MySQL Performance Schema

Abstract

This is the MySQL Performance Schema extract from the MySQL 5.5 Reference Manual.

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

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

Document generated on: 2016-08-12 (revision: 48538)

Table of Contents

Preface and Legal Notices

This is the MySQL Performance Schema extract from the MySQL 5.5 Reference Manual.

Legal Notices

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 is available as of MySQL 5.5.3 and 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) and file I/O calls 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).
- Current events are available, as well as event histories and summaries. This enables you to determine how many times instrumented activities were performed and how much time they took. Event information is available to show the activities of specific threads, or activity associated with particular objects such as a mutex or file.
- The `PERFORMANCE_SCHEMA` storage engine collects event data using “instrumentation points” in server source code.
- Collected events are stored in tables in the `performance_schema` database. These tables can be queried using `SELECT` statements like other tables.
- Performance Schema configuration can be modified dynamically by updating tables in the `performance_schema` database through SQL statements. Configuration changes affect data collection immediately.
- Tables in the `performance_schema` database are views or temporary tables that use no persistent on-disk storage.
- Monitoring is available on all platforms supported by MySQL.

  Some limitations might apply: The types of timers might vary per platform. Instruments that apply to storage engines might not be implemented for all storage engines. Instrumentation of each third-party engine is the responsibility of the engine maintainer. See also Restrictions on Performance Schema.
- Data collection is implemented by modifying the server source code to add instrumentation. There are no separate threads associated with the Performance Schema, unlike other features such as replication or the Event Scheduler.

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

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

Chapter 2 Performance Schema Quick Start

This section briefly introduces the Performance Schema with examples that show how to use it. For additional examples, see Chapter 12, *Using the Performance Schema to Diagnose Problems*.

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

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

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

Assuming that the Performance Schema is available, it is disabled by default. To enable it, start the server with the `performance_schema` variable enabled. For example, use these lines in your `my.cnf` file:

```
[mysqld]
performance_schema
```

When the server starts, it sees `performance_schema` and attempts to initialize the Performance Schema. To verify successful initialization, use this statement:

```
mysql> SHOW VARIABLES LIKE 'performance_schema';
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| performance_schema | ON    |
+--------------------+-------+
```

A value of `ON` means that the Performance Schema initialized successfully and is ready for use. A value of `OFF` means that some error occurred. Check the server error log for information about what went wrong.

The Performance Schema is implemented as a storage engine. If this engine is available (which you should already have checked earlier), you should see it listed with a `SUPPORT` value of `YES` in the output from the `INFORMATION_SCHEMA.ENGINES` table or the `SHOW ENGINES` statement:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.ENGINES
    -> WHERE ENGINE='PERFORMANCE_SCHEMA'\G
*************************** 1. row ***************************
      ENGINE: PERFORMANCE_SCHEMA
     SUPPORT: YES
     COMMENT: Performance Schema
TRANSACTIONS: NO
          XA: NO
  SAVEPOINTS: NO
mysql> SHOW ENGINES\G
...
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
```

```
  Savepoints: NO
...
```

The PERFORMANCE_SCHEMA storage engine operates on tables in the performance_schema database. You can make performance_schema the default database so that references to its tables need not be qualified with the database name:

```
mysql> USE performance_schema;
```

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

Performance Schema tables are stored in the performance_schema database. Information about the structure of this database and its tables can be obtained, as for any other database, by selecting from the INFORMATION_SCHEMA database or by using SHOW statements. For example, use either of these statements to see what Performance Schema tables exist:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA = 'performance_schema';
+----------------------------------------------+
| TABLE_NAME                                   |
+----------------------------------------------+
| cond_instances                               |
| events_waits_current                         |
| events_waits_history                         |
| events_waits_history_long                    |
| events_waits_summary_by_instance             |
| events_waits_summary_by_thread_by_event_name |
| events_waits_summary_global_by_event_name    |
| file_instances                               |
| file_summary_by_event_name                   |
| file_summary_by_instance                     |
| mutex_instances                              |
| performance_timers                           |
| rwlock_instances                             |
| setup_consumers                              |
| setup_instruments                            |
| setup_timers                                 |
| threads                                      |
+----------------------------------------------+
mysql> SHOW TABLES FROM performance_schema;
+----------------------------------------------+
| Tables_in_performance_schema                 |
+----------------------------------------------+
| cond_instances                               |
| events_waits_current                         |
| events_waits_history                         |
...
```

The number of Performance Schema tables is expected to increase over time as implementation of additional instrumentation proceeds.

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

> **Note**
>
> Before MySQL 5.5.8, the table names were uppercase, which caused problems on some systems for certain values of the lower_case_table_names system variable.

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

```
mysql> SHOW CREATE TABLE setup_timers\G
*************************** 1. row ***************************
       Table: setup_timers
```

```
Create Table: CREATE TABLE `setup_timers` (
  `NAME` varchar(64) NOT NULL,
  `TIMER_NAME` enum('CYCLE','NANOSECOND','MICROSECOND','MILLISECOND','TICK')
   NOT NULL
) ENGINE=PERFORMANCE_SCHEMA DEFAULT CHARSET=utf8
```

Table structure is also available by selecting from tables such as `INFORMATION_SCHEMA.COLUMNS` or by using statements such as `SHOW COLUMNS`.

Tables in the `performance_schema` database can be grouped according to the type of information in them: Current events, event histories and summaries, object instances, and setup (configuration) information. The following examples illustrate a few uses for these tables. For detailed information about the tables in each group, see Chapter 8, *Performance Schema Table Descriptions*.

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

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

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

- The ID columns indicate which thread the event comes from and the event number.
- `EVENT_NAME` indicates what was instrumented and `SOURCE` indicates which source file contains the instrumented code.
- The timer columns show when the event started and stopped and how long it took. If an event is still in progress, the `TIMER_END` and `TIMER_WAIT` values are `NULL`. Timer values are approximate and expressed in picoseconds. For information about timers and event time collection, see Section 3.3.1, “Performance Schema Event Timing”.

