PERCONA

MySQL Logs

Vol. 1

By Bill Karwin, Peter Boros,
Aleksandr Kuzminsky and Peter Zaitsev

www.percona.com

Table of Contents

Get me some query logs!

By Bill Karwin

One of my favorite tools in the Percona Toolkit is pt-query-digest. This tool is indispensable for identifying your top SQL queries, and analyzing which queries are accounting for your database load.

But the report you get from pt-query-digest is only as good as the log of queries you give it as input. You need a large enough sample of query logs, collected over a period of time when you have representative traffic on your database.

You also need the log to include *all* the queries, not just those that take more than N seconds. The reason is that some queries are individually quick, and would not be logged if you set the long_query_time configuration variable to 1 or more seconds. You want that threshold to be 0 seconds while you're collecting logs.

However, activating such high-volume query log collection can be costly. Every statement executed on your database will cause file I/O, even when the query itself was served out of your buffer pool memory. That's a lot of overhead, so we need to be careful about how and when we collect logs, and for how long we leave that running.

I've put together a simple shell script to help automate this. I have given it the functional but unimaginative name **full-slow-log**.

The script configures full logging, then sleeps for a number of seconds to allow queries to be collected in the logs. After it finishes sleeping, or if you interrupt the script, the script restores log configuration back to the values they started with.

```
$ full-slow-log [ -v ] [ -s seconds ] [ -c config ]
```

- **-v** is for verbose output.
- **-s seconds** allows you to specify the number of seconds to sleep. The default is 5 seconds, which is probably too short for most sites, but the value is chosen to be as low impact as possible if you forget to give another value.
- **-c config** allows you to specify a MySQL config file other than $HOME/.my.cnf, so you can host, user, and password.

Here's an example of running it with verbose output:

```
$ full-slow-log -v
Discovering slow_query_log=1
Discovering slow_query_log_file=mysql-slow.log
Discovering long_query_time=60.000000
Setting long_query_time=0
Setting slow_query_log_file=mysql-slow.log-full-20121122112413
Setting slow_query_log=1
Flushing slow query log
Sleeping 5 seconds... done.
Restoring slow_query_log_file=mysql-slow.log
Restoring long_query_time=60.000000
Restoring slow_query_log=1
Flushing logs during restore
```

Notice that the script also redirects the slow query log to a new file, with a filename based on the timestamp. This is so you have a distinct file that contains only the specific time range of logs you collected.

The restoration of settings is in a "trap" which is a shell scripting feature that serves as both an exit handler and signal handler. So if you interrupt the script before it's done, you have some assurance that it will do the right thing to restore settings anyway.

My full-slow-log script is now available on a Github project (along with a few other experimental scripts I have written). See https://github.com/billkarwin/bk-tools

Rotating MySQL slow logs safely

By Peter Boros

Here are a some lessons we learned when logging a high volume of queries to the slow log.

Do not use copytruncate

Logrotate offers two techniques for log rotation (your log rotation scheme likely offers similar options with a different name):

1. *copytruncate* – Copies the file to a new name, and then truncates the original file.
2. *no copytruncate* – Uses the `rename()` system call to move the file to a new name, and then expects the daemon to be signaled to reopen its log file.

MySQL has a mutex for slow log writes. Truncation can block MySQL because the OS serializes access to the inode during the truncate operation. This problem is particularly evident when using the ext3 file system (instead of xfs).

Use FLUSH LOGS instead of sending SIGHUP

When copytruncate is disabled, MySQL must be told to reopen the slow log file. There are two options for signaling:

1. Send a HUP signal to the mysqld process.
2. Use the mysql console or mysqladmin utility to `FLUSH LOGS;`

These options should be equivalent, but MySQL bug 65481 explains that the HUP signal also flushes tables in addition to logs. Flushing tables can impact running queries.

Disable MySQL slow logs during rotation

Flushing logs takes time. Meanwhile, queries are still being executed. To prevent MySQL from filling the slow log buffer, we disable the MySQL slow logs temporarily during log rotation.

