_ERRORS](#), [COUNT_AUTH_PLUGIN_ERRORS](#), [COUNT_HANDSHAKE_ERRORS](#). New return codes are an optional extension to the existing plugin API. Unknown or unexpected plugin errors are counted in the [COUNT_AUTH_PLUGIN_ERRORS](#) column.
- [COUNT_HANDSHAKE_ERRORS](#)
The number of errors detected at the wire protocol level.
- [COUNT_PROXY_USER_ERRORS](#)
The number of errors detected when proxy user A is proxied to another user B who does not exist.
- [COUNT_PROXY_USER_ACL_ERRORS](#)
The number of errors detected when proxy user A is proxied to another user B who does exist but for whom A does not have the [PROXY](#) privilege.
- [COUNT_AUTHENTICATION_ERRORS](#)
The number of errors caused by failed authentication.
- [COUNT_SSL_ERRORS](#)
The number of errors due to SSL problems.
- [COUNT_MAX_USER_CONNECTIONS_ERRORS](#)
The number of errors caused by exceeding per-user connection quotas. See [Setting Account Resource Limits](#).
- [COUNT_MAX_USER_CONNECTIONS_PER_HOUR_ERRORS](#)
The number of errors caused by exceeding per-user connections-per-hour quotas. See [Setting Account Resource Limits](#).

- [COUNT_DEFAULT_DATABASE_ERRORS](#)

The number of errors related to the default database. For example, the database does not exist or the user has no privileges to access it.

- [COUNT_INIT_CONNECT_ERRORS](#)

The number of errors caused by execution failures of statements in the `init_connect` system variable value.

- [COUNT_LOCAL_ERRORS](#)

The number of errors local to the server implementation and not related to the network, authentication, or authorization. For example, out-of-memory conditions fall into this category.

- [COUNT_UNKNOWN_ERRORS](#)

The number of other, unknown errors not accounted for by other columns in this table. This column is reserved for future use, in case new error conditions must be reported, and if preserving the backward compatibility and structure of the `host_cache` table is required.

- [FIRST_SEEN](#)

The timestamp of the first connection attempt seen from the client in the `IP` column.

- [LAST_SEEN](#)

The timestamp of the most recent connection attempt seen from the client in the `IP` column.

- [FIRST_ERROR_SEEN](#)

The timestamp of the first error seen from the client in the `IP` column.

- [LAST_ERROR_SEEN](#)

The timestamp of the most recent error seen from the client in the `IP` column.

The `FLUSH HOSTS` statement, `TRUNCATE TABLE host_cache` statement, and `mysqladmin flush-hosts` command have the same effect: They clear the host cache, remove all rows from the `host_cache` table that exposes the cache contents, and unblock any blocked hosts (see [Host 'host_name' is blocked](#)). `FLUSH HOSTS` and `mysqladmin flush-hosts` require the `RELOAD` privilege. `TRUNCATE TABLE` requires the `DROP` privilege for the `host_cache` table.

10.10.2 The performance_timers Table

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

```
mysql> SELECT * FROM performance_schema.performance_timers;
```

TIMER_NAME	TIMER_FREQUENCY	TIMER_RESOLUTION	TIMER_OVERHEAD
CYCLE	2389029850	1	72
NANOSECOND	1000000000	1	112
MICROSECOND	1000000	1	136
MILLISECOND	1036	1	168
TICK	105	1	2416

If the values associated with a given timer name are `NULL`, that timer is not supported on your platform. The rows that do not contain `NULL` indicate which timers you can use in `setup_timers`. For an explanation of how event timing occurs, see [Section 5.1, “Performance Schema Event Timing”](#).

The `performance_timers` table has these columns:

- [TIMER_NAME](#)

The 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.

[TRUNCATE TABLE](#) is not permitted for the [performance_timers](#) table.

10.10.3 The threads Table

The [threads](#) table contains a row for each server thread. Each row contains information about a thread and indicates whether monitoring is enabled for it:

```
mysql> SELECT * FROM performance_schema.threads\G
***** 1. row *****
      THREAD_ID: 1
        NAME: thread/sql/main
        TYPE: BACKGROUND
  PROCESSLIST_ID: NULL
  PROCESSLIST_USER: NULL
  PROCESSLIST_HOST: NULL
  PROCESSLIST_DB: NULL
  PROCESSLIST_COMMAND: NULL
  PROCESSLIST_TIME: 80284
  PROCESSLIST_STATE: NULL
  PROCESSLIST_INFO: NULL
  PARENT_THREAD_ID: NULL
          ROLE: NULL
  INSTRUMENTED: YES
...
***** 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 performance_schema.threads
  PARENT_THREAD_ID: 1
          ROLE: NULL
  INSTRUMENTED: YES
...
```

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` column value for new threads is determined by the contents of the `setup_actors` table. For information about how to use the `setup_actors` table to control this column, see [Section 5.6, “Pre-Filtering by Thread”](#).

