MySQL Connect

September 29 – 30, 2012
Hilton San Francisco

Improving Performance with the MySQL Performance Schema

Jesper Wisborg Krogh
Principal Technical Support Engineer, MySQL

Background Information

To login to the virtual machine:

Username: mysqlconnect

Password: mysqlconnect

Root password: Oracle123

Starting and stopping MySQL:

There are two MySQL instances installed in a master-slave configuration. Both are version 5.6.6 and can be started and stopped using mysqld_multi. The master has option group number 0 and the slave has option group number 1.

To start/restart/stop both master and slave:

```
shell# service mysqld start|restart|stop
```

To start/restart/stop just the master:

```
shell# service mysqld start|restart|stop 0
```

To start/restart/stop just the slave:

```
shell# service mysqld start|restart|stop 1
```

To log into the master:

```
shell# mysql
```

To log into the slave:

```
shell# mysql --socket=/var/lib/mysql_slave/mysql.sock
```

The following databases are installed:

- employees: approximately 160M data in 4 million rows.
  https://dev.mysql.com/doc/employee/en/index.html
- sakila: a medium sized sample database with views, stored programs, etc.
  https://dev.mysql.com/doc/sakila/en/index.html
- world: the standard World sample database.
  https://dev.mysql.com/doc/index-other.html
- ps_helper: Mark Leith's ps_helper views and procedures for the Performance Schema.
  http://www.markleith.co.uk/ps_helper/
- ps_tools: Similar to ps_helper (will be loaded during this session). A mix of tools created by Mark Leith and Jesper Krogh.
- mysqlconnect: an empty database.

Running the Tests

The tests in this hands-on lab will be run as the root user – both with respect to the operating system and to MySQL.

You can change to become the root user with the following command in the Linux shell:

```
shell$ su -l
```

Tour of the MySQL Performance Schema

Configuration

We will start out taking a look at how MySQL has been configured with respect to the MySQL Performance Schema.

Starting from MySQL 5.6.6 the Performance Schema is enabled by default, so it is no longer to explicitly enabling it.

However not everything is enabled by default. You have instruments which are the things you can measure, and consumers which are those that use the measurements. Not all instruments and consumers are enabled out of the box, so to ensure we have everything enabled, a few options have been added to the MySQL configuration file. To look at these changes:

```
shell# cat /etc/my.cnf
…
performance_schema_instrument = '%=on'
performance_schema_consumer_events_stages_current            = ON
performance_schema_consumer_events_stages_history            = ON
performance_schema_consumer_events_stages_history_long       = ON
performance_schema_consumer_events_statements_current        = ON
performance_schema_consumer_events_statements_history        = ON
performance_schema_consumer_events_statements_history_long   = ON
performance_schema_consumer_events_waits_current             = ON
performance_schema_consumer_events_waits_history             = ON
performance_schema_consumer_events_waits_history_long        = ON
performance_schema_consumer_global_instrumentation           = ON
performance_schema_consumer_thread_instrumentation           = ON
performance_schema_consumer_statements_digest                = ON
…
```

The first setting `performance_schema_instrument = '%=on'` switched on all instruments (% is a wildcard that matches all instruments).

For the consumers it is necessary to enable each explicitly. This is done by pre-pending the name of the consumer with `performance_schema_consumer_`, for example to enable the `statements_digest` consumer, use the setting and set it to `ON`.

Start MySQL

1. Stop MySQL

```
shell# mysqladmin shutdown
shell# mysqladmin --socket=/var/lib/mysql_slave/mysql.sock shutdown
```

2. Update the MySQL configuration file

Change `innodb_buffer_pool_size` and `innodb_log_file_size` by opening /etc/my.cnf and in the `[mysqld0]` group edit, so:

```
innodb_buffer_pool_size = 100M
innodb_log_file_size    = 6M
```

3. Move the existing log files out of the way

```
shell# mv /var/lib/mysql/ib_log* /tmp
```

4. Start MySQL

```
shell# service mysqld_multi start
```

5. Load Tools

Load some extra Performance Schema tools into MySQL – these are stored in the ps_tools database and are similar to ps_helper.

```
shell# mysql < /tmp/hol/ps_tools_56.sql
shell# mysql --socket=/var/lib/mysql_slave/mysql.sock < /tmp/hol/ps_tools_56.sql
```

6. Connect to the master:

```
shell# mysql performance_schema
```

Performance Schema Variables:

In addition to the options for which instruments and consumers are enabled at start up, there are a number of variables:

Query 1

```
mysql> SHOW GLOBAL VARIABLES LIKE 'performance\_schema%';
+--------------------------------------------------------+-------+
| Variable_name                                          | Value |
+--------------------------------------------------------+-------+
| performance_schema                                     | ON    |
| performance_schema_accounts_size                       | 100   |
| performance_schema_digests_size                        | 5000  |
…
| performance_schema_setup_actors_size                   | 100   |
| performance_schema_setup_objects_size                  | 100   |
| performance_schema_users_size                          | 100   |
+--------------------------------------------------------+-------+
31 rows in set (0.03 sec)
```

These defines the size of the various Performance Schema tables. Several of the values are calculated automatically based on the other settings such as `max_connections`.

As all the Performance Schema data is in-memory changing the size of the tables, affects the memory usage. The memory usage of the Performance Schema can be checked with SHOW ENGINE PERFORMANCE_SCHEMA STATUS:

Query 2

```
mysql> SHOW ENGINE PERFORMANCE_SCHEMA STATUS;
+--------------------+-----------------------------------------+----------+
| Type               | Name                                    | Status   |
+--------------------+-----------------------------------------+----------+
| performance_schema | events_waits_current.row_size           | 184      |
| performance_schema | events_waits_current.row_count          | 2268     |
…
| performance_schema | (user_hash).count                       | 2        |
| performance_schema | (user_hash).size                        | 100      |
| performance_schema | performance_schema.memory               | 77388712 |
+--------------------+-----------------------------------------+----------+
154 rows in set (0.01 sec)
```

The last row with `Name = performance_schema.memory` has the total memory usage for the Performance Schema.

Setup Tables

There are five setup tables for the Performance Schema:

Query 3

```
mysql> SHOW TABLES LIKE 'setup\_%';
+------------------------------------------+
| Tables_in_performance_schema (setup\_%)  |
+------------------------------------------+
| setup_actors                             |
| setup_consumers                          |
| setup_instruments                        |
| setup_objects                            |
| setup_timers                             |
+------------------------------------------+
5 rows in set (0.02 sec)
```

The setup tables include the current settings and allow for dynamically changes of the settings at runtime.

Changes to the setup tables in general takes effect immediately. One exception is changes to `setup_actors` which will only affect new connections.

Note: while it is possible to configure most of the Performance Schema settings dynamically, these changes are not persistent when MySQL restarts.

setup_actors

The `setup_actors` table controls which user accounts are instrumented by default (see also the `threads` table). The `setup_actors` table has the following content per default:

Query 4

```
mysql> SELECT * FROM setup_actors;
+------+------+------+
| HOST | USER | ROLE |
+------+------+------+
| %    | %    | %    |
+------+------+------+
1 row in set (0.00 sec)
```

The `HOST` and `USER` fields correspond to the same fields in `mysql.user`. The `ROLE` field is currently not used.

The rule is that is any row in `setup_actors` matches the user account, the connection will be instrumented.

setup_objects

The table `setup_objects` define which database objects will be instrumented. Currently this can only be configured for tables, however wildcards are allowed.

The default content of the table is:

Query 5

```
mysql> SELECT * FROM setup_objects;
+-------------+--------------------+-------------+---------+-------+
| OBJECT_TYPE | OBJECT_SCHEMA      | OBJECT_NAME | ENABLED | TIMED |
+-------------+--------------------+-------------+---------+-------+
| TABLE       | mysql              | %           | NO      | NO    |
| TABLE       | performance_schema | %           | NO      | NO    |
| TABLE       | information_schema | %           | NO      | NO    |
| TABLE       | %                  | %           | YES     | YES   |
+-------------+--------------------+-------------+---------+-------+
4 rows in set (0.00 sec)
```

For `setup_objects`, the most specific match is used. The difference between `ENABLED` and `TIMED` is whether when a table is instrumented the events are only counted or also timed.