Putting it all together

```
/var/mysql/slow_query.log {
    nocompress
    create 660 mysql mysql
    size 1G
    dateext
    missingok
    notifempty
    sharedscripts
    postrotate
       /usr/local/bin/mysql -e 'select @@global.long_query_time into @lqt_save; set global long_query_time=2000; select sleep(2); FLUSH LOGS; select sleep(2); set global long_query_time=@lqt_save;'
    endscript
    rotate 150
```

Impact of logging on MySQL's performance

By Aleksandr Kuzminsky

Introduction

When people think about Percona's microslow patch immediately a question arises how much logging impacts on performance. When we do performance audit often we log every query to find not only slow queries. A query may take less than a second to execute, but a huge number of such queries may significantly load a server. On one hand logging causes sequential writes which can't impair performance much, on other hand when every query is logged there is a plenty of write operations and obviously performance suffers. Let's investigate how much.

I took DBT2, an OSDL's implementation of TPC-C.

Hardware used

The benchmark was run on a DELL server running CentOS release 4.7 (Final)
There are four CPUs Intel(R) Xeon(R) CPU 5150 @ 2.66GHz, 32GB RAM. There are 8 disks in RAID10(a mirror of 4+4 striped disks).

Software

It was used MySQL 5.0.75-percona-b11 on CentOS release 4.7

MySQL setting

There were two cases considered CPU- and IO-bound.
Each case had three options:

- logging turned off;
- logging queries which take more than a second to execute;
- logging every query;

MySQL was run with default settings except following:

```
[mysqld]
user=root
max_connections=3000
innodb_log_file_size=128M
innodb_flush_log_at_trx_commit=1
innodb_file_per_table
table_cache=2000
```

Depending on workload different InnoDB buffer was used.
In CPU-bound case

```
innodb_buffer_pool_size=2G
```

In IO-bound case

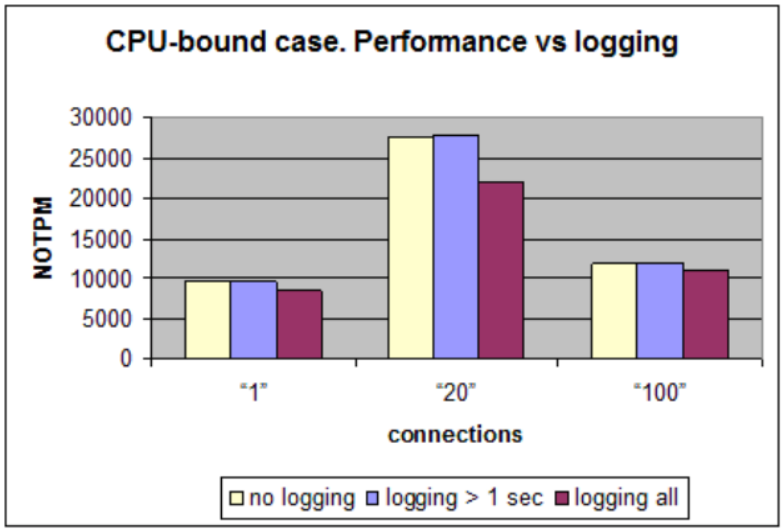
```
innodb_buffer_pool_size=512M
innodb_flush_method=O_DIRECT
```

DBT2 settings
For CPU-bound case number of warehouses was 10(1.31GiB). In case of IO-bound load — 100 warehouses which is 10GiB in terms of database size.
The test was run with 1, 20 and 100 database connections
Results
To reduce random error the test was run 3 times per each parameter set.
The metric of a DBT2 test is NOTPM (New Order Transaction per Minute) the more the better.