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

```
mysql> SELECT EVENT_ID, EVENT_NAME, TIMER_WAIT
    -> FROM events_waits_history WHERE THREAD_ID = 13
    -> ORDER BY EVENT_ID;
+----------+----------------------------------------+------------+
| EVENT_ID | EVENT_NAME                             | TIMER_WAIT |
+----------+----------------------------------------+------------+
|       86 | wait/synch/mutex/mysys/THR_LOCK::mutex |     686322 |
|       87 | wait/synch/mutex/mysys/THR_LOCK_malloc |     320535 |
```

```
|       88 | wait/synch/mutex/mysys/THR_LOCK_malloc  |     339390 |
|       89 | wait/synch/mutex/mysys/THR_LOCK_malloc  |     377100 |
|       90 | wait/synch/mutex/sql/LOCK_plugin        |     614673 |
|       91 | wait/synch/mutex/sql/LOCK_open          |     659925 |
|       92 | wait/synch/mutex/sql/THD::LOCK_thd_data |     494001 |
|       93 | wait/synch/mutex/mysys/THR_LOCK_malloc  |     222489 |
|       94 | wait/synch/mutex/mysys/THR_LOCK_malloc  |     214947 |
|       95 | wait/synch/mutex/mysys/LOCK_alarm       |     312993 |
+----------+-----------------------------------------+------------+
```

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

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

```
mysql> SELECT EVENT_NAME, COUNT_STAR
    -> FROM events_waits_summary_global_by_event_name
    -> ORDER BY COUNT_STAR DESC LIMIT 10;
+-----------------------------------------------------+------------+
| EVENT_NAME                                          | COUNT_STAR |
+-----------------------------------------------------+------------+
| wait/synch/mutex/mysys/THR_LOCK_malloc              |       6419 |
| wait/io/file/sql/FRM                                |        452 |
| wait/synch/mutex/sql/LOCK_plugin                    |        337 |
| wait/synch/mutex/mysys/THR_LOCK_open                |        187 |
| wait/synch/mutex/mysys/LOCK_alarm                   |        147 |
| wait/synch/mutex/sql/THD::LOCK_thd_data             |        115 |
| wait/io/file/myisam/kfile                           |        102 |
| wait/synch/mutex/sql/LOCK_global_system_variables |         89 |
| wait/synch/mutex/mysys/THR_LOCK::mutex              |         89 |
| wait/synch/mutex/sql/LOCK_open                      |         88 |
+-----------------------------------------------------+------------+
mysql> SELECT EVENT_NAME, SUM_TIMER_WAIT
    -> FROM events_waits_summary_global_by_event_name
    -> ORDER BY SUM_TIMER_WAIT DESC LIMIT 10;
+---------------------------------------+----------------+
| EVENT_NAME                            | SUM_TIMER_WAIT |
+---------------------------------------+----------------+
| wait/io/file/sql/MYSQL_LOG            |     1599816582 |
| wait/synch/mutex/mysys/THR_LOCK_malloc |     1530083250 |
| wait/io/file/sql/binlog_index         |     1385291934 |
| wait/io/file/sql/FRM                  |     1292823243 |
| wait/io/file/myisam/kfile             |      411193611 |
| wait/io/file/myisam/dfile             |      322401645 |
| wait/synch/mutex/mysys/LOCK_alarm     |      145126935 |
| wait/io/file/sql/casetest             |      104324715 |
| wait/synch/mutex/sql/LOCK_plugin      |       86027823 |
| wait/io/file/sql/pid                  |       72591750 |
+---------------------------------------+----------------+
```

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

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

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

```
mysql> SELECT * FROM file_instances\G
```

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

Setup tables are used to configure and display monitoring characteristics. For example, to see which event timer is selected, query the `setup_timers` tables:

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

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

```
mysql> SELECT * FROM setup_instruments;
+-------------------------------------------------------------+---------+-------+
| NAME                                                        | ENABLED | TIMED |
+-------------------------------------------------------------+---------+-------+
...
| wait/synch/mutex/sql/LOCK_global_read_lock                  | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables           | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db                           | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager                           | YES     | YES   |
...
| wait/synch/rwlock/sql/LOCK_grant                            | YES     | YES   |
| wait/synch/rwlock/sql/LOGGER::LOCK_logger                   | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_connect                 | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_slave                   | YES     | YES   |
...
| wait/io/file/sql/binlog                                     | YES     | YES   |
| wait/io/file/sql/binlog_index                               | YES     | YES   |
| wait/io/file/sql/casetest                                   | YES     | YES   |
| wait/io/file/sql/dbopt                                      | YES     | YES   |
...
```

To understand how to interpret instrument names, see Chapter 5, *Performance Schema Instrument Naming Conventions*.

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

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

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

```
mysql> SELECT * FROM setup_consumers;
+-------------------------------------------+---------+
| NAME                                      | ENABLED |
+-------------------------------------------+---------+
| events_waits_current                      | YES     |
```

```
| events_waits_history                            | YES     |
| events_waits_history_long                       | YES     |
| events_waits_summary_by_thread_by_event_name | YES     |
| events_waits_summary_by_event_name              | YES     |
| events_waits_summary_by_instance                | YES     |
| file_summary_by_event_name                      | YES     |
| file_summary_by_instance                        | YES     |
+-------------------------------------------------+---------+
```

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

For more information about the setup tables and how to use them to control event collection, see Section 3.3.2, “Performance Schema Event Filtering”.

There are some miscellaneous tables that do not fall into any of the previous groups. For example, `performance_timers` lists the available event timers and their characteristics. For information about timers, see Section 3.3.1, “Performance Schema Event Timing”.