To demonstrate the use of the `setup_objects` table, consider the following example:

Query 6a

```
mysql> TRUNCATE table_io_waits_summary_by_table;
Query OK, 0 rows affected (0.00 sec)
```

This resets the `table_io_waits_summary_by_table` table.

Query 6b

```
mysql> SELECT OBJECT_SCHEMA, OBJECT_NAME, COUNT_STAR, SUM_TIMER_WAIT FROM
table_io_waits_summary_by_table WHERE OBJECT_SCHEMA = 'world' AND
OBJECT_NAME = 'Country';
Empty set (0.00 sec)
```

So the table does not have any row for the `world.Country` table at this point – just as would be expected just after truncating a table.

Query 6c

```
mysql> SELECT COUNT(*) FROM world.Country;
+-----------+
| COUNT(*)  |
+-----------+
|       239 |
+-----------+
1 row in set (0.05 sec)
```

After executing a query using the `world.Country` table, what does `table_io_waits_summary_by_table` now show?

Query 6d

```
mysql> SELECT OBJECT_SCHEMA, OBJECT_NAME, COUNT_STAR, SUM_TIMER_WAIT FROM
table_io_waits_summary_by_table WHERE OBJECT_SCHEMA = 'world' AND
OBJECT_NAME = 'Country';
+---------------+-------------+------------+----------------+
| OBJECT_SCHEMA | OBJECT_NAME | COUNT_STAR | SUM_TIMER_WAIT |
+---------------+-------------+------------+----------------+
| world         | Country     |        240 |    22678351616 |
+---------------+-------------+------------+----------------+
1 row in set (0.00 sec)
```

So there are 240 events for the `world.Country` table now and a total of 226789351616 pico seconds (10^{-12} seconds) has been spent using the table.

Now try the same again, but with a rule in the `setup_objects` table that turns off timing of the events on the `world.Country` table.

Query 6e

```
mysql> INSERT INTO setup_objects VALUES ('TABLE', 'world', 'Country', 'YES',
'NO');
Query OK, 1 row affected (0.01 sec)
```

Query 6f

```
mysql> TRUNCATE table_io_waits_summary_by_table;
Query OK, 0 rows affected (0.00 sec)
```

Query 6g

```
mysql> SELECT OBJECT_SCHEMA, OBJECT_NAME, COUNT_STAR, SUM_TIMER_WAIT FROM
table_io_waits_summary_by_table WHERE OBJECT_SCHEMA = 'world' AND
OBJECT_NAME = 'Country';
+---------------+-------------+------------+----------------+
| OBJECT_SCHEMA | OBJECT_NAME | COUNT_STAR | SUM_TIMER_WAIT |
+---------------+-------------+------------+----------------+
| world         | Country     |          0 |              0 |
+---------------+-------------+------------+----------------+
1 row in set (0.00 sec)
```

Now what is that? We just truncated the `table_io_waits_summary_by_table` table, but there is still content in it! For summary rules, `TRUNCATE` does in general not delete any of the existing rows, instead the counters are set to 0. This is what also happened in this case.

Query 6h

```
mysql> SELECT COUNT(*) FROM world.Country;
+----------+
| COUNT(*) |
+----------+
|      239 |
+----------+
1 row in set (0.00 sec)
```

Query 6i

```
mysql> SELECT OBJECT_SCHEMA, OBJECT_NAME, COUNT_STAR, SUM_TIMER_WAIT FROM
table_io_waits_summary_by_table WHERE OBJECT_SCHEMA = 'world' AND
OBJECT_NAME = 'Country';
+---------------+-------------+------------+----------------+
| OBJECT_SCHEMA | OBJECT_NAME | COUNT_STAR | SUM_TIMER_WAIT |
+---------------+-------------+------------+----------------+
| world         | Country     |        240 |              0 |
+---------------+-------------+------------+----------------+
1 row in set (0.00 sec)
```

Here the effect of setting `TIMED = 'NO'` is that the timer fields (here `SUM_TIMER_WAIT`) is not updated, but we can still see how many times the `world.Country` has been accessed.

Finally we will change back so `world.Country` is instrumented fully again.

Query 6j

```
mysql> DELETE FROM setup_objects WHERE OBJECT_SCHEMA = 'world' AND
OBJECT_NAME = 'Country';
Query OK, 1 row affected (0.02 sec)
```

setup_timers

The `setup_timers` table defines which timer is used for the each of the instrument types.

Query 7a

```
mysql> SELECT * FROM setup_timers;
+-----------+-------------+
| NAME      | TIMER_NAME  |
+-----------+-------------+
| idle      | MICROSECOND |
| wait      | CYCLE       |
| stage     | NANOSECOND  |
| statement | NANOSECOND  |
+-----------+-------------+
4 rows in set (0.00 sec)
```

The `TIMER_NAME` can be set to any of the values available from `performance_timer` table:

Query 7b

```
mysql> SELECT * FROM performance_timers;
+-------------+-----------------+------------------+----------------+
| TIMER_NAME  | TIMER_FREQUENCY | TIMER_RESOLUTION | TIMER_OVERHEAD |
+-------------+-----------------+------------------+----------------+
| CYCLE       |      1454950248 |                1 |             42 |
| NANOSECOND  |      1000000000 |                1 |             91 |
| MICROSECOND |         1000000 |                1 |            105 |
| MILLISECOND |            1142 |                1 |            119 |
| TICK        |             112 |                1 |            882 |
+-------------+-----------------+------------------+----------------+
5 rows in set (0.01 sec)
```

From the performance_timer table, you can also see the timer frequency, resolution, and overhead (in number of cycles) using that particular timer.

setup_instruments

The `setup_instruments` contain one row per instrumentation point in the source code. These are the events that can be collected. It is possible to specify both whether an instrument is producing events and if so whether it is timed; this is very similar to the `setup_objects` table.

Query 8

```
mysql> SELECT * FROM setup_instruments LIMIT 1;
+---------------------------------+---------+-------+
| NAME                            | ENABLED | TIMED |
+---------------------------------+---------+-------+
| wait/synch/mutex/sql/PAGE::lock | YES     | YES   |
+---------------------------------+---------+-------+
1 row in set (0.00 sec)
```

The name is constructed by components which form a hierarchy. The number of components depends on the name. The components are separated by `/`. When `ENABLED` is `YES`, the instrument produces events. `TIMED` is whether the events are times or just counted.

The default for which instruments are enabled can be set in the MySQL configuration file using the performance_schema_instrument option.

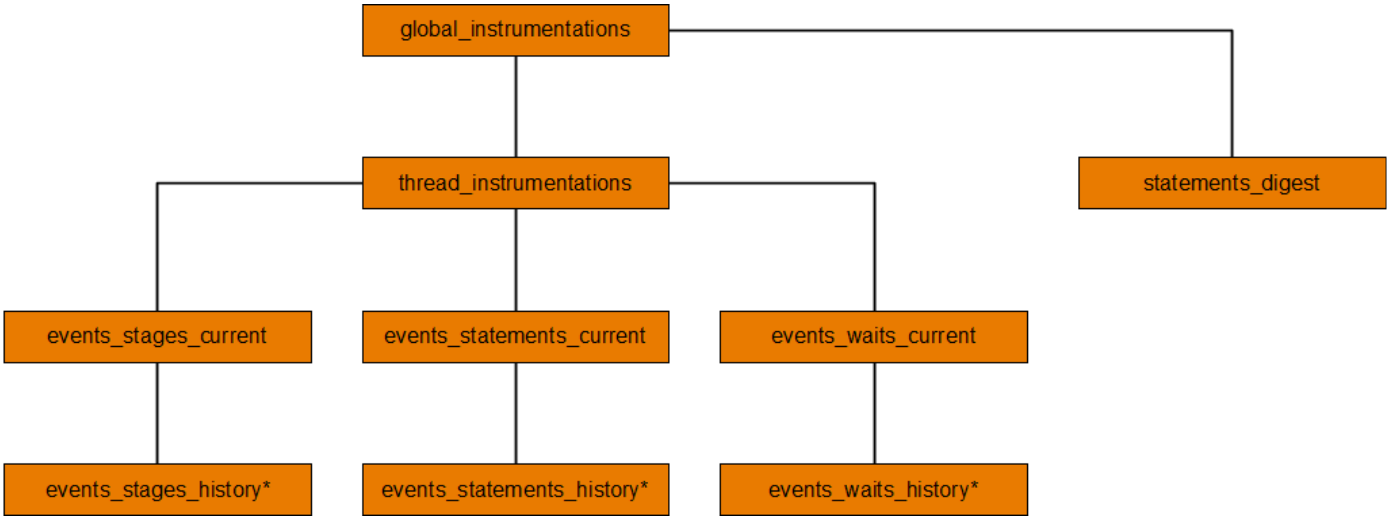
setup_consumers

The last setup table is setup_consumers which lists the consumers of events from the instruments and allows you to specify whether it is enabled or not.