Removal of rows from the `threads` table occurs when threads end. For a thread associated with a client session, removal occurs when the session ends. If a client has auto-reconnect enabled and the session reconnects after a disconnect, the session becomes associated with a new row in the `threads` table that has a different `PROCESSLIST_ID` value. The initial `INSTRUMENTED` value for the new thread may be different from that of the original thread: The `setup_actors` table may have changed in the meantime, and if the `INSTRUMENTED` value for the original thread was changed after it was initialized, that 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). To control the initial `INSTRUMENTED` value for new foreground threads, use the `setup_actors` table. To control monitoring of existing threads, set the `INSTRUMENTED` column of `threads` table rows. (For more information about the conditions under which thread monitoring occurs, see the description of the `INSTRUMENTED` column.)

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 performance_schema.setup_instruments
      WHERE NAME LIKE 'wait/io/socket%';
+-----+-----+-----+
| NAME                                     | ENABLED | TIMED |
+-----+-----+-----+
| wait/io/socket/sql/server_tcpip_socket | NO      | NO    |
| wait/io/socket/sql/server_unix_socket  | NO      | NO    |
| wait/io/socket/sql/client_connection    | NO      | NO    |
+-----+-----+-----+
3 rows in set (0.01 sec)
mysql> UPDATE performance_schema.setup_instruments
      SET ENABLED='YES'
      WHERE NAME LIKE 'wait/io/socket%';
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3  Changed: 3  Warnings: 0
mysql> SELECT * FROM performance_schema.socket_instances\G
***** 1. row *****
EVENT_NAME: wait/io/socket/sql/client_connection
OBJECT_INSTANCE_BEGIN: 140612577298432
      THREAD_ID: 31
      SOCKET_ID: 53
          IP: ::ffff:127.0.0.1
          PORT: 55642
          STATE: ACTIVE
...

```

- [PROCESSLIST_DB](#)

The default database for the thread, or [NULL](#) if 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 is [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. Thread spawning occurs, for example, to handle insertion of rows from [INSERT DELAYED](#) statements.

- [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. This is the only [threads](#) table column that can be modified.

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.

[TRUNCATE TABLE](#) is not permitted for the [threads](#) table.

Chapter 11 Performance Schema and Plugins

Removing a plugin with `UNINSTALL PLUGIN` does not affect information already collected for code in that plugin. Time spent executing the code while the plugin was loaded was still spent even if the plugin is unloaded later. The associated event information, including aggregate information, remains readable in `performance_schema` database tables. For additional information about the effect of plugin installation and removal, see [Chapter 8, Performance Schema Status Monitoring](#).

A plugin implementor who instruments plugin code should document its instrumentation characteristics to enable those who load the plugin to account for its requirements. For example, a third-party storage engine should include in its documentation how much memory the engine needs for mutex and other instruments.

Chapter 12 Performance Schema System Variables

The Performance Schema implements several system variables that provide configuration information:

```
mysql> SHOW VARIABLES LIKE 'perf%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| performance_schema | ON |
| performance_schema_accounts_size | 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 |
| performance_schema_max_file_classes | 50 |
| performance_schema_max_file_handles | 32768 |
| performance_schema_max_file_instances | 10000 |
| performance_schema_max_mutex_classes | 200 |
| performance_schema_max_mutex_instances | 1000000 |
| performance_schema_max_rwlock_classes | 30 |
| performance_schema_max_rwlock_instances | 1000000 |
| performance_schema_max_socket_classes | 10 |
| performance_schema_max_socket_instances | 1000 |
| performance_schema_max_stage_classes | 150 |
| performance_schema_max_statement_classes | 165 |
| performance_schema_max_table_handles | 10000 |
| performance_schema_max_table_instances | 1000 |
| performance_schema_max_thread_classes | 50 |
| performance_schema_max_thread_instances | 1000 |
| performance_schema_session_connect_attrs_size | 512 |
| performance_schema_setup_actors_size | 100 |
| performance_schema_setup_objects_size | 100 |
| performance_schema_users_size | 100 |
+-----+-----+
```

Performance Schema system variables can be set at server startup on the command line or in option files, and many can be set at runtime. See [Performance Schema Option and Variable Reference](#).

The Performance Schema automatically sizes the values of several of its parameters at server startup if they are not set explicitly. For more information, see [Chapter 4, Performance Schema Startup Configuration](#).

Performance Schema system variables have the following meanings:

- `performance_schema`

Property	Value
Command-Line Format	<code>--performance-schema=#</code>
System Variable	<code>performance_schema</code>
Scope	Global
Dynamic	No
Type	Boolean
Default Value (>= 5.6.6)	ON
Default Value (<= 5.6.5)	OFF

The value of this variable is `ON` or `OFF` to indicate whether the Performance Schema is enabled. By default, the value is `ON`. At server startup, you can specify this variable with no value or a value of `ON` or 1 to enable it, or with a value of `OFF` or 0 to disable it.

- `performance_schema_accounts_size`

Property	Value
Command-Line Format	<code>--performance-schema-accounts-size=#</code>
Introduced	5.6.3
System Variable	<code>performance_schema_accounts_size</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	10
Minimum Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Minimum Value (<= 5.6.5)	0
Maximum Value	1048576

The number of rows in the `accounts` table. If this variable is 0, the Performance Schema does not maintain connection statistics in the `accounts` table.

- `performance_schema_digests_size`

Property	Value
Command-Line Format	<code>--performance-schema-digests-size=#</code>
Introduced	5.6.5
System Variable	<code>performance_schema_digests_size</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1
Maximum Value	1048576

The maximum number of rows in the `events_statements_summary_by_digest` table. If this maximum is exceeded such that a digest cannot be instrumented, the Performance Schema increments the `Performance_schema_digest_lost` status variable.

For more information about statement digesting, see [Performance Schema Statement Digests](#).

- `performance_schema_events_stages_history_long_size`

Property	Value
Command-Line Format	<code>--performance-schema-events-stages-history-long-size=#</code>

Property	Value
Introduced	5.6.3
System Variable	<code>performance_schema_events_stages_history_lo</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10000

The number of rows in the `events_stages_history_long` table.

- `performance_schema_events_stages_history_size`

Property	Value
Command-Line Format	<code>--performance-schema-events-stages-history-size=#</code>
Introduced	5.6.3
System Variable	<code>performance_schema_events_stages_history_si</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10

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

- `performance_schema_events_statements_history_long_size`

Property	Value
Command-Line Format	<code>--performance-schema-events-statements-history-long-size=#</code>
Introduced	5.6.3
System Variable	<code>performance_schema_events_statements_histor</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10000

The number of rows in the `events_statements_history_long` table.

- `performance_schema_events_statements_history_size`

Property	Value
Command-Line Format	<code>--performance-schema-events-statements-history-size=#</code>

Property	Value
Introduced	5.6.3
System Variable	<code>performance_schema_events_statements_history_s</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10

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

- `performance_schema_events_waits_history_long_size`

Property	Value
Command-Line Format	<code>--performance-schema-events-waits-history-long-size=#</code>
System Variable	<code>performance_schema_events_waits_history_long_s</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10000

The number of rows in the `events_waits_history_long` table.

- `performance_schema_events_waits_history_size`

Property	Value
Command-Line Format	<code>--performance-schema-events-waits-history-size=#</code>
System Variable	<code>performance_schema_events_waits_history_size</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10

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

- `performance_schema_hosts_size`

Property	Value
Command-Line Format	<code>--performance-schema-hosts-size=#</code>
Introduced	5.6.3
System Variable	<code>performance_schema_hosts_size</code>
Scope	Global

Property	Value
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	10
Minimum Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Minimum Value (<= 5.6.5)	0
Maximum Value	1048576

The number of rows in the `hosts` table. If this variable is 0, the Performance Schema does not maintain connection statistics in the `hosts` table.

- `performance_schema_max_cond_classes`

Property	Value
Command-Line Format	<code>--performance-schema-max-cond-classes=#</code>
System Variable	<code>performance_schema_max_cond_classes</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	80
Minimum Value	0
Maximum Value	256

The maximum number of condition instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_cond_instances`

Property	Value
Command-Line Format	<code>--performance-schema-max-cond-instances=#</code>
System Variable	<code>performance_schema_max_cond_instances</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	1000

The maximum number of instrumented condition objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_digest_length`

Property	Value
Command-Line Format	<code>--performance-schema-max-digest-length=#</code>
Introduced	5.6.26
System Variable	<code>performance_schema_max_digest_length</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	1024
Minimum Value	0
Maximum Value	1048576

The maximum number of bytes of memory reserved per statement for computation of normalized statement digest values in the Performance Schema. This variable is related to `max_digest_length`; see the description of that variable in [Server System Variables](#).

For more information about statement digesting, including considerations regarding memory use, see [Performance Schema Statement Digests](#).

- `performance_schema_max_file_classes`

Property	Value
Command-Line Format	<code>--performance-schema-max-file-classes=#</code>
System Variable	<code>performance_schema_max_file_classes</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	50
Minimum Value	0
Maximum Value	256

The maximum number of file instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_file_handles`

Property	Value
Command-Line Format	<code>--performance-schema-max-file-handles=#</code>
System Variable	<code>performance_schema_max_file_handles</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	32768

The maximum number of opened file objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