Chapter 3 Performance Schema Configuration

Table of Contents

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

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

3.1 Performance Schema Build Configuration

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

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

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

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

If you install MySQL over a previous installation that was configured without the Performance Schema (or with an older version of the Performance Schema that may not have all the current tables), run `mysql_upgrade` after starting the server to ensure that the `performance_schema` database exists with all current tables. Then restart the server. One indication that you need to do this is the presence of messages such as the following in the error log:

```
[ERROR] Native table 'performance_schema'.'events_waits_history'
has the wrong structure
[ERROR] Native table 'performance_schema'.'events_waits_history_long'
has the wrong structure
...
```

To verify whether a server was built with Performance Schema support, check its help output. If the Performance Schema is available, the output will mention several variables with names that begin with `performance_schema`:

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

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

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

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

A line for `PERFORMANCE_SCHEMA` in the `SHOW ENGINES` output means that the Performance Schema is available, not that it is enabled. To enable it, you must do so at server startup, as described in the next section.

3.2 Performance Schema Startup Configuration

The Performance Schema is disabled by default. To enable it, start the server with the `performance_schema` variable enabled. For example, use these lines in your `my.cnf` file:

```
[mysqld]
performance_schema
```

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 includes several system variables that provide configuration information:

```
mysql> SHOW VARIABLES LIKE 'perf%';
+----------------------------------------------------+---------+
| Variable_name                                      | Value   |
+----------------------------------------------------+---------+
| performance_schema                                 | ON      |
| performance_schema_events_waits_history_long_size | 10000   |
| performance_schema_events_waits_history_size      | 10      |
| 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_table_handles              | 100000  |
| performance_schema_max_table_instances            | 50000   |
| performance_schema_max_thread_classes             | 50      |
| performance_schema_max_thread_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
```

3.3 Performance Schema Runtime Configuration

Performance Schema setup tables contain information about monitoring configuration:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA = 'performance_schema'
    -> AND TABLE_NAME LIKE 'setup%';
+-------------------+
| TABLE_NAME        |
+-------------------+
| setup_consumers   |
| setup_instruments |
| setup_timers      |
+-------------------+
```

You can examine the contents of these tables to obtain information about Performance Schema monitoring characteristics. If you have the `UPDATE` privilege, you can change Performance Schema operation by modifying setup tables to affect how monitoring occurs. For additional details about these tables, see Section 8.2, “Performance Schema Setup Tables”.

To see which event timer is selected, query the `setup_timers` tables:

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

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. There are only “wait” instruments, so this table has only one row and the timer applies to all instruments.

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

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

```
+------+------------+
| NAME | TIMER_NAME |
+------+------------+
| wait | NANOSECOND |
+------+------------+
```

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

The `setup_instruments` and `setup_consumers` tables list the instruments for which events can be collected and the types of consumers for which event information actually is collected, respectively. Section 3.3.2, “Performance Schema Event Filtering”, discusses how you can modify these tables to affect event collection.

If there are Performance Schema configuration changes that must be made at runtime using SQL statements and you would like these changes to take effect each time the server starts, put the statements in a file and start the server with the `--init-file=file_name` option. This strategy can also be useful if you have multiple monitoring configurations, each tailored to produce a different kind of monitoring, such as casual server health monitoring, incident investigation, application behavior troubleshooting, and so forth. Put the statements for each monitoring configuration into their own file and specify the appropriate file as the `--init-file` argument when you start the server.

3.3.1 Performance Schema Event Timing

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

Performance Schema Timers

Two Performance Schema tables provide timer information:

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

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

Timers vary in precision and amount of overhead. To see what timers are available and their characteristics, check the `performance_timers` table:

```
mysql> SELECT * FROM performance_timers;
+-------------+-----------------+------------------+----------------+
| TIMER_NAME  | TIMER_FREQUENCY | TIMER_RESOLUTION | TIMER_OVERHEAD |
+-------------+-----------------+------------------+----------------+
| CYCLE       |      2389029850 |                1 |             72 |
| NANOSECOND  |      1000000000 |                1 |            112 |
| MICROSECOND |         1000000 |                1 |            136 |
| MILLISECOND |            1036 |                1 |            168 |
| TICK        |             105 |                1 |           2416 |
+-------------+-----------------+------------------+----------------+
```

The columns have these meanings:

- The `TIMER_NAME` column shows the names of the available timers. `CYCLE` refers to the timer that is based on the CPU (processor) cycle counter. The timers in `setup_timers` that you can use are those that do not have `NULL` in the other columns. If the values associated with a given timer name are `NULL`, that timer is not supported on your platform.
- `TIMER_FREQUENCY` indicates the number of timer units per second. For a cycle timer, the frequency is generally related to the CPU speed. The value shown was obtained on a system with a 2.4GHz processor. The other timers are based on fixed fractions of seconds. For `TICK`, the frequency may vary by platform (for example, some use 100 ticks/second, others 1000 ticks/second).

- `TIMER_RESOLUTION` indicates the number of timer units by which timer values increase at a time. If a timer has a resolution of 10, its value increases by 10 each time.
- `TIMER_OVERHEAD` is the minimal number of cycles of overhead to obtain one timing with the given timer. The overhead per event is twice the value displayed because the timer is invoked at the beginning and end of the event.

To see which timer is in effect or to change the timer, access the `setup_timers` table:

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

By default, the Performance Schema uses the best timer available for each instrument type, but you can select a different one. Generally the best timer is `CYCLE`, which uses the CPU cycle counter whenever possible to provide high precision and low overhead.

The precision offered by the cycle counter depends on processor speed. If the processor runs at 1 GHz (one billion cycles/second) or higher, the cycle counter delivers sub-nanosecond precision. Using the cycle counter is much cheaper than getting the actual time of day. For example, the standard `gettimeofday()` function can take hundreds of cycles, which is an unacceptable overhead for data gathering that may occur thousands or millions of times per second.

Cycle counters also have disadvantages:

- End users expect to see timings in wall-clock units, such as fractions of a second. Converting from cycles to fractions of seconds can be expensive. For this reason, the conversion is a quick and fairly rough multiplication operation.
- Processor cycle rate might change, such as when a laptop goes into power-saving mode or when a CPU slows down to reduce heat generation. If a processor's cycle rate fluctuates, conversion from cycles to real-time units is subject to error.
- Cycle counters might be unreliable or unavailable depending on the processor or the operating system. For example, on Pentiums, the instruction is `RDTSC` (an assembly-language rather than a C instruction) and it is theoretically possible for the operating system to prevent user-mode programs from using it.
- Some processor details related to out-of-order execution or multiprocessor synchronization might cause the counter to seem fast or slow by up to 1000 cycles.

MySQL works with cycle counters on x386 (Windows, OS X, Linux, Solaris, and other Unix flavors), PowerPC, and IA-64.

Performance Schema Timer Representation in Events

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

The `setup_instruments` table has an `ENABLED` column to indicate the instruments for which to collect events. The table also has a `TIMED` column to indicate which instruments are timed. If

an instrument is not enabled, it produces no events. If an enabled instrument is not timed, events produced by the instrument have `NULL` for the `TIMER_START`, `TIMER_END`, and `TIMER_WAIT` timer values. This in turn causes those values to be ignored when calculating the sum, minimum, maximum, and average time values in summary tables.

Within events, times are stored in picoseconds (trillionths of a second) to normalize them to a standard unit, regardless of which timer is selected. The timer used for an event is the one in effect when event timing begins. This timer is used to convert start and end values to picoseconds for storage in the event.

Modifications to the `setup_timers` table affect monitoring immediately. Events already measured are stored using the original timer unit, and events in progress may use the original timer for the begin time and the new timer for the end time. To avoid unpredictable results if 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.

3.3.2 Performance Schema Event Filtering

Events are processed in a producer/consumer fashion:

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

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