Query 9

```
mysql> SELECT * FROM setup_consumers;
+--------------------------------+---------+
| NAME                           | ENABLED |
+--------------------------------+---------+
| events_stages_current          | YES     |
| events_stages_history          | YES     |
| events_stages_history_long     | YES     |
| events_statements_current      | YES     |
| events_statements_history      | YES     |
| events_statements_history_long | YES     |
| events_waits_current           | YES     |
| events_waits_history           | YES     |
| events_waits_history_long      | YES     |
| global_instrumentation         | YES     |
| thread_instrumentation         | YES     |
| statements_digest              | YES     |
+--------------------------------+---------+
12 rows in set (0.00 sec)
```

The consumers also form a hierarchy:



For a consumer to collect events, not only does it have to be enabled, all consumers above it in the hierarchy must be enabled as well.

Instance Tables

The instance tables include information about the objects being instrumented. They provide event names and explanatory notes or status information. The relation to the setup tables is that the instance table has a `NAME` or `EVENT_NAME` column that corresponds to the `NAME` column in the `setup_instruments` table.

Query 10

```
mysql> SHOW TABLES LIKE '%\_instances';
+--------------------------------------------+
| Tables_in_performance_schema (%\_instances) |
+--------------------------------------------+
| cond_instances                             |
| file_instances                             |
| mutex_instances                            |
| rwlock_instances                           |
| socket_instances                           |
+--------------------------------------------+
5 rows in set (0.01 sec)
```

Event Tables

The event tables are the main entry point for looking at the collected data. There are three groups of event tables depending on the type of event:

- Stages: The same stages as in the `State` column of `SHOW PROCESSLIST`, for example `Sending data`.
- Statements: The SQL statements that have been run on the server.
- Waits: Where the server is spending time – the instrumentations points from `setup_instruments`.

For each event type there are three tables with the actual data collected:

- *_current: the last event for each thread. Note in the case of wait events, some events are *molecular* events, so there can be two events for one thread.
- *_history: the last 10 (by default) events for each thread. The number of events per thread can be configured using the `performance_schema_events_*_history_size` options.
- *_history_long: the last 1000 (by default) events. The size of the table can be configured with the `performance_schema_events_*_history_long_size` options.

Additionally there are a number of summary tables for each event type. The naming convention for the event summary tables is that the table name has two or more parts:

- `event_*_summary`: specifies the event type and it is a summary table.
- One or more _by_<field>: specifies a field the summary is grouped by.

An example is `events_stages_summary_by_account_by_event_name`: a summary of stages grouped by account and event name.

The event stages tables are:

Query 11a

```
mysql> SHOW TABLES LIKE 'events\_stages\_%';
+----------------------------------------------------+
| Tables_in_performance_schema (events\_stages\_%) |
+----------------------------------------------------+
| events_stages_current                              |
| events_stages_history                              |
| events_stages_history_long                         |
| events_stages_summary_by_account_by_event_name     |
| events_stages_summary_by_host_by_event_name        |
| events_stages_summary_by_thread_by_event_name      |
| events_stages_summary_by_user_by_event_name        |
| events_stages_summary_global_by_event_name         |
+----------------------------------------------------+
8 rows in set (0.00 sec)
```

The event statements tables are:

Query 11b

```
mysql> SHOW TABLES LIKE 'events\_statements\_%';
+----------------------------------------------------+
| Tables_in_performance_schema (events\_statements\_%) |
+----------------------------------------------------+
| events_statements_current                          |
| events_statements_history                          |
| events_statements_history_long                     |
| events_statements_summary_by_account_by_event_name |
| events_statements_summary_by_digest                |
| events_statements_summary_by_host_by_event_name    |
| events_statements_summary_by_thread_by_event_name  |
| events_statements_summary_by_user_by_event_name    |
| events_statements_summary_global_by_event_name     |
+----------------------------------------------------+
9 rows in set (0.00 sec)
```

The event waits tables are:

Query 11c

```
mysql> SHOW TABLES LIKE 'events\_waits\_%';
+-----------------------------------------------+
| Tables_in_performance_schema (events\_waits\_%) |
+-----------------------------------------------+
| events_waits_current                          |
| events_waits_history                          |
| events_waits_history_long                     |
| events_waits_summary_by_account_by_event_name |
| events_waits_summary_by_host_by_event_name    |
| events_waits_summary_by_instance              |
| events_waits_summary_by_thread_by_event_name  |
| events_waits_summary_by_user_by_event_name    |
| events_waits_summary_global_by_event_name     |
+-----------------------------------------------+
9 rows in set (0.02 sec)
```

Other Summary Tables

In addition to the event summary tables above, there are also a few other summary tables:

- For objects (effectively per table)
- For files
- For table I/O and Lock Wait
- For sockets

Connection Tables

There are tables showing the current and total number of connections per user, host, or account (user@host). For example for accounts:

Query 12

```
mysql> SELECT * FROM accounts;
+------+-----------+---------------------+-------------------+
| USER | HOST      | CURRENT_CONNECTIONS | TOTAL_CONNECTIONS |
+------+-----------+---------------------+-------------------+
| NULL | NULL      |                  18 |                20 |
| root | localhost |                   2 |                 3 |
+------+-----------+---------------------+-------------------+
2 rows in set (0.01 sec)
```

This shows another aspect of the Performance Schema: note the row having both `USER` and `HOST` set to `NULL`. That is for the background threads, so not only can the Performance Schema give information about the client connections (foreground threads), it can also give insight into what the internal threads such as the InnoDB threads are doing.

Connection Attribute Tables

Related to the connection tables are the two tables giving access to connection attributes:

- `session_account_connect_attrs`
- `session_connect_attrs`

Query 13a

```
mysql> SELECT * FROM session_connect_attrs;
+----------------+-----------------+------------+------------------+
| PROCESSLIST_ID | ATTR_NAME       | ATTR_VALUE | ORDINAL_POSITION |
+----------------+-----------------+------------+------------------+
|              1 | _os             | linux2.6   |                0 |
|              1 | _client_name    | libmysql   |                1 |
|              1 | _pid            | 4671       |                2 |
|              1 | _client_version | 5.6.6-m9   |                3 |
|              1 | _platform       | x86_64     |                4 |
|              1 | program_name    | mysql      |                5 |
+----------------+-----------------+------------+------------------+
6 rows in set (0.00 sec)
```

The difference between the two tables is that `session_connect_attrs` includes the all connections whereas `session_account_connect_attrs` only includes the connections for the same account as the current user. That is you can get the content of `session_account_connect_attrs` from `session_connect_attrs` using the following query:

Query 13b

```
SELECT a.*
  FROM session_connect_attrs a
       INNER JOIN threads t USING (PROCESSLIST_ID)
 WHERE     t.PROCESSLIST_USER = SUBSTRING_INDEX(USER(), '@', 1)
       AND t.PROCESSLIST_HOST = SUBSTRING_INDEX(USER(), '@', -1);
```

Threads

The `threads` table is one of the most central tables in the Performance Schema. The `THREAD_ID` is for example a "key" for all of the non-summary event tables.

