

MySQL Performance Schema

Instrumenting code

Marc Alff
Sr Software Engineer
Oracle Corporation
marc.alf@sun.com



*These slides released under the Creative Commons
Attribution-Noncommercial-Share Alike License*



Legal Disclaimer and disclosures

- I (Marc Alff) am a former Oracle, MySQL and Sun Microsystem employee,
- I am an Oracle employee. I am not an Oracle authorized representative, views expressed here are my own opinions,
- The “MySQL Performance Schema” feature is not part of any GA release of the MySQL server,
- This feature may be changed at any time, for any reason, without notice,
- This feature may or may not be available in future releases,
- **The content of this presentation is only based on material made previously publicly available by Sun Microsystems, Inc,**
- **Customers may not assume this feature will be available in future releases, when making purchasing decisions.**



Agenda

- Introduction
- Instrumenting code
- Functionality gained
- Cost / Benefit
- Q & A



Thoughts ...

- The performance schema is to users what “open source” is to developers.
- “Given enough [instrumentation] eyeballs, all [performance] bugs are shallow”.
- Adding performance schema instrumentation will benefit all parties in the MySQL ecosystem: Code maintainers (Oracle, community, third party), support, consultants, DBAs, production engineers, application engineers, final users, tool vendors.
- It is in the best benefit of all parties to have performance instrumentation.
- Having instrumentation is a competitive advantage.



Scope of this presentation

- Only a subset of the Performance Schema
- Covers how to instrument *code*:
 - Only applies to instrumentation of *code* artifacts.
 - Does not cover *other* instrumentation types.
- Target audience are server / plug-in developers
 - Assumes basic MySQL development knowledge.
 - Other presentations will cover the performance schema usage.



Prerequisites

- MySQL 5.5.99-m3 (Celosia)

lp: mysql-next-mr

<https://code.launchpad.net/~mysql/mysql-server/mysql-next-mr>

- `./configure -with-perfschema`
 - Compiled in by default for “max” builds.
- `./mysqld --performance-schema`
 - Enabled by default in the MTR test suite

- MySQL Manual

<http://dev.mysql.com/doc/performance-schema/en/index.html>

- Other resources

<http://marcalff.blogspot.com/>



Instrumenting code



Instrumentation basics

- Non intrusive
 - Only minor code changes are required
- Can be disabled
 - At compile time
 - At runtime
- Optional
 - Mixing instrumented / non instrumented code in the same binary is ok (for integration with third party code)
- Highly recommended
 - Instrument everything: simpler code base, simpler maintenance, better functionality.



Naming of instruments

- `mysql> select * from SETUP_INSTRUMENTS;`
- Concept of name space
 - “sql” for the server
 - “mysys” for mysys
 - “xxx” for plugin xxx
- Example of names
 - `wait/synch/mutex/sql/LOCK_open`
 - `wait/synch/cond/sql/COND_refresh`
 - `wait/synch/rwlock/sql/LOCK_grant`
 - `wait/io/file/sql/binlog`



Definition of instruments

```
PSI_mutex_key key_LOCK_open;  
static PSI_mutex_info all_server_mutexes[]=  
{  
    { &key_LOCK_open, "LOCK_open", PSI_FLAG_GLOBAL},  
    ...  
};
```

Keys are always global.

Here, key_LOCK_open is bound to a name, “LOCK_open”.

PSI_FLAG_GLOBAL means this instrumented mutex is a singleton.



Registration of instruments

```
void init_server_psi_keys(void)
{
    PSI_server->register_mutex("sql", all_server_mutexes,
array_elements(all_server_mutexes));
}
```

Register all instruments early, at server startup or plugin init.

Instruments must be visible in SETUP_INSTRUMENTS for the DBA to turn flags on/off, delayed "lazy" registration is a bug.



Instrumenting code

- Instrument a mutex:
 - `pthread_mutex_t` --> `mysql_mutex_t`
- When creating a mutex, use the instrument key:
 - `mysql_mutex_init(key_LOCK_open, &LOCK_open, MY_MUTEX_INIT_FAST);`
- Every other API uses the same arguments as the non instrumented one:
 - `pthread_mutex_lock()` → `mysql_mutex_lock()`
 - `pthread_mutex_unlock()` → `mysql_mutex_unlock()`
 - `pthread_mutex_destroy()` → `mysql_mutex_destroy()`
 - ...



Instrumenting code - example (before)

```
void do_something()  
{  
    pthread_mutex_t m;  
    pthread_mutex_init(& m, ...);  
    pthread_mutex_lock(& m, ...);  
    pthread_mutex_unlock(& m, ...);  
    pthread_mutex_destroy(& m, ...);  
}
```



Instrumenting code - example (after)

```
extern PSI_mutex_key key_mutex_m;  
void do_something()  
{  
    mysql_mutex_t m;  
    mysql_mutex_init(key_mutex_m, & m, ...);  
    mysql_mutex_lock(& m, ...);  
    mysql_mutex_unlock(& m, ...);  
    mysql_mutex_destroy(& m, ...);  
}
```



What if I have my own mutex implementation ?

- Q: My code does not use 'pthread_mutex_t', can I still add performance instrumentation ?
- A: YES
- `mysql_mutex_t` / `mysql_mutex_lock` / etc are *helpers* that expands at compile time to instrumented code.
- The *real instrumentation API* is exposed at a lower level: `PSI_server->start_mutex_wait()`, `end_mutex_wait()`, ...
- Look at `include/mysql/psi/mysql_mutex.h`
- Implement similar helpers for your implementation:
 - `xyz_mutex_t`, `xyz_mutex_init()`, `xyz_mutex_lock()`, ...



General wait/synch instrumentation

- Mutexes:
 - PSI_mutex_key, register_mutex(), mysql_mutex_t, mysql_mutex_init(), etc
- Conditions:
 - PSI_cond_key, register_condition(), mysql_cond_t, mysql_cond_init(), etc
- Read-Write locks:
 - PSI_rwlock_key, register_rwlock(), mysql_rwlock_t, mysql_rwlock_init(), etc
- They all work the same, very consistent API set.



File IO instrumentation

- File IO:
 - `PSI_file_key`, `register_file()`
- For APIs using a file descriptor:
 - `my_open` → `mysql_file_open(key_file_xxx, ...)`,
 - Usage of “int fd” is unchanged.
- For APIs using a file stream:
 - `my_fopen` → `mysql_file_fopen(key_file_xxx, ...)`,
 - `FILE` → `MYSQL_FILE`



Thread instrumentation

- Thread:
 - `PSI_thread_key`, `register_thread()`
- To instrument a thread:
 - `pthread_create` → `mysql_thread_create(key_thread_xxx, ...)`
- A thread which is not instrumented (`pthread_create`) will not generate any event.
- Instrument background threads to add visibility to a storage engine internal implementation.



Functionality gained



Effects of the instrumentation (1/2)

- TABLE SETUP_INSTRUMENTS
 - Shows every instrument key
“wait/synch/mutex/sql/LOCK_open”, etc
- TABLE EVENTS_WAITS_CURRENT, _HISTORY, _HISTORY_LONG
 - Shows waits in the code
- TABLE EVENTS_WAITS_SUMMARY_BY_EVENT_NAME, _BY_INSTANCES, _BY_THREAD_BY_EVENT_NAME
 - Show various aggregations



Effects of the instrumentation (2/2)

- TABLE MUTEX_INSTANCES
 - Shows the mutex (or mutexes when not a singleton)
 - Shows the locking thread !
- TABLE COND_INSTANCES
- TABLE RWLOCK_INSTANCES
- TABLE FILE_INSTANCES
- TABLE FILE_SUMMARY_BY_EVENT_NAME
- TABLE FILE_SUMMARY_BY_INSTANCE
- TABLE PROCESSLIST
 - Shows every thread, including background.
- For more, refer to the manual ...



Costs / Benefits



Effort in development

- A few lines of boilerplate code, in a plugin init function.
- For each instrumented mutex:
 - Manual change:
 - LINE 1: `PSI_mutex_key key_foo;`
 - LINE 2: `{ &key_foo, "foo", ...},`
 - LINE 3: `mysql_mutex_init(key_foo, ...)`
 - Search and replace `pthread_mutex` → `mysql_mutex`
- **3 lines of code per mutex !**
- Likewise for cond, rwlock, file, thread.
- The API set is very easy to learn, instrumenting code is very mechanical.



Benefits

- All the current performance schema functionality,
- All the future performance schema functionality,
 - When new aggregations are implemented, new `SUMMARY_BY_XYZ` tables will show new data for free, for code already instrumented.
- Better tooling for everybody,
- Better bug reports from users,
- Better support to users,
- Better performance tuning from experts,
- General performance schema documentation, support, ...



Conclusion ...

- With a few lines of code change, performance instrumentation can be added, which enables new functionality.
- The main burden of documentation, training, support, bug fixes, for the performance schema itself is assumed by the performance schema implementation (i.e., the server)
- A storage engine maintainer has everything to gain and nothing to loose by adding instrumentation.



Questions & Answers



Thank you