```
...
```

- 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:

```
mysql> SELECT * FROM setup_consumers;
+----------------------------------------------+---------+
| NAME                                         | ENABLED |
+----------------------------------------------+---------+
| events_waits_current                         | YES     |
| events_waits_history                         | YES     |
| events_waits_history_long                    | YES     |
| events_waits_summary_by_thread_by_event_name | YES     |
| events_waits_summary_by_event_name           | YES     |
| events_waits_summary_by_instance             | YES     |
| file_summary_by_event_name                   | YES     |
| file_summary_by_instance                     | YES     |
+----------------------------------------------+---------+
```

Filtering can be done at different stages of performance monitoring:

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

  Reasons to use pre-filtering:

  - To reduce overhead. Performance Schema overhead should be minimal even with all instruments enabled, but perhaps you want to reduce it further. Or you do not care about timing events and want to disable the timing code to eliminate timing overhead.
  - To avoid filling the current-events or history tables with events in which you have no interest. Pre-filtering leaves more “room” in these tables for instances of rows for enabled instrument types. If you enable only file instruments with pre-filtering, no rows are collected for nonfile instruments. With post-filtering, nonfile events are collected, leaving fewer rows for file events.
  - To avoid maintaining some kinds of event tables. If you disable a consumer, the server does not spend time maintaining destinations for that consumer. For example, if you do not care about event histories, you can disable the history table consumers to improve performance.

- **Post-filtering.** This involves the use of `WHERE` clauses in queries that select information from Performance Schema tables, to specify which of the available events you want to see. Post-filtering is performed on a per-user basis because individual users select which of the available events are of interest.

  Reasons to use post-filtering:

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

The following sections provide more detail about pre-filtering and provide guidelines for naming instruments or consumers in filtering operations. For information about writing queries to retrieve information (post-filtering), see Chapter 4, *Performance Schema Queries*.

3.3.3 Event Pre-Filtering

Pre-filtering is done by modifying Performance Schema configuration so that only certain types of events are collected from producers, and collected events update only certain consumers. This type of 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 affect pre-filtering at the producer stage, modify the `setup_instruments` table. An instrument can be enabled or disabled by setting its `ENABLED` value to `YES` or `NO`. An instrument can be configured whether to collect timing information by setting its `TIMED` value to `YES` or `NO`.
- To affect pre-filtering at the consumer stage, modify the `setup_consumers` table. A consumer can be enabled or disabled by setting its `ENABLED` value to `YES` or `NO`.

Here are some examples that show the types of pre-filtering operations available:

- Disable all instruments:

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

  Now no events will be collected. This change, like other pre-filtering operations, affects other users as well, even if they want to see event information.

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

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

- Disable only file instruments, enable all other instruments:

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

  The preceding queries use the `LIKE` operator and the pattern `'wait/io/file/%'` to match all instrument names that begin with `'wait/io/file/`. For additional information about specifying patterns to select instruments, see Section 3.3.4, “Naming Instruments or Consumers for Filtering Operations”.

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

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

- Disable a specific instrument:

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

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

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

- Disable timing for all events:

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

Setting the `TIMED` column for instruments affects Performance Schema table contents as described in Section 3.3.1, “Performance Schema Event Timing”.

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

After making instrumentation changes, you might want to truncate the summary tables to clear aggregate information for previously collected events. The effect of `TRUNCATE TABLE` for summary tables is to reset the summary columns to 0 or `NULL`, not to remove rows.

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

```
mysql> UPDATE setup_consumers
    -> SET ENABLED = 'NO' WHERE NAME LIKE '%history%';
```

3.3.4 Naming Instruments or Consumers for Filtering Operations

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

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

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

```
mysql> UPDATE setup_instruments
    -> SET ENABLED = 'NO'
    -> WHERE NAME LIKE 'wait/synch/mutex/%';
mysql> UPDATE setup_consumers
    -> SET ENABLED = 'NO' WHERE NAME LIKE '%history%';
```

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

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

A pattern of `'%/file/%'` will match other instruments that have a component of `'/file/'` anywhere in the name. Even less suitable is the pattern `'%file%'` because it will match instruments with `'file'` anywhere in the name, such as `wait/synch/mutex/sql/LOCK_des_key_file`.

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

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

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

3.3.5 Determining What Is Instrumented

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

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

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

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

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

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

In Section 3.3.3, “Event Pre-Filtering”, an example showed how to pre-filter for file instruments. If the event tables contain both file and nonfile information, post-filtering is another way to see information only for file events. Add a `WHERE` clause to queries to restrict event selection appropriately:

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

Chapter 5 Performance Schema Instrument Naming Conventions

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