This example below includes both a background thread (`THREAD_ID` = 17) and a foreground thread (`THREAD_ID` = 21).

Background threads are the ones created by MySQL to handle the internal server activity – in this case it is the master InnoDB thread.

Foreground threads are client connections where `PROCESSLIST_ID` is the same as the Id displayed by `SHOW PROCESSLIST`. The active connection's processlist id can be found using the `CONNECTION_ID()` function.

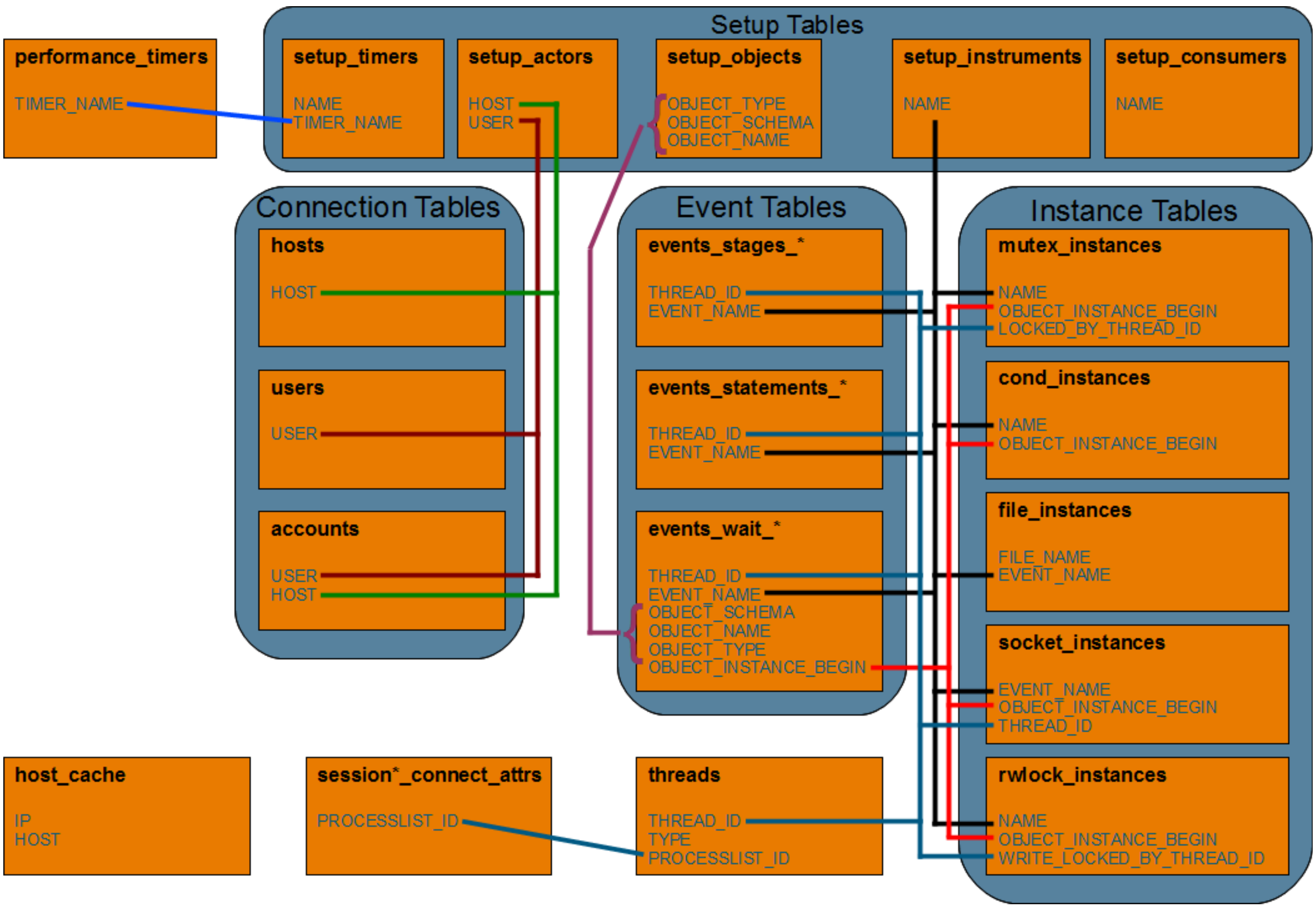
The `INSTRUMENTED` column tells whether the thread is being instrumented. This column is updatable, so for a given thread, instrumentation can be enabled and disabled on demand.

Query 14

```
mysql> SELECT * FROM threads WHERE NAME = 'thread/innodb/srv_master_thread'
OR PROCESSLIST_ID = CONNECTION_ID()\G
*************************** 1. row ***************************
          THREAD_ID: 17
               NAME: thread/innodb/srv_master_thread
               TYPE: BACKGROUND
     PROCESSLIST_ID: NULL
   PROCESSLIST_USER: NULL
   PROCESSLIST_HOST: NULL
     PROCESSLIST_DB: NULL
PROCESSLIST_COMMAND: NULL
   PROCESSLIST_TIME: NULL
  PROCESSLIST_STATE: NULL
   PROCESSLIST_INFO: NULL
   PARENT_THREAD_ID: NULL
               ROLE: NULL
       INSTRUMENTED: YES
*************************** 2. row ***************************
          THREAD_ID: 20
               NAME: thread/sql/one_connection
               TYPE: FOREGROUND
     PROCESSLIST_ID: 1
   PROCESSLIST_USER: root
   PROCESSLIST_HOST: localhost
     PROCESSLIST_DB: performance_schema
PROCESSLIST_COMMAND: Query
   PROCESSLIST_TIME: 0
  PROCESSLIST_STATE: Sending data
   PROCESSLIST_INFO: SELECT * FROM threads WHERE NAME =
'thread/innodb/srv_master_thread' OR PROCESSLIST_ID = CONNECTION_ID()
   PARENT_THREAD_ID: 1
               ROLE: NULL
       INSTRUMENTED: YES
2 rows in set (0.03 sec)
```

Overview of the Relation Between Tables

The following diagram shows how the Performance Schema tables relate to each other.



Using the Performance Schema

The following will show some examples of how the Performance Schema can be used:

- Replacing SHOW PROCESSLIST.
- Replacing the slow query log.
- Investigating a slow query.
- Investigating a general high server load.

SHOW PROCESSLIST

As the output above from the `threads` table showed, there are a number of columns where the name starts with `PROCESSLIST_`. Each of these corresponds to a field in the output of `SHOW PROCESSLIST`. So recreating the output of `SHOW PROCESSLIST` is straight forward:

Query 15a

```
mysql> SELECT PROCESSLIST_ID AS Id, PROCESSLIST_USER AS User,
    ->        PROCESSLIST_HOST AS Host, PROCESSLIST_DB AS db,
    ->        PROCESSLIST_COMMAND AS Command,PROCESSLIST_TIME AS Time,
    ->        PROCESSLIST_State AS State,
    ->        LEFT(PROCESSLIST_INFO, 100) AS Info
    ->   FROM threads t
    ->  WHERE TYPE = 'FOREGROUND'\G
*************************** 1. row ***************************
     Id: 1
   User: root
   Host: localhost
     db: performance_schema
Command: Query
   Time: 0
  State: Sending data
   Info: SELECT PROCESSLIST_ID AS Id, PROCESSLIST_USER AS User,
       PROCESSLIST_HOST AS Host, PROCESSLIST_
1 row in set (0.01 sec)
```

So why use the Performance Schema – after all it is much longer to type than just typing `SHOW PROCESSLIST`? There are some good reasons to make the change:

- `SHOW PROCESSLIST` requires several mutexes including some that affects all the connections. So if you are fetching the processlist often, it can affect performance.
- Querying the `threads` table doesn't take any locks and mutexes other than would be needed for other queries.
- The threads table also include information about background threads.
- It is possible to join on other Performance Schema tables to get additional information.
- It is possible to configure which threads are instrumented.