The value of `performance_schema_max_file_handles` should be greater than the value of `open_files_limit`: `open_files_limit` affects the maximum number of open file handles the server can support and `performance_schema_max_file_handles` affects how many of these file handles can be instrumented.

- `performance_schema_max_file_instances`

Property	Value
Command-Line Format	<code>--performance-schema-max-file-instances=#</code>
System Variable	<code>performance_schema_max_file_instances</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	10000

The maximum number of instrumented file objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_mutex_classes`

Property	Value
Command-Line Format	<code>--performance-schema-max-mutex-classes=#</code>
System Variable	<code>performance_schema_max_mutex_classes</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	200
Minimum Value	0
Maximum Value	256

The maximum number of mutex instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_mutex_instances`

Property	Value
Command-Line Format	<code>--performance-schema-max-mutex-instances=#</code>
System Variable	<code>performance_schema_max_mutex_instances</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	1000

The maximum number of instrumented mutex objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_rwlock_classes`

Property	Value
Command-Line Format	<code>--performance-schema-max-rwlock-classes=#</code>
System Variable	<code>performance_schema_max_rwlock_classes</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value ($\geq 5.6.15$)	40
Default Value ($\geq 5.6.1$, $\leq 5.6.14$)	30
Default Value (5.6.0)	20
Minimum Value	0
Maximum Value	256

The maximum number of rwlock instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_rwlock_instances`

Property	Value
Command-Line Format	<code>--performance-schema-max-rwlock-instances=#</code>
System Variable	<code>performance_schema_max_rwlock_instances</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value ($\geq 5.6.6$)	-1 (signifies autosizing; do not assign this literal value)
Default Value ($\leq 5.6.5$)	1000

The maximum number of instrumented rwlock objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_socket_classes`

Property	Value
Command-Line Format	<code>--performance-schema-max-socket-classes=#</code>
Introduced	5.6.3
System Variable	<code>performance_schema_max_socket_classes</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	10

Property	Value
Minimum Value	0
Maximum Value	256

The maximum number of socket instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- [performance_schema_max_socket_instances](#)

Property	Value
Command-Line Format	<code>--performance-schema-max-socket-instances=#</code>
Introduced	5.6.3
System Variable	performance_schema_max_socket_instances
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	1000

The maximum number of instrumented socket objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- [performance_schema_max_stage_classes](#)

Property	Value
Command-Line Format	<code>--performance-schema-max-stage-classes=#</code>
Introduced	5.6.3
System Variable	performance_schema_max_stage_classes
Scope	Global
Dynamic	No
Type	Integer
Default Value	150
Minimum Value	0
Maximum Value	256

The maximum number of stage instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- [performance_schema_max_statement_classes](#)

Property	Value
Command-Line Format	<code>--performance-schema-max-statement-classes=#</code>
Introduced	5.6.3
System Variable	performance_schema_max_statement_classes
Scope	Global

Property	Value
Dynamic	No
Type	Integer
Default Value	-1 (signifies autosizing; do not assign this literal value)

The maximum number of statement instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

The default value is calculated at server build time based on the number of commands in the client/server protocol and the number of SQL statement types supported by the server.

This variable should not be changed, unless to set it to 0 to disable all statement instrumentation and save all memory associated with it. Setting the variable to nonzero values other than the default has no benefit; in particular, values larger than the default cause more memory to be allocated than is needed.

- `performance_schema_max_table_handles`

Property	Value
Command-Line Format	<code>--performance-schema-max-table-handles=#</code>
System Variable	<code>performance_schema_max_table_handles</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	100000

The maximum number of opened table objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_table_instances`

Property	Value
Command-Line Format	<code>--performance-schema-max-table-instances=#</code>
System Variable	<code>performance_schema_max_table_instances</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	50000

The maximum number of instrumented table objects. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_thread_classes`

Property	Value
Command-Line Format	<code>--performance-schema-max-thread-classes=#</code>
System Variable	<code>performance_schema_max_thread_classes</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	50
Minimum Value	0
Maximum Value	256

The maximum number of thread instruments. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

- `performance_schema_max_thread_instances`

Property	Value
Command-Line Format	<code>--performance-schema-max-thread-instances=#</code>
System Variable	<code>performance_schema_max_thread_instances</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (>= 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (<= 5.6.5)	1000

The maximum number of instrumented thread objects. The value controls the size of the `threads` table. If this maximum is exceeded such that a thread cannot be instrumented, the Performance Schema increments the `performance_schema_thread_instances_lost` status variable. For information about how to set and use this variable, see [Chapter 8, Performance Schema Status Monitoring](#).

The `max_connections` system variable affects how many threads are run in the server. `performance_schema_max_thread_instances` affects how many of these running threads can be instrumented. The default value of `performance_schema_max_thread_instances` is autosized based on the value of `max_connections`.

- `performance_schema_session_connect_attrs_size`

Property	Value
Command-Line Format	<code>--performance-schema-session-connect-attrs-size=#</code>
Introduced	5.6.6
System Variable	<code>performance_schema_session_connect_attrs_size</code>
Scope	Global
Dynamic	No
Type	Integer

Property	Value
Default Value	-1 (signifies autosizing; do not assign this literal value)
Minimum Value	-1
Maximum 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 10.8, “Performance Schema Connection Attribute Tables”](#).

- `performance_schema_setup_actors_size`

Property	Value
Command-Line Format	<code>--performance-schema-setup-actors-size=#</code>
Introduced	5.6.1
System Variable	<code>performance_schema_setup_actors_size</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	100

The number of rows in the `setup_actors` table.

- `performance_schema_setup_objects_size`

Property	Value
Command-Line Format	<code>--performance-schema-setup-objects-size=#</code>
Introduced	5.6.1
System Variable	<code>performance_schema_setup_objects_size</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value	100

The number of rows in the `setup_objects` table.

-
- `performance_schema_users_size`

Property	Value
Command-Line Format	<code>--performance-schema-users-size=#</code>
Introduced	5.6.3
System Variable	<code>performance_schema_users_size</code>
Scope	Global
Dynamic	No
Type	Integer
Default Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Default Value (\leq 5.6.5)	10
Minimum Value (\geq 5.6.6)	-1 (signifies autosizing; do not assign this literal value)
Minimum Value (\leq 5.6.5)	0
Maximum Value	1048576

The number of rows in the `users` table. If this variable is 0, the Performance Schema does not maintain connection statistics in the `users` table.

Chapter 13 Performance Schema Status Variables

The Performance Schema implements several status variables that provide information about instrumentation that could not be loaded or created due to memory constraints:

```
mysql> SHOW STATUS LIKE 'perf%';
```

Variable_name	Value
Performance_schema_accounts_lost	0
Performance_schema_cond_classes_lost	0
Performance_schema_cond_instances_lost	0
Performance_schema_file_classes_lost	0
Performance_schema_file_handles_lost	0
Performance_schema_file_instances_lost	0
Performance_schema_hosts_lost	0
Performance_schema_locker_lost	0
Performance_schema_mutex_classes_lost	0
Performance_schema_mutex_instances_lost	0
Performance_schema_rwlock_classes_lost	0
Performance_schema_rwlock_instances_lost	0
Performance_schema_socket_classes_lost	0
Performance_schema_socket_instances_lost	0
Performance_schema_stage_classes_lost	0
Performance_schema_statement_classes_lost	0
Performance_schema_table_handles_lost	0
Performance_schema_table_instances_lost	0
Performance_schema_thread_classes_lost	0
Performance_schema_thread_instances_lost	0
Performance_schema_users_lost	0

For information on using these variables to check Performance Schema status, see [Chapter 8, Performance Schema Status Monitoring](#).

Performance Schema status variables have the following meanings:

- `Performance_schema_accounts_lost`
The number of times a row could not be added to the `accounts` table because it was full.
- `Performance_schema_cond_classes_lost`
How many condition instruments could not be loaded.
- `Performance_schema_cond_instances_lost`
How many condition instrument instances could not be created.
- `Performance_schema_digest_lost`
The number of digest instances that could not be instrumented in the `events_statements_summary_by_digest` table. This can be nonzero if the value of `performance_schema_digests_size` is too small.
- `Performance_schema_file_classes_lost`
How many file instruments could not be loaded.
- `Performance_schema_file_handles_lost`
How many file instrument instances could not be opened.
- `Performance_schema_file_instances_lost`
How many file instrument instances could not be created.

-
- `Performance_schema_hosts_lost`

The number of times a row could not be added to the `hosts` table because it was full.

- `Performance_schema_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_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_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 than the reserved storage permitted by the value of the `performance_schema_session_connect_attrs_size` system variable, the Performance Schema truncates the attribute data and increments `Performance_schema_session_connect_attrs_lost`. If this value is nonzero, you may wish to set `performance_schema_session_connect_attrs_size` to a larger value.

For more information about connection attributes, see [Section 10.8, “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.

- `Performance_schema_table_instances_lost`

How many table instrument instances could not be created.

- `Performance_schema_thread_classes_lost`

How many thread instruments could not be loaded.

- `Performance_schema_thread_instances_lost`

The number of thread instances that could not be instrumented in the `threads` table. This can be nonzero if the value of `performance_schema_max_thread_instances` is too small.

- `Performance_schema_users_lost`

The number of times a row could not be added to the `users` table because it was full.

Chapter 14 Using the Performance Schema to Diagnose Problems

Table of Contents

14.1 Query Profiling Using Performance Schema	122
---	-----

The Performance Schema is a tool to help a DBA do performance tuning by taking real measurements instead of “wild guesses.” This section demonstrates some ways to use the Performance Schema for this purpose. The discussion here relies on the use of event filtering, which is described in [Section 5.2, “Performance Schema Event Filtering”](#).

The following example provides one methodology that you can use to analyze a repeatable problem, such as investigating a performance bottleneck. To begin, you should have a repeatable use case where performance is deemed “too slow” and needs optimization, and you should enable all instrumentation (no pre-filtering at all).

1. Run the use case.
2. Using the Performance Schema tables, analyze the root cause of the performance problem. This analysis 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.

- You can determine what the thread is waiting for:

```
SELECT * FROM performance_schema.events_waits_current
WHERE THREAD_ID = thread_1;
```

Say the query result identifies that the thread is waiting for mutex A, found in `events_waits_current.OBJECT_INSTANCE_BEGIN`.

- You can determine which thread is holding mutex A:

```
SELECT * FROM performance_schema.mutex_instances
WHERE OBJECT_INSTANCE_BEGIN = mutex_A;
```

Say the query result identifies that it is thread 2 holding mutex A, as found in `mutex_instances.LOCKED_BY_THREAD_ID`.

- You can see what thread 2 is doing:

```
SELECT * FROM performance_schema.events_waits_current
WHERE THREAD_ID = thread_2;
```

14.1 Query Profiling Using Performance Schema

The following example demonstrates how to use Performance Schema statement events and stage events to retrieve data comparable to profiling information provided by `SHOW PROFILES` and `SHOW PROFILE` statements.

In this example, statement and stage event data is collected in the `events_statements_history_long` and `events_stages_history_long` tables. On a busy server with many active foreground threads, data could age out of the history tables before you are able to retrieve the information you want to analyze. If you encounter this problem, options include:

- Running the query on a test instance where there is less foreground thread activity.
- Disabling instrumentation for other existing foreground threads by setting the `INSTRUMENTED` field of the `threads` table to `NO` for other thread records. For example, the following statement disables instrumentation for all foreground threads except the `test_user` thread:

```
mysql> UPDATE performance_schema.threads
      SET INSTRUMENTED = 'NO'
      WHERE TYPE='FOREGROUND' AND PROCESSLIST_USER NOT LIKE 'test_user';
```

However, be aware that new threads are always instrumented by default.

- Increasing the number of rows in the `events_statements_history_long` and `events_stages_history_long` tables. The `performance_schema_events_statements_history_size` and `performance_schema_events_stages_history_size` configuration options are autosized by default but can also be set explicitly at startup. You can view current settings by running `SHOW VARIABLES`. For information about autosized Performance Schema parameters, see [Chapter 4, Performance Schema Startup Configuration](#).

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.

- 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/%';
```

2. 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_%';
```

3. 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 |
+-----+-----+-----+-----+-----+-----+
```

4. 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 |
+-----+-----+-----+
```

5. 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 |
+-----+-----+
```