```
wait/io/file/myisam/log
wait/io/file/mysys/charset
wait/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/innodb/trx_sys_lock
wait/synch/rwlock/sql/Query_cache_query::lock
```

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`.

In MySQL 5.5, there is a single top-level component, `wait`, indicating a wait instrument. The naming tree for wait instruments has this structure:

- `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/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 6 Performance Schema Status Monitoring

There are several status variables associated with the Performance Schema:

```
mysql> SHOW STATUS LIKE 'perf%';
+-----------------------------------------+-------+
| Variable_name                           | Value |
+-----------------------------------------+-------+
| Performance_schema_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_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_table_handles_lost   | 0     |
| Performance_schema_table_instances_lost | 0     |
| Performance_schema_thread_classes_lost  | 0     |
| Performance_schema_thread_instances_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: performance_schema
  Name: performance_schema.memory
Status: 26459600
...
```

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

Chapter 7 Performance Schema General Table Characteristics

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

Most tables in the `performance_schema` database are read only and cannot be modified. Some of the setup tables have columns that can be modified to affect Performance Schema operation. Truncation is permitted to clear collected events, so `TRUNCATE TABLE` can be used on tables containing those kinds of information, such as tables named with a prefix of `events_waits_`.

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

Privileges are as for other databases and tables:

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

Chapter 8 Performance Schema Table Descriptions

Table of Contents

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

- Setup tables. These tables are used to configure and display monitoring characteristics.
- Current events table. The `events_waits_current` table contains the most recent event for each thread.
- History tables. These tables have the same structure as `events_waits_current` but contain more rows. The `events_waits_history` table contains the most recent 10 events per thread. `events_waits_history_long` contains the most recent 10,000 events.

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

8.1 Performance Schema Table Index

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

Table 8.1 Performance Schema Tables

Table Name	Description
`cond_instances`	synchronization object instances

Table Name	Description
events_waits_current	Current wait events
events_waits_history	Most recent wait events for each thread
events_waits_history_long	Most recent wait events overall
events_waits_summary_by_instance	Wait events per instance
events_waits_summary_by_thread_by_event_name	Wait events per thread 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
mutex_instances	Mutex synchronization object instances
performance_timers	Which event timers are available
rwlock_instances	Lock synchronization object instances
setup_consumers	Consumers for which event information can be stored
setup_instruments	Classes of instrumented objects for which events can be collected
setup_timers	Current event timer
threads	Information about server threads

8.2 Performance Schema Setup Tables

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

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

These setup tables are available:

- setup_consumers: The types of consumers for which event information can be stored
- setup_instruments: The classes of instrumented objects for which events can be collected
- setup_timers: The current event timer

8.2.1 The setup_consumers Table

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

```
mysql> SELECT * FROM setup_consumers;
+----------------------------------------------+---------+
| NAME                                         | ENABLED |
+----------------------------------------------+---------+
| events_waits_current                         | YES     |
| events_waits_history                         | YES     |
| events_waits_history_long                    | YES     |
| events_waits_summary_by_thread_by_event_name | YES     |
| events_waits_summary_by_event_name           | YES     |
| events_waits_summary_by_instance             | YES     |
| file_summary_by_event_name                   | YES     |
| file_summary_by_instance                     | YES     |
```

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

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.

Disabling the `events_waits_current` consumer disables everything else that depends on waits, such as the `events_waits_history` and `events_waits_history_long` tables, and all summary tables.

8.2.2 The setup_instruments Table

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

```
mysql> SELECT * FROM setup_instruments;
+-------------------------------------------------------------+---------+-------+
| NAME                                                        | ENABLED | TIMED |
+-------------------------------------------------------------+---------+-------+
...
| wait/synch/mutex/sql/LOCK_global_read_lock                  | YES     | YES   |
| wait/synch/mutex/sql/LOCK_global_system_variables           | YES     | YES   |
| wait/synch/mutex/sql/LOCK_lock_db                           | YES     | YES   |
| wait/synch/mutex/sql/LOCK_manager                           | YES     | YES   |
...
| wait/synch/rwlock/sql/LOCK_grant                            | YES     | YES   |
| wait/synch/rwlock/sql/LOGGER::LOCK_logger                   | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_connect                 | YES     | YES   |
| wait/synch/rwlock/sql/LOCK_sys_init_slave                   | YES     | YES   |
...
| wait/io/file/sql/binlog                                     | YES     | YES   |
| wait/io/file/sql/binlog_index                               | YES     | YES   |
| wait/io/file/sql/casetest                                   | YES     | YES   |
| wait/io/file/sql/dbopt                                      | YES     | YES   |
...
```

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

The `setup_instruments` table has these columns:

- `NAME`

  The instrument name. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 5, *Performance Schema Instrument Naming Conventions*. Events produced from execution of an instrument have an `EVENT_NAME` value that is taken from the instrument `NAME` value. (Events do not really have a “name,” but this provides a way to associate events with instruments.)

- `ENABLED`

  Whether the instrument is enabled. The value is `YES` or `NO`. This column can be modified. A disabled instrument produces no events.

- `TIMED`

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

If an enabled instrument is not timed, the instrument code is enabled, but the timer is not. Events produced by the instrument have `NULL` for the `TIMER_START`, `TIMER_END`, and `TIMER_WAIT` timer values. This in turn causes those values to be ignored when calculating the sum, minimum, maximum, and average time values in summary tables.

8.2.3 The setup_timers Table

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

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

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

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

The `setup_timers` table has these columns:

- `NAME`

  The type of instrument the timer is used for.

- `TIMER_NAME`

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

8.3 Performance Schema Instance Tables

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

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

These tables list instrumented synchronization objects and files. There are three types of synchronization objects: `cond`, `mutex`, and `rwlock`. Each instance table has an `EVENT_NAME` or `NAME` column to indicate the instrument associated with each row. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 5, *Performance Schema Instrument Naming Conventions*.

The `mutex_instances.LOCKED_BY_THREAD_ID` and `rwlock_instances.WRITE_LOCKED_BY_THREAD_ID` columns are extremely important for investigating performance bottlenecks or deadlocks. For examples of how to use them for this purpose, see Chapter 12, *Using the Performance Schema to Diagnose Problems*

8.3.1 The cond_instances Table

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

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

The `cond_instances` table has these columns:

- `NAME`

  The instrument name associated with the condition.

- `OBJECT_INSTANCE_BEGIN`

  The address in memory of the instrumented condition.

8.3.2 The file_instances Table

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

The `file_instances` table has these columns:

- `FILE_NAME`

  The file name.

- `EVENT_NAME`

  The instrument name associated with the file.

- `OPEN_COUNT`

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

8.3.3 The mutex_instances Table

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

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

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

The `mutex_instances` table has these columns:

- `NAME`

  The instrument name associated with the mutex.

- `OBJECT_INSTANCE_BEGIN`

  The address in memory of the instrumented mutex.

- `LOCKED_BY_THREAD_ID`

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

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

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

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

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

8.3.4 The rwlock_instances Table

The `rwlock_instances` table lists all the `rwlock` 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.

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

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

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

8.4 Performance Schema Wait Event Tables

These tables store wait events:

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

The following sections describe those tables. There are also summary tables that aggregate information about wait events; see Section 8.5.1, “Event Wait Summary Tables”.

Wait Event Configuration

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

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