So lets create a view that can be used to get the process list with more information about the processes:

Query 15b

```
CREATE OR REPLACE SQL SECURITY INVOKER VIEW mysqlconnect.processlist AS
SELECT t.PROCESSLIST_ID AS Id, t.PROCESSLIST_USER AS User, t.PROCESSLIST_HOST AS Host,
       t.PROCESSLIST_DB AS db, t.PROCESSLIST_COMMAND AS Command,
       t.PROCESSLIST_TIME AS Time, ps_helper.format_time(SUM(SUM_TIMER_WAIT)) AS TotalExecTime,
       IF(t.PROCESSLIST_INFO IS NULL, '', t.PROCESSLIST_STATE) AS State,
       IF(s.TIMER_END IS NULL, 'YES', 'NO') AS IsExecuting,
       SUM(ste.COUNT_STAR) AS TotalStatements,
       s.ERRORS,                   SUM(ste.SUM_ERRORS)                  AS TotalErrors,
       s.WARNINGS,                 SUM(ste.SUM_WARNINGS)                AS TotalWarnings,
       s.ROWS_AFFECTED,            SUM(ste.SUM_ROWS_AFFECTED)           AS TotalRowsAffected,
       s.ROWS_SENT,                SUM(ste.SUM_ROWS_SENT)               AS TotalRowsSent,
       s.ROWS_EXAMINED,            SUM(ste.SUM_ROWS_EXAMINED)           AS TotalRowsExamnied,
       s.CREATED_TMP_DISK_TABLES,  SUM(ste.SUM_CREATED_TMP_DISK_TABLES) AS TotalTmpDiskTables,
       s.CREATED_TMP_TABLES,       SUM(ste.SUM_CREATED_TMP_TABLES)      AS TotalTmpTables,
       s.SELECT_FULL_JOIN,         SUM(ste.SUM_SELECT_FULL_JOIN)        AS TotalFullJoin,
       s.SELECT_FULL_RANGE_JOIN,   SUM(ste.SUM_SELECT_FULL_RANGE_JOIN)  AS TotalFullRangeJoin,
       s.SELECT_RANGE,             SUM(ste.SUM_SELECT_RANGE)            AS TotalRange,
       s.SELECT_RANGE_CHECK,       SUM(ste.SUM_SELECT_RANGE_CHECK)      AS TotalRangeCheck,
       s.SELECT_SCAN,              SUM(ste.SUM_SELECT_SCAN)             AS TotalScan,
       s.SORT_MERGE_PASSES,        SUM(ste.SUM_SORT_MERGE_PASSES)       AS TotalSortMergePasses,
       s.SORT_RANGE,               SUM(ste.SUM_SORT_RANGE)              AS TotalSortRange,
       s.SORT_ROWS,                SUM(ste.SUM_SORT_ROWS)               AS TotalSortRows,
       s.SORT_SCAN,                SUM(ste.SUM_SORT_SCAN)               AS TotalSortScan,
       s.NO_INDEX_USED,            SUM(ste.SUM_NO_INDEX_USED)           AS TotalNoIndex,
       s.NO_GOOD_INDEX_USED,       SUM(ste.SUM_NO_GOOD_INDEX_USED)      AS TotalNoGoodIndex,
       LEFT(s.SQL_TEXT, 100) AS Info
  FROM performance_schema.threads t
       INNER JOIN performance_schema.events_statements_current s USING (THREAD_ID)
       INNER JOIN performance_schema.events_statements_summary_by_thread_by_event_name ste
                  USING (THREAD_ID)
 WHERE t.TYPE = 'FOREGROUND'
 GROUP BY THREAD_ID;
```

Using the new `mysqlconnect.processlist` view gives:

Query 15c

```
mysql> SELECT * FROM mysqlconnect.processlist\G
*************************** 1. row ***************************
                    Id: 1
                  User: root
                  Host: localhost
                    db: performance_schema
               Command: Query
                  Time: 0
         TotalExecTime: 51.81 ms
                 State: Sending data
           IsExecuting: YES
       TotalStatements: 55
                ERRORS: 0
           TotalErrors: 0
              WARNINGS: 0
         TotalWarnings: 0
         ROWS_AFFECTED: 0
     TotalRowsAffected: 0
             ROWS_SENT: 0
         TotalRowsSent: 68
         ROWS_EXAMINED: 0
     TotalRowsExamined: 68
CREATED_TMP_DISK_TABLES: 0
     TotalTmpDiskTables: 0
     CREATED_TMP_TABLES: 2
         TotalTmpTables: 2
       SELECT_FULL_JOIN: 2
          TotalFullJoin: 0
 SELECT_FULL_RANGE_JOIN: 0
     TotalFullRangeJoin: 0
           SELECT_RANGE: 0
             TotalRange: 0
     SELECT_RANGE_CHECK: 0
        TotalRangeCheck: 0
            SELECT_SCAN: 2
              TotalScan: 3
      SORT_MERGE_PASSES: 0
   TotalSortMergePasses: 0
             SORT_RANGE: 0
         TotalSortRange: 0
              SORT_ROWS: 0
          TotalSortRows: 0
              SORT_SCAN: 0
          TotalSortScan: 0
          NO_INDEX_USED: 1
           TotalNoIndex: 3
     NO_GOOD_INDEX_USED: 0
       TotalNoGoodIndex: 0
                  Info: SELECT * FROM mysqlconnect.processlist
1 row in set (0.15 sec)
```

Some notes about the view:

- The `events_statements_*` tables are used to get more information about the connections and queries.
- There will be a statement for each connected thread in `events_statements_current` irrespectively of whether the connection is currently executing a query or sleeping. If the connection is sleeping, the last completed query is listed.
- The `events_statements_summary_by_thread_by_event_name` is used to get historical data for the connections. Note that only statistics for completed queries are included in this table, so the `Total*` columns does not include a currently executing query.
- The `IsExecuting` column has been added to tell whether the query is currently executing. As the `TIMER_END` column in events_statements_current is populated when a query finished execution, so whether `TIMER_END` is `NULL` can be used for this purpose.
- To display the total execution time of completed queries in a human readable format, the `format_time()` function from `ps_helper` (http://www.markleith.co.uk/ps_helper/) is used.
- We use `PROCESSLIST_TIME` instead of `TIMER_WAIT` even though the latter is in picoseconds (trillionths of a second - or 10^{-12} seconds) as `TIMER_WAIT` is not filled in until the statement has finished executing. So `TIMER_WAIT` is only useful for completed statements, and the function to obtain the current time is not exposed at the SQL layer.

The Slow Query Log

Continuing from the process list, we can also use the Performance Schema to recreate the slow query log. Take the example:

Query 16a

```
mysql> SET SESSION long_query_time = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT SLEEP(2);
+----------+
| SLEEP(2) |
+----------+
|        0 |
+----------+
1 row in set (2.01 sec)
```

The above gives an entry in the slow query log like:

```
# Time: 120923 13:48:40
# User@Host: root[root] @ localhost []  Id:    14
# Query_time: 2.031227  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
use performance_schema;
SET timestamp=1348372120;
SELECT SLEEP(2);
```

To recreate this entry using the Performance Schema, you can use a query like:

Query 16b

```
mysql> SET @SERVER_START = NOW() - INTERVAL (SELECT VARIABLE_VALUE FROM
information_schema.GLOBAL_STATUS WHERE VARIABLE_NAME = 'Uptime') SECOND;

mysql> SELECT CONCAT('# Time: ', DATE_FORMAT(@SERVER_START + INTERVAL
(TIMER_END/1000000000000) SECOND, '%y%m%d %H:%i:%s'),'
# User@Host: ps @ localhost []  Id: ', THREAD_ID, '
# Query_time: ', ROUND(TIMER_WAIT/1000000000000, 6), '  Lock_time: ',
ROUND(LOCK_TIME/1000000000000, 6), ' Rows_sent: ', ROWS_SENT, '
Rows_examined: ', ROWS_EXAMINED, '
use ', CURRENT_SCHEMA, ';
SET timestamp=', IFNULL(ROUND(UNIX_TIMESTAMP(@SERVER_START + INTERVAL
(TIMER_END/1000000000000) SECOND), 0), 'NULL'), ';
', SQL_TEXT, ';') AS 'SlowQueryLogEvent'
  FROM `performance_schema`.`events_statements_history_long`
 WHERE TIMER_WAIT > @@session.long_query_time*1000000000000
       AND SQL_TEXT IS NOT NULL
 ORDER BY TIMER_END DESC
 LIMIT 1;
+------------------------------------------------------------------------+
| SlowQueryLogEvent                                                      |
+------------------------------------------------------------------------+
| # Time: 120927 01:52:05
# User@Host: ps @ localhost []  Id: 20
# Query_time: 2.025005  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
use performance_schema;
SET timestamp=1348674726;
SELECT SLEEP(2); |
+------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Or use the `slow_query_log procedure` in the `ps_tools` database. This procedure takes two arguments: the threshold for including queries (equivalent to `long_query_time`) and the earliest time to consider (use `NULL` to include all history):

```
shell# mysql -BN --raw -e "CALL ps_tools.slow_query_log(1, NULL)"
/usr/sbin/mysqld, Version: 5.6.6-m9-log (MySQL Community Server (GPL)).
started with:
Tcp port: 3306  Unix socket: /usr/lib/mysql/mysql.sock
Time                 Id Command    Argument
# Time: 120928 16:40:21
# User@Host: root @ localhost []  Id: 2
# Query_time: 2.013918  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
use performance_schema;
SET timestamp=1348814421;
SELECT SLEEP(2);
```

Notes about the query:

- The user has been set to ps@localhost as the actual account is only known while the connection is still connected. We could try to see whether the `THREAD_ID` still exists in the threads table and use the information if available.
- Likewise the process list id is not in general known, so here it has been replaced by the thread id.
- The time of the query will only be approximate as the `TIMER_START` and `TIMER_END` values may drift compared to the server start. See also https://dev.mysql.com/doc/refman/5.6/en/performance-schema-timing.html
- You can also add more information about the query such as number of internal temporary tables, however note that the `mysqldumpslow` script does not handle extra comments line.
- It would make sense to create a stored procedure which can be called from the command line if you intend to use this query.

Instead of recreating the slow query log entries and pass it through `mysqldumpslow`, we can take it a step further and do something similar directly from inside MySQL. For this we will use the `statements_with_runtimes_in_95th_percentile` view in `ps_helper`.

The view uses the `events_statements_summary_by_digest` table to find the most expensive queries based on run time.

Query 16c

```
mysql> SELECT * from ps_helper.statements_with_runtimes_in_95th_percentile\G
*************************** 1. row ***************************
        query: SELECT SLEEP (?)
    full_scan:
   exec_count: 1
    err_count: 0
   warn_count: 0
total_latency: 2.03 s
  max_latency: 2.03 s
  avg_latency: 2.03 s
    rows_sent: 1
rows_sent_avg: 1
 rows_scanned: 0
       digest: ea1906ef8f864f6dfa5a2c8a3f25477c
*************************** 2. row ***************************
        query: SELECT * FROM mysqlconnect . processlist
    full_scan: *
   exec_count: 1
    err_count: 0
   warn_count: 0
total_latency: 152.39 ms
  max_latency: 152.39 ms
  avg_latency: 152.39 ms
    rows_sent: 1
rows_sent_avg: 1
 rows_scanned: 22
       digest: 799cec2d5c9cbca2ca1ff3fae3b698de
2 rows in set (0.05 sec)
```

If you look at the first query (`SELECT SLEEP (?)`) you will notice that the argument to `SLEEP()` has been replaced with a question mark. The `events_statements_summary_by_digest` table is grouping by the query digest rather than the actual query. The digest is based on a normalised version of the query which allows similar queries to be considered the same. Using the digest is in general more useful than specific for determining the types of queries that are slow. The `mysqldumpslow` script does a similar normalisation when aggregating the slow query log.

To see how this work, let us look at an example:

Query 16d

```
mysql> TRUNCATE events_statements_history;
Query OK, 0 rows affected (0.00 sec)

mysql> TRUNCATE events_statements_summary_by_digest;
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT Code, Name FROM world.Country WHERE Code = 'AUS';
+------+-----------+
| Code | Name      |
+------+-----------+
| AUS  | Australia |
+------+-----------+
1 row in set (0.00 sec)

mysql> SELECT Code, Name FROM world.Country WHERE Code = 'USA';
+------+---------------+
| Code | Name          |
+------+---------------+
| USA  | United States |
+------+---------------+
1 row in set (0.00 sec)

mysql> SELECT DIGEST, DIGEST_TEXT, SQL_TEXT FROM events_statements_history\G
*************************** 1. row ***************************
     DIGEST: 0a918f84e9308d683a431f41d2dada4c
DIGEST_TEXT: TRUNCATE `events_statements_history`
   SQL_TEXT: TRUNCATE events_statements_history
*************************** 2. row ***************************
     DIGEST: 6b90709ef8b40f1aead03e5a7a3e2cf9
DIGEST_TEXT: TRUNCATE `events_statements_summary_by_digest`
   SQL_TEXT: TRUNCATE events_statements_summary_by_digest
*************************** 3. row ***************************
     DIGEST: dedcd88c723e7b04e925975f78b8ae73
DIGEST_TEXT: SELECT CODE , NAME FROM `world` . `Country` WHERE CODE = ?
   SQL_TEXT: SELECT Code, Name FROM world.Country WHERE Code = 'AUS'
*************************** 4. row ***************************
     DIGEST: dedcd88c723e7b04e925975f78b8ae73
DIGEST_TEXT: SELECT CODE , NAME FROM `world` . `Country` WHERE CODE = ?
   SQL_TEXT: SELECT Code, Name FROM world.Country WHERE Code = 'USA'
4 rows in set (0.14 sec)
```

The events_statements_history table has a row for each of the three queries, but the `DIGEST` and `DIGEST_TEXT` is the same for the two queries against the `world.Country` table.

DIGEST_TEXT is the normalised query and DIGEST is the md5 sum of the normalised query; both can be found in all of the events_statements_current, events_statements_history, and events_statements_history_long tables, and the DIGEST is used for summarising the statements in the events_statements_summary_by_digest table as used in the statements_with_runtimes_in_95th_percentile view.

Query 16e

```
mysql> SELECT DIGEST, DIGEST_TEXT, COUNT_STAR
    ->   FROM events_statements_summary_by_digest
    ->   WHERE DIGEST_TEXT LIKE '%world%'\G
*************************** 1. row ***************************
     DIGEST: dedcd88c723e7b04e925975f78b8ae73
DIGEST_TEXT: SELECT CODE , NAME FROM `world` . `Country` WHERE CODE = ?
 COUNT_STAR: 2
1 row in set (0.11 sec)
```

Investigating a Slow Query

Let us run a slow query and see what kind of information is available from the Performance Schema.

First find the THREAD_ID of the thread running the query and truncate the events_statements_history table.

Query 17a

```
mysql> SELECT THREAD_ID FROM threads WHERE PROCESSLIST_ID = CONNECTION_ID();
+-----------+
| THREAD_ID |
+-----------+
|        23 |
+-----------+
1 row in set (0.00 sec)

mysql> use sakila
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> TRUNCATE performance_schema.events_statements_history;
Query OK, 0 rows affected (0.01 sec)
```

The query we will be investigating is:

Query 17b

```
mysql> SELECT FID, title, LEFT(actors, 20) FROM nicer_but_slower_film_list;
+------+------------------------------+----------------------+
| FID  | title                        | LEFT(actors, 20)     |
+------+------------------------------+----------------------+
|    1 | ACADEMY DINOSAUR             | Penelope Guiness, Ch |
…
| 1000 | ZORRO ARK                    | Ian Tandy, Nick Dege |
+------+------------------------------+----------------------+
997 rows in set (0.41 sec)
```

We can now start the investigation by looking at the query in the `events_statements_history` table:

Query 17c