CPU-bound case – 10 warehouses

Database size 1.31 GiB

# of connections	No logging, NOTPM	Logging queries >1 SEC, NOTPM	ratio 1 sec / no_logging	Logging all queries, NOTPM	Ratio all_logging / no_logging
1	9607	9632	1.00	8434	0.88
20	27612	27720	1.00	22105	0.80
100	11704	11741	1.00	10956	0.94

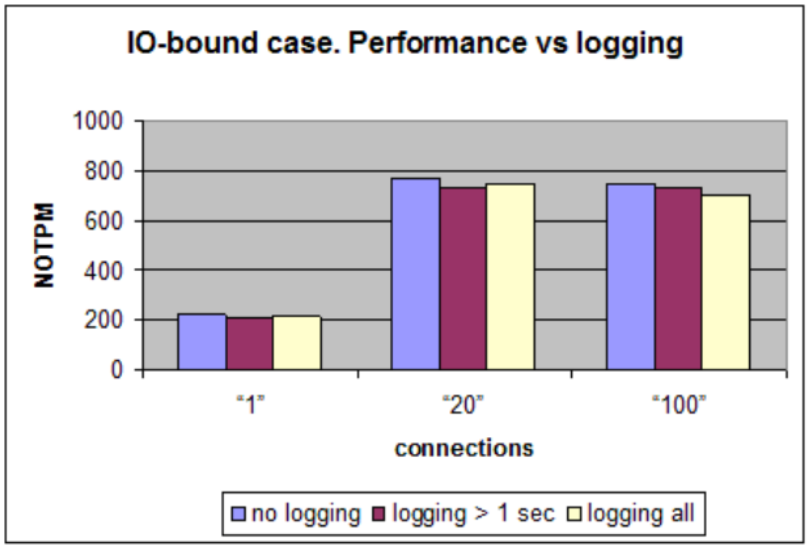


We see here that logging all queries decreases MySQL's performance on 6-20% depending on a number of connections to a database.

It should be noted during the test it was executed roughly 20-25k queries per second. If all queries are logged — a slow log is populated at rate about 10MB/sec. This is the highest rate observed.
IO-bound case's 100 warehouses

Database size 10GiB

# of connections	No logging, NOTPM	Logging queries > 1 sec, NOTPM	Ratio no_logging / 1 sec_logging	Logging all, NOTPM	Ratio no_logging / all_logging
1	225 Â± 9	211 Â± 3	0.94	213 Â± 9	0.95
20	767 Â± 41	730 Â± 35	0.95	751 Â± 33	0.98
100	746 Â± 54	731 Â± 12	0.98	703 Â± 36	0.94



In this case every test was run 5 times and random measurement error was calculated. As it is seen from the chart above the performance almost doesn't depend on logging — the difference doesn't exceed the measurement error.

The query rate in this case is about 1000 per second.

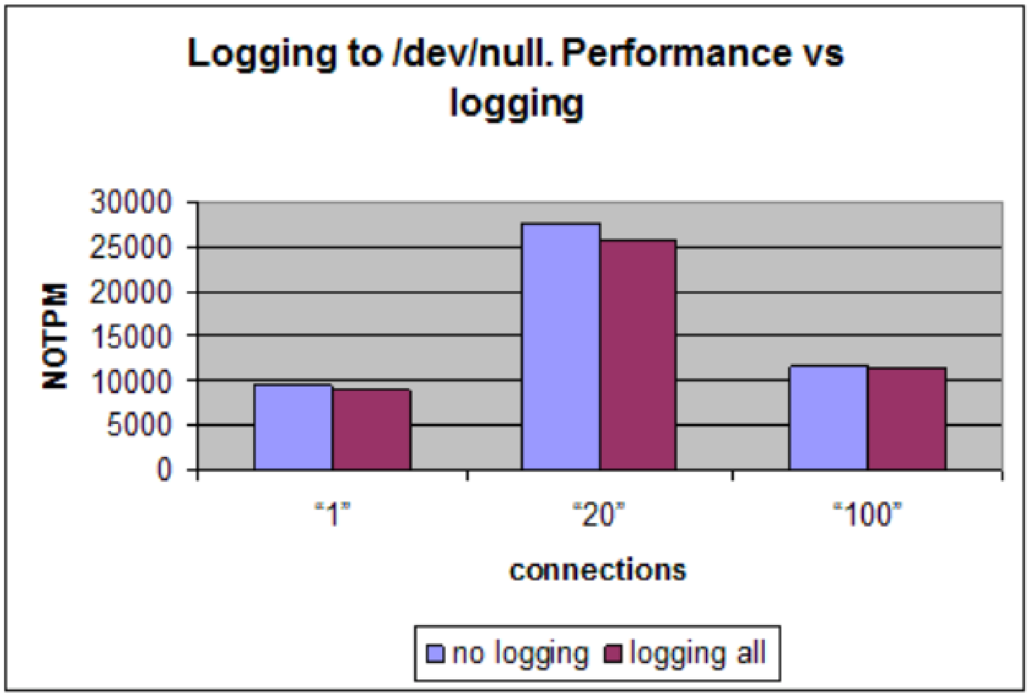
Logging to /dev/null

It is interesting to know how much from performance degradation caused by the microslow patch itself. Let's do the same tests but logging to /dev/null.

Please proceed to the next page

CPU-bound case – 10 warehouses, Database size: 1.31 GiB

# of connections	No logging, NOTPM	Logging all queries, NOTPM	Ratio all_logging /no_logging
1	9512	8943	0.94
20	27675	25869	0.93
100	11609	11236	0.97



Conclusion

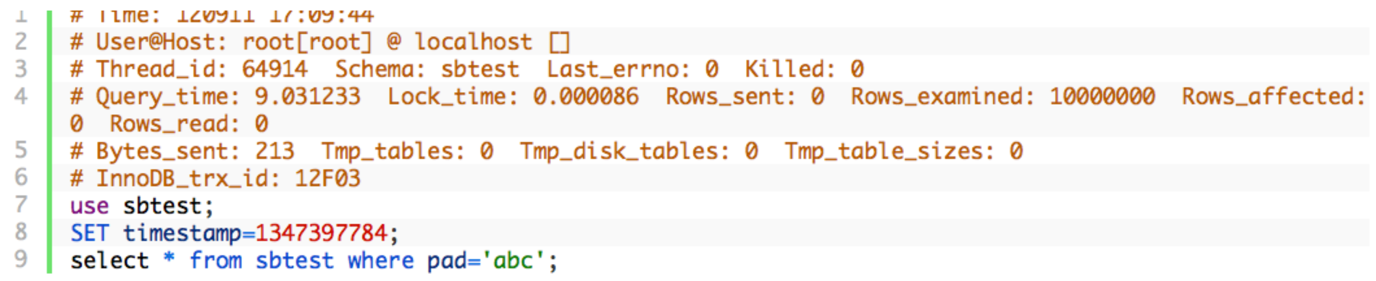
From the all tests above there are two conclusions can be made:

1. It is safe to log slow queries with execution time bigger than a second without worry about performance impact in case of CPU-bound workload. The performance impact is negligibly small in IO-bound workload even if all queries are logged.
2. In general logging **all** queries can hurt MySQL and you should consider the load while using it, especially in CPU-bound case.

How to find MySQL queries worth optimizing?

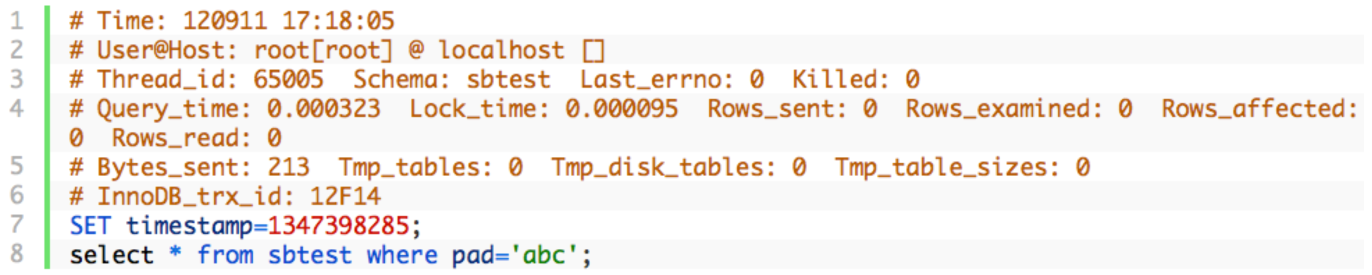
By Peter Zaitsev

One question I often get is how one can find out queries which should be optimized. By looking at pt-query-digest report it is easy to find slow queries or queries which cause the large portion of the load on the system but how do we know whenever there is any possibility to make this query run better ? The full answer to this question will indeed require complex analyses as there are many possible ways query can be optimized. There is however one extremely helpful metric which you can use – ratio between rows sent and rows analyzed. Lets look at this example:



```
# Time: 120911 17:09:44
# User@Host: root[root] @ localhost □
# Thread_id: 64914  Schema: sbtest  Last_errno: 0  Killed: 0
# Query_time: 9.031233  Lock_time: 0.000086  Rows_sent: 0  Rows_examined: 10000000  Rows_affected: 0  Rows_read: 0
# Bytes_sent: 213  Tmp_tables: 0  Tmp_disk_tables: 0  Tmp_table_sizes: 0
# InnoDB_trx_id: 12F03
use sbtest;
SET timestamp=1347397784;
select * from sbtest where pad='abc';
```

The query in this case has sent zero rows (as there are no matches) but it had to examine 10Mil rows to produce result. What would be good scenario ? – query examining same amount of rows as they end up sending. In this case if I index the table I get the following record in the slow query log:



```
# Time: 120911 17:18:05
# User@Host: root[root] @ localhost □
# Thread_id: 65005  Schema: sbtest  Last_errno: 0  Killed: 0
# Query_time: 0.000323  Lock_time: 0.000095  Rows_sent: 0  Rows_examined: 0  Rows_affected: 0  Rows_read: 0
# Bytes_sent: 213  Tmp_tables: 0  Tmp_disk_tables: 0  Tmp_table_sizes: 0
# InnoDB_trx_id: 12F14
SET timestamp=1347398285;
select * from sbtest where pad='abc';
```

Rows_examined=0 same as Rows_sent meaning this query is optimized quite well. Note you may be thinking in this case there is no database access happening at all – you would be wrong. The index lookup is being perform but as only actual rows which are found and returned up to the top level MySQL part for processing are counted the Rows_examined remains zero.

It looks simple so far but it also a huge oversimplification. You can do such simple math only to the queries without aggregate functions/group by and only to ones which examine one table only. What is about queries which query more than one table?

```
# Time: 120911 17:25:22
# User@Host: root[root] @ localhost □
# Thread_id: 65098  Schema: sbtest  Last_errno: 0  Killed: 0
# Query_time: 0.000234  Lock_time: 0.000063  Rows_sent: 1  Rows_examined: 1  Rows_affected:
0  Rows_read: 1
# Bytes_sent: 719  Tmp_tables: 0  Tmp_disk_tables: 0  Tmp_table_sizes: 0
# InnoDB_trx_id: 12F1D
SET timestamp=1347398722;
select * from sbtest a,sbtest b where a.id=5 and b.id=a.k;

mysql> explain select * from sbtest a,sbtest b where a.id=5 and b.id=a.k;
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | a     | const | PRIMARY,k     | PRIMARY | 4       | const |    1 |       |
|  1 | SIMPLE      | b     | const | PRIMARY       | PRIMARY | 4       | const |    1 |       |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+-------+
2 rows in set (0.00 sec)
```

In this case we actually join 2 tables but because the access type to the tables is “const” MySQL does not count it as access to two tables. In case of “real” access to the data it will:

```
# Time: 120911 17:28:12
# User@Host: root[root] @ localhost □
# Thread_id: 65099  Schema: sbtest  Last_errno: 0  Killed: 0
# Query_time: 0.000273  Lock_time: 0.000052  Rows_sent: 1  Rows_examined: 2  Rows_affected:
0  Rows_read: 1
# Bytes_sent: 719  Tmp_tables: 0  Tmp_disk_tables: 0  Tmp_table_sizes: 0
# InnoDB_trx_id: 12F23
SET timestamp=1347398892;
select * from sbtest a,sbtest b where a.k=2 and b.id=a.id;

+----+-------------+-------+--------+---------------+---------+---------+-------------+------+-----
--+
| id | select_type | table | type   | possible_keys | key     | key_len | ref         