```
mysql> SELECT * FROM setup_instruments
    -> WHERE NAME LIKE 'wait/io/file/innodb%';
+---------------------------------------+---------+-------+
| NAME                                  | ENABLED | TIMED |
+---------------------------------------+---------+-------+
| wait/io/file/innodb/innodb_data_file  | YES     | YES   |
| wait/io/file/innodb/innodb_log_file   | YES     | YES   |
| wait/io/file/innodb/innodb_temp_file  | YES     | YES   |
+---------------------------------------+---------+-------+
mysql> SELECT * FROM setup_instruments WHERE
    -> NAME LIKE 'wait/io/socket/%';
+----------------------------------------+---------+-------+
| NAME                                   | ENABLED | TIMED |
+----------------------------------------+---------+-------+
| wait/io/socket/sql/server_tcpip_socket | NO      | NO    |
| wait/io/socket/sql/server_unix_socket  | NO      | NO    |
```

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

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

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

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

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

To enable all wait consumers, do this:

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

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

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

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

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

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

8.4.1 The events_waits_current Table

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

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

Of the tables that contain wait event rows, `events_waits_current` is the most fundamental. Other tables that contain wait event rows are logically derived from the current events. For example, the `events_waits_history` and `events_waits_history_long` tables are collections of the most recent wait events, up to a fixed number of rows.

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

The `events_waits_current` table has these columns:

- THREAD_ID, EVENT_ID

  The thread associated with the event and the thread current event number when the event starts. The THREAD_ID and EVENT_ID values taken together form a primary key that uniquely identifies the row. No two rows will have the same pair of values.

- EVENT_NAME

  The name of the instrument that produced the event. This is a NAME value from the setup_instruments table. Instrument names may have multiple parts and form a hierarchy, as discussed in Chapter 5, *Performance Schema Instrument Naming Conventions*.

- SOURCE

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

- TIMER_START, TIMER_END, TIMER_WAIT

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

  If an event has not finished, TIMER_END and TIMER_WAIT are NULL.

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

  For discussion of picoseconds as the unit for event times and factors that affect time values, see Section 3.3.1, “Performance Schema Event Timing”.

- SPINS

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

- OBJECT_SCHEMA, OBJECT_NAME, OBJECT_TYPE, OBJECT_INSTANCE_BEGIN

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

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

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

  For a file I/O object:

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

  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.

- `NESTING_EVENT_ID`

  Always `NULL`.

- `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.

- `FLAGS`

  Reserved for future use.

8.4.2 The events_waits_history Table

The `events_waits_history` table contains the most recent 10 wait events per thread. To change the table size, modify the `performance_schema_events_waits_history_size` system variable at server startup. Wait events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full.

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

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

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

8.4.3 The events_waits_history_long Table

The `events_waits_history_long` table contains the most recent 10,000 wait events. To change the table size, modify the `performance_schema_events_waits_history_long_size` system variable at server startup. Wait events are not added to the table until they have ended. As new events are added, older events are discarded if the table is full. When a thread ends, its rows are removed from the table.

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

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

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

8.5 Performance Schema Summary Tables

Summary tables provide aggregated information for terminated events over time. The tables in this group summarize event data in different ways.

Event Wait Summaries:

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

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

File I/O Summaries:

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

The `events_waits_summary_global_by_event_name` table was named `EVENTS_WAITS_SUMMARY_BY_EVENT_NAME` before MySQL 5.5.7.

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`. The effect is to reset the summary columns to 0 or `NULL`, not to remove rows. This enables you to clear collected values and restart aggregation. That might be useful, for example, after you have made a runtime configuration change.

8.5.1 Event Wait Summary Tables

The Performance Schema maintains tables for collecting current and recent wait events, and aggregates that information in summary tables. Section 8.4, “Performance Schema Wait Event Tables” describes the events on which wait summaries are based. See that discussion for information about the content of wait events, the current and recent wait event tables, and how to control wait event collection.

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

- `events_waits_summary_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.
- `events_waits_summary_by_instance` has `EVENT_NAME` and `OBJECT_INSTANCE_BEGIN` columns. Each row summarizes events for a given event name and object. If an instrument is used to create multiple instances, each instance has a unique `OBJECT_INSTANCE_BEGIN` value, so these instances are summarized separately in this table.
- `events_waits_summary_by_thread_by_event_name` has `THREAD_ID` and `EVENT_NAME` columns. Each row summarizes events for a given thread and event name.

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

- `COUNT_STAR`

  The number of summarized events. This value includes all events, whether timed or nontimed.

- `SUM_TIMER_WAIT`

  The total wait time of the summarized timed events. This value is calculated only for timed events because nontimed events have a wait time of `NULL`. The same is true for the other *`xxx`*`_TIMER_WAIT` values.

- `MIN_TIMER_WAIT`

  The minimum wait time of the summarized timed events.

- `AVG_TIMER_WAIT`

  The average wait time of the summarized timed events.

- `MAX_TIMER_WAIT`

  The maximum wait time of the summarized timed events.

Example wait event summary information:

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

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

8.5.2 File I/O Summary Tables

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

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

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

Each file I/O summary table has these summary columns containing aggregated values:

- `COUNT_READ`

  The number of read operations in the summarized events.

- `COUNT_WRITE`

  The number of write operations in the summarized events.

- `SUM_NUMBER_OF_BYTES_READ`

  The number of bytes read in the summarized events.

- `SUM_NUMBER_OF_BYTES_WRITE`

  The number of bytes written in the summarized events.

Example file I/O event summary information:

```
mysql> SELECT * FROM file_summary_by_instance\G
...
*************************** 2. row ***************************
               FILE_NAME: /var/mysql/share/english/errmsg.sys
              EVENT_NAME: wait/io/file/sql/ERRMSG
              COUNT_READ: 3
             COUNT_WRITE: 0
 SUM_NUMBER_OF_BYTES_READ: 42211
SUM_NUMBER_OF_BYTES_WRITE: 0
...
*************************** 6. row ***************************
               FILE_NAME: /var/mysql/data/binlog.000001
              EVENT_NAME: wait/io/file/sql/binlog
              COUNT_READ: 0
             COUNT_WRITE: 0
 SUM_NUMBER_OF_BYTES_READ: 0
SUM_NUMBER_OF_BYTES_WRITE: 0
...
```

`TRUNCATE TABLE` is permitted for file I/O summary tables. It resets the summary columns to zero rather than removing rows.

The MySQL server uses several techniques to avoid I/O operations by caching information read from files, so it is possible that statements you might expect to result in I/O events will not. You may be able to ensure that I/O does occur by flushing caches or restarting the server to reset its state.

8.6 Performance Schema Miscellaneous Tables

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

- `performance_timers`: Which event timers are available
- `threads`: Information about server threads

8.6.1 The performance_timers Table

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

```
mysql> SELECT * FROM performance_timers;
+-------------+-----------------+------------------+----------------+
| TIMER_NAME  | TIMER_FREQUENCY | TIMER_RESOLUTION | TIMER_OVERHEAD |
+-------------+-----------------+------------------+----------------+
| CYCLE       |      2389029850 |                1 |             72 |
| NANOSECOND  |            NULL |             NULL |           NULL |
| MICROSECOND |         1000000 |                1 |            585 |
| MILLISECOND |            1035 |                1 |            738 |
| TICK        |             101 |                1 |            630 |
+-------------+-----------------+------------------+----------------+
```

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 `performance_timers` table has these columns:

- `TIMER_NAME`

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

- `TIMER_FREQUENCY`

  The number of timer units per second. For a cycle timer, the frequency is generally related to the CPU speed. For example, on a system with a 2.4GHz processor, the `CYCLE` may be close to 2400000000.

- `TIMER_RESOLUTION`

  Indicates the number of timer units by which timer values increase. If a timer has a resolution of 10, its value increases by 10 each time.

- `TIMER_OVERHEAD`

  The minimal number of cycles of overhead to obtain one timing with the given timer. The Performance Schema determines this value by invoking the timer 20 times during initialization and picking the smallest value. The total overhead really is twice this amount because the instrumentation invokes the timer at the start and end of each event. The timer code is called only for timed events, so this overhead does not apply for nontimed events.

8.6.2 The threads Table

The `threads` table contains a row for each server thread:

```
mysql> SELECT * FROM threads;
+-----------+----------------+----------------------------------------+
| THREAD_ID | PROCESSLIST_ID | NAME                                   |
+-----------+----------------+----------------------------------------+
|         0 |              0 | thread/sql/main                        |
|         1 |              0 | thread/innodb/io_handler_thread        |
|        16 |              0 | thread/sql/signal_handler              |
|        23 |              7 | thread/sql/one_connection              |
|         5 |              0 | thread/innodb/io_handler_thread        |
|        12 |              0 | thread/innodb/srv_lock_timeout_thread  |
|        22 |              6 | thread/sql/one_connection              |
...
```

> **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`

  This is the unique identifier of an instrumented thread.

- `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 0, so the values are not unique.

  This column was named `ID` before MySQL 5.5.8.

- `NAME`

  `NAME` is the name associated with the instrumentation of the 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.

The `threads` table was named `PROCESSLIST` before MySQL 5.5.6.

Removing a plugin with `UNINSTALL PLUGIN` does not affect information already collected for code in that plugin. Time spent executing the code while the plugin was loaded was still spent even if the plugin is unloaded later. The associated event information, including aggregate information, remains readable in `performance_schema` database tables. For additional information about the effect of plugin installation and removal, see Chapter 6, *Performance Schema Status Monitoring*.

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

Chapter 10 Performance Schema System Variables

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