```
mysql> SELECT ps_helper.format_time(TIMER_WAIT) AS TIMER_WAIT,
    ->        ROWS_SENT, ROWS_EXAMINED,
    ->        CREATED_TMP_DISK_TABLES AS TMP_DISK_TABLES,
    ->        CREATED_TMP_TABLES AS TMP_TABLES, SELECT_SCAN
    ->   FROM performance_schema.events_statements_history
    ->  WHERE THREAD_ID = 23 AND ROWS_SENT > 100
    ->  ORDER BY TIMER_START;
+------------+-----------+---------------+-----------------+------------+-------------+
| TIMER_WAIT | ROWS_SENT | ROWS_EXAMINED | TMP_DISK_TABLES | TMP_TABLES | SELECT_SCAN |
+------------+-----------+---------------+-----------------+------------+-------------+
| 412.02 ms  |       997 |         24861 |               2 |          3 |           2 |
+------------+-----------+---------------+-----------------+------------+-------------+
1 row in set (0.02 sec)
```

We can already see a few potential problems:

- A seemingly simple query is creating 3 internal temporary tables of which 2 are created on disk.
- It does 2 `SELECT_SCAN`s, so furthermore a clear indication that it is not a simple table (but rather a view).
- 24861 rows are examined to return 997 rows – can we improve that?

Using `EXPLAIN` on the query gives:

Query 17d

```
mysql> EXPLAIN SELECT FID, title, LEFT(actors, 20) FROM nicer_but_slower_film_list;
+----+-------------+---------------+--------+...+------+----------------------------------+
| id | select_type | table         | type   |...| rows | Extra                            |
+----+-------------+---------------+--------+...+------+----------------------------------+
|  1 | PRIMARY     | <derived2>    | ALL    |...|  992 | NULL                             |
|  2 | DERIVED     | category      | ALL    |...|   16 | Using temporary; Using filesort  |
|  2 | DERIVED     | film_category | ref    |...|   31 | Using where; Using index         |
|  2 | DERIVED     | film          | eq_ref |...|    1 | NULL                             |
|  2 | DERIVED     | film_actor    | ref    |...|    2 | Using index                      |
|  2 | DERIVED     | actor         | eq_ref |...|    1 | NULL                             |
+----+-------------+---------------+--------+...+------+----------------------------------+
6 rows in set (0.06 sec)
```

So indeed it is a view using five tables, and furthermore it's materialising the view as a temporary table. The view definition is:

Query 17e

```
CREATE VIEW nicer_but_slower_film_list
AS
SELECT film.film_id AS FID, film.title AS title, film.description AS description,
       category.name AS category, film.rental_rate AS price, film.length AS length,
       film.rating AS rating, GROUP_CONCAT(CONCAT(CONCAT(UCASE(SUBSTR(actor.first_name,1,1)),
       LCASE(SUBSTR(actor.first_name,2,LENGTH(actor.first_name))),_utf8' ',
       CONCAT(UCASE(SUBSTR(actor.last_name,1,1)),
       LCASE(SUBSTR(actor.last_name,2,LENGTH(actor.last_name)))))) SEPARATOR ', ') AS actors
  FROM category
       LEFT JOIN film_category ON category.category_id = film_category.category_id
       LEFT JOIN film ON film_category.film_id = film.film_id
       JOIN film_actor ON film.film_id = film_actor.film_id
       JOIN actor ON film_actor.actor_id = actor.actor_id
 GROUP BY film.film_id;
```

The three internal temporary tables come from:

- The `GROUP BY`
- Materialisation of the view
- The `GROUP_CONCAT`

To address this, we can try rewriting the query making the following changes:

- Convert the LEFT JOINs to INNER JOINs (as all the films have a category).
- Force the film table to be the first table through a STRAIGHT_JOIN to ensure the index on film.film_id can be used for the GROUP BY.

Query 17f

```
CREATE OR REPLACE VIEW nicer_but_slower_film_list2
AS
 SELECT film.film_id AS FID, film.title AS title, film.description AS description,
        category.name AS category, film.rental_rate AS price, film.length AS length,
        film.rating AS rating, GROUP_CONCAT(CONCAT(CONCAT(UCASE(SUBSTR(actor.first_name,1,1)),
        LCASE(SUBSTR(actor.first_name,2,LENGTH(actor.first_name))),_utf8' ',
        CONCAT(UCASE(SUBSTR(actor.last_name,1,1)),
        LCASE(SUBSTR(actor.last_name,2,LENGTH(actor.last_name)))))) SEPARATOR ', ') AS actors
   FROM film
        STRAIGHT_JOIN film_category ON film_category.film_id = film.film_id
        JOIN category ON category.category_id = film_category.category_id
        JOIN film_actor ON film.film_id = film_actor.film_id
        JOIN actor ON film_actor.actor_id = actor.actor_id
  GROUP BY film.film_id;
```

Running the query again and checking `events_statements_history` now gives:

Query 17g

```
mysql> SELECT ps_helper.format_time(TIMER_WAIT) AS TIMER_WAIT,
    ->        ROWS_SENT, ROWS_EXAMINED,
    ->        CREATED_TMP_DISK_TABLES AS TMP_DISK_TABLES,
    ->        CREATED_TMP_TABLES AS TMP_TABLES, SELECT_SCAN
    ->   FROM performance_schema.events_statements_history
    ->  WHERE THREAD_ID = 23 AND ROWS_SENT > 100
    ->  ORDER BY TIMER_START;
+------------+-----------+---------------+-----------------+------------+-------------+
| TIMER_WAIT | ROWS_SENT | ROWS_EXAMINED | TMP_DISK_TABLES | TMP_TABLES | SELECT_SCAN |
+------------+-----------+---------------+-----------------+------------+-------------+
| 412.02 ms  |       997 |         24861 |               2 |          3 |           2 |
| 261.86 ms  |       997 |         14921 |               1 |          2 |           2 |
+------------+-----------+---------------+-----------------+------------+-------------+
2 rows in set (0.02 sec)
```

So somewhat of an improvement. But what if we avoid the overhead of having to materialise the view in a temporary table? Lets try to run the SELECT from nicer_but_slower_film_list directly:

Query 17h

```
mysql> SELECT ps_helper.format_time(TIMER_WAIT) AS TIMER_WAIT,
    ->        ROWS_SENT, ROWS_EXAMINED,
    ->        CREATED_TMP_DISK_TABLES AS TMP_DISK_TABLES,
    ->        CREATED_TMP_TABLES AS TMP_TABLES, SELECT_SCAN
    ->   FROM performance_schema.events_statements_history
    ->  WHERE THREAD_ID = 23 AND ROWS_SENT > 100
    ->  ORDER BY TIMER_START;
+------------+-----------+---------------+-----------------+------------+-------------+
| TIMER_WAIT | ROWS_SENT | ROWS_EXAMINED | TMP_DISK_TABLES | TMP_TABLES | SELECT_SCAN |
+------------+-----------+---------------+-----------------+------------+-------------+
| 412.02 ms  |       997 |         24861 |               2 |          3 |           2 |
| 261.86 ms  |       997 |         14921 |               1 |          2 |           2 |
| 186.43 ms  |       997 |         13924 |               0 |          1 |           1 |
+------------+-----------+---------------+-----------------+------------+-------------+
2 rows in set (0.02 sec)
```

Investigating General Server Load

First we will make some changes to the MySQL configuration.

1. Run some queries to generate load

```
shell# bash /tmp/hol/run_queries.sh
```

2. Start the investigation

Query 18

```
mysql> use performance_schema
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT EVENT_NAME, COUNT_STAR,
    ->        ps_helper.format_time(SUM_TIMER_WAIT) AS SUM_TIMER_WAIT
    ->   FROM events_waits_summary_global_by_event_name s
    ->  ORDER BY s.SUM_TIMER_WAIT DESC
    ->  LIMIT 10;
+-------------------------------------------+------------+----------------+
| EVENT_NAME                                | COUNT_STAR | SUM_TIMER_WAIT |
+-------------------------------------------+------------+----------------+
| idle                                      |     228840 | 00:19:22.6267  |
| wait/synch/mutex/innodb/buf_pool_mutex    |     780910 | 00:07:40.5441  |
| wait/io/socket/sql/client_connection      |     458138 | 00:07:14.7444  |
| wait/io/table/sql/handler                 |    1171278 | 00:05:13.2516  |
| wait/io/file/innodb/innodb_data_file      |     120084 | 00:01:02.7647  |
| wait/synch/mutex/innodb/lock_mutex        |    2763636 | 00:01:02.3667  |
| wait/io/file/innodb/innodb_log_file       |       9855 | 46.82 s        |
| wait/synch/rwlock/innodb/hash table locks |    4928397 | 22.11 s        |
| wait/synch/mutex/mysys/THR_LOCK::mutex    |    1700455 | 16.95 s        |
| wait/synch/cond/sql/BINARY_LOG::COND_done |       1185 | 13.33 s        |
+-------------------------------------------+------------+----------------+
10 rows in set (0.01 sec)
```

Of the events spending most time, some of them we are not interested in here:

- `idle`: that means something has been doing anything – so it is not putting load on the server, so not a problem.
- `wait/io/socket/sql/client_connection`: this is related to creating the connections.
- `wait/io/table/sql/handler`: Table I/O events are so called *molecular* events, i.e. they include other events.

That leaves the following events as the biggest:

- `wait/synch/mutex/buf_pool_mutex`
- `wait/io/file/innodb/innodb_data_file`
- `wait/synch/mutex/innodb/lock_space`
- `wait/io/file/innodb/innodb_log_file`

The wait/synch/mutex/buf_pool_mutex, wait/io/file/innodb/innodb_data_file, and wait/io/file/innodb/innodb_log_file are signs of the InnoDB log files and possibly the buffer pool are too small. Too small log settings cause excessive flushing from the log files and buffer pool, and when the circular redo log is about to be overwritten, a checkpoint is forced (uses the data file) so that recovery will not break.

So the first step here would be to increase the size of the InnoDB log files and the InnoDB buffer pool. Then rerun the test see what the effect is.

Slave Load

One of the important things in a master-slave setup is to monitor the slave to be sure that the slave keeps up with the master. Traditionally Seconds_Behind_Master from SHOW SLAVE STATUS has been used for this, but it has its problems, for example it is reactive monitoring.

The Performance Schema allows to proactively monitoring the slave.

First log into the slave:

```
shell# mysql --socket=/var/lib/mysql_slave/mysql.sock
```

ps_tools has a stored procedure – compute_slave_load_average – for this:

Query 19a

```
mysql> use ps_tools;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> CALL compute_slave_load_average();
Query OK, 0 rows affected (0.12 sec)
```

The procedure updates the `ps_tools.slave_sql_load_average` table with the slave load statistics:

Query 19b

```
mysql> SHOW CREATE TABLE slave_sql_load_average\G
*************************** 1. row ***************************
       Table: slave_sql_load_average
Create Table: CREATE TABLE `slave_sql_load_average` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `tstamp` timestamp(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6) ON UPDATE CURRENT_TIMESTAMP(6),
  `idle_avg` varchar(12) DEFAULT NULL,
  `idle_delta_formatted` varchar(12) DEFAULT NULL,
  `busy_pct` decimal(5,2) DEFAULT NULL,
  `one_min_avg` decimal(5,2) DEFAULT NULL,
  `five_min_avg` decimal(5,2) DEFAULT NULL,
  `fifteen_min_avg` decimal(5,2) DEFAULT NULL,
  `idle_sum` bigint(20) DEFAULT NULL,
  `idle_delta` bigint(20) DEFAULT NULL,
  `events_sum` int(11) DEFAULT NULL,
  `events_delta` int(11) DEFAULT NULL,
  `current_wait` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `tstamp` (`tstamp`)
) ENGINE=InnoDB AUTO_INCREMENT=1407 DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

In ps_tools an event is run every five seconds calling `compute_slave_load_average`, but it could also be done having an external process, such as a monitoring system, run the equivalent queries.

The way it works is to check how long time the SQL thread () is spending in the wait/synch/cond/sql/RELAY_LOG::update_cond event out of the whole interval between calling `compute_slave_load_average`. The need to work with deltas benefits from the support for microseconds in timestamps in MySQL 5.6.

To see how the slaved performed during the previous tests:

Query 19c

```
mysql> SELECT id, DATE_FORMAT(tstamp, '%H:%i:%s') AS tstamp,
    ->        idle_delta_formatted AS idle, busy_pct,
    ->        one_min_avg AS avg_1, five_min_avg AS avg_5,
    ->        fifteen_min_avg as avg_15
    ->   FROM slave_sql_load_average s
    ->  ORDER BY s.tstamp;
+------+----------+-----------+----------+-------+-------+--------+
| id   | tstamp   | idle      | busy_pct | avg_1 | avg_5 | avg_15 |
+------+----------+-----------+----------+-------+-------+--------+
…
| 1740 | 08:58:36 | 3.64 s    |    24.38 | 33.29 |  7.10 |   2.41 |
| 1741 | 08:58:41 | 2.80 s    |    42.65 | 33.99 |  7.80 |   2.64 |
| 1742 | 08:58:46 | 1.91 s    |    60.36 | 36.98 |  8.79 |   2.98 |
| 1743 | 08:58:51 | 3.25 s    |    33.61 | 36.72 |  9.34 |   3.17 |
| 1744 | 08:58:56 | 2.83 s    |    42.69 | 37.08 | 10.04 |   3.40 |
| 1745 | 08:59:01 | 2.88 s    |    39.41 | 37.26 | 10.69 |   3.62 |
| 1746 | 08:59:06 | 663.44 ms |    86.44 | 42.26 | 12.11 |   4.10 |
| 1747 | 08:59:11 | 4.99 s    |     0.00 | 39.36 | 12.11 |   4.10 |
| 1748 | 08:59:16 | 5.00 s    |     0.00 | 36.33 | 12.11 |   4.10 |
…
```

To use this optimally plot it, so the busy % and averages can be see over time.

Notes:

- This feature is actually also available in MySQL 5.5 (requires a few changes in `compute_slave_load_average` and `slave_sql_load_average`.
- The load averages are simple averages of busy %, so it is not the same as the load average on Linux.
- If you want to read more about the slave SQL load average, see also http://www.markleith.co.uk/2012/07/24/a-mysql-replication-load-average-with-performance-schema/. This page also includes the MySQL 5.5 version of the procedure and table.

Stack Trace

If the `events_stages_history_long`, `events_statements_history_long`, and `events_waits_history_long` consumers are all enabled, it is possible to create a stack trace. The `ps_helper` procedure `dump_thread_stack` does this.

Query 20

```
mysql> use mysqlconnect
Database changed
mysql> CREATE TABLE t3 (id int unsigned PRIMARY KEY);
Query OK, 0 rows affected (0.38 sec)

mysql> connect
Connection id:    9
Current database: mysqlconnect

mysql> SELECT THREAD_ID FROM performance_schema.threads WHERE PROCESSLIST_ID = CONNECTION_ID();
+-----------+
| THREAD_ID |
+-----------+
|        28 |
+-----------+
1 row in set (0.00 sec)

mysql> INSERT INTO t3 VALUES (1);
Query OK, 1 row affected (0.03 sec)
```

Run the `dump_thread_stack` from the command line (to be able to get the raw output):

```
shell# mysql -BN -e "CALL ps_helper.dump_thread_stack(28, TRUE)" > /tmp/stack.dot
shell# dot -Tpdf /tmp/stack.dot -o /tmp/stack.pdf
shell# evince /tmp/stack.pdf
```