```
mysql> SHOW VARIABLES LIKE 'perf%';
+----------------------------------------------------+---------+
| Variable_name                                      | Value   |
+----------------------------------------------------+---------+
| performance_schema                                 | ON      |
| performance_schema_events_waits_history_long_size | 10000   |
| performance_schema_events_waits_history_size       | 10      |
| 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_table_handles               | 100000  |
| performance_schema_max_table_instances             | 50000   |
| performance_schema_max_thread_classes              | 50      |
| performance_schema_max_thread_instances            | 1000    |
+----------------------------------------------------+---------+
```

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.

Performance Schema system variables have the following meanings:

- `performance_schema`

Introduced	5.5.3	
Command-Line Format	`--performance_schema=#`	
System Variable	**Name**	`performance_schema`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`boolean`
	Default	`OFF`

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

- `performance_schema_events_waits_history_long_size`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_events_waits_history_long_size=#`	
System Variable	**Name**	`performance_schema_events_waits_history_long_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`

	Default	`10000`

The number of rows in the `events_waits_history_long` table.

- `performance_schema_events_waits_history_size`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_events_waits_history_size=#`	
System Variable	**Name**	`performance_schema_events_waits_history_size`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`10`

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

- `performance_schema_max_cond_classes`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_cond_classes=#`	
System Variable	**Name**	`performance_schema_max_cond_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`80`

The maximum number of condition instruments.

- `performance_schema_max_cond_instances`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_cond_instances=#`	
System Variable	**Name**	`performance_schema_max_cond_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`1000`

The maximum number of instrumented condition objects.

- `performance_schema_max_file_classes`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_file_classes=#`	
System Variable	**Name**	`performance_schema_max_file_classes`

	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`50`

The maximum number of file instruments.

- `performance_schema_max_file_handles`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_file_handles=#`	
System Variable	**Name**	`performance_schema_max_file_handles`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`32768`

The maximum number of opened file objects.

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

- `performance_schema_max_file_instances`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_file_instances=#`	
System Variable	**Name**	`performance_schema_max_file_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`10000`

The maximum number of instrumented file objects.

- `performance_schema_max_mutex_classes`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_mutex_classes=#`	
System Variable	**Name**	`performance_schema_max_mutex_classes`
	Variable Scope	Global
	Dynamic Variable	No

Permitted Values	**Type**	`integer`
	Default	`200`

The maximum number of mutex instruments.

- `performance_schema_max_mutex_instances`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_mutex_instances=#`	
System Variable	**Name**	`performance_schema_max_mutex_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`1000`

The maximum number of instrumented mutex objects.

- `performance_schema_max_rwlock_classes`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_rwlock_classes=#`	
System Variable	**Name**	`performance_schema_max_rwlock_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values (<= 5.5.6)	**Type**	`integer`
	Default	`20`
Permitted Values (>= 5.5.7)	**Type**	`integer`
	Default	`30`

The maximum number of rwlock instruments.

- `performance_schema_max_rwlock_instances`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_rwlock_instances=#`	
System Variable	**Name**	`performance_schema_max_rwlock_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`1000`

The maximum number of instrumented rwlock objects.

- `performance_schema_max_table_handles`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_table_handles=#`	
System Variable	**Name**	`performance_schema_max_table_handles`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`100000`

The maximum number of opened table objects.

- `performance_schema_max_table_instances`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_table_instances=#`	
System Variable	**Name**	`performance_schema_max_table_instances`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`50000`

The maximum number of instrumented table objects.

- `performance_schema_max_thread_classes`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_thread_classes=#`	
System Variable	**Name**	`performance_schema_max_thread_classes`
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	**Type**	`integer`
	Default	`50`

The maximum number of thread instruments.

- `performance_schema_max_thread_instances`

Introduced	5.5.3	
Command-Line Format	`--performance_schema_max_thread_instances=#`	
System Variable	**Name**	`performance_schema_max_thread_instances`
	Variable Scope	Global
	Dynamic Variable	No

Permitted Values	**Type**	`integer`
	Default	`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.

The `max_connections` and `max_delayed_threads` system variables affect how many threads are run in the server. `performance_schema_max_thread_instances` affects how many of these running threads can be instrumented. If you increase `max_connections` or `max_delayed_threads`, you should consider increasing `performance_schema_max_thread_instances` so that `performance_schema_max_thread_instances` is greater than the sum of `max_connections` and `max_delayed_threads`.

Chapter 11 Performance Schema Status Variables

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

```
mysql> SHOW STATUS LIKE 'perf%';
+-----------------------------------------+-------+
| Variable_name                           | Value |
+-----------------------------------------+-------+
| Performance_schema_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_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_table_handles_lost   | 0     |
| Performance_schema_table_instances_lost | 0     |
| Performance_schema_thread_classes_lost  | 0     |
| Performance_schema_thread_instances_lost | 0     |
+-----------------------------------------+-------+
```

Performance Schema status variables have the following meanings:

- `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_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_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_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.

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

Chapter 12 Using the Performance Schema to Diagnose Problems

The Performance Schema is a tool to help a DBA do performance tuning by taking real measurements instead of “wild guesses.” This section demonstrates some ways to use the Performance Schema for this purpose. The discussion here relies on the use of event filtering, which is described in Section 3.3.2, “Performance Schema Event Filtering”.

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

1. Run the use case.
2. Using the Performance Schema tables, analyze the root cause of the performance problem. This analysis will rely heavily on post-filtering.
3. For problem areas that are ruled out, disable the corresponding instruments. For example, if analysis shows that the issue is not related to file I/O in a particular storage engine, disable the file I/O instruments for that engine. Then truncate the history and summary tables to remove previously collected events.
4. Repeat the process at step 1.

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

   At each iteration, investigation should lead closer and closer to the root cause of the problem, as the “signal/noise” ratio will improve, making analysis easier.
5. Once a root cause of performance bottleneck is identified, take the appropriate corrective action, such as:

   - Tune the server parameters (cache sizes, memory, and so forth).
   - Tune a query by writing it differently,
   - Tune the database schema (tables, indexes, and so forth).
   - Tune the code (this applies to storage engine or server developers only).
6. Start again at step 1, to see the effects of the changes on performance.

The `mutex_instances.LOCKED_BY_THREAD_ID` and `rwlock_instances.WRITE_LOCKED_BY_THREAD_ID` columns are extremely important for investigating performance bottlenecks or deadlocks. This is made possible by Performance Schema instrumentation as follows:

1. Suppose that thread 1 is stuck waiting for a mutex.
2. You can determine what the thread is waiting for:

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

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

3. You can determine which thread is holding mutex A:

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

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

4. You can see what thread 2 is doing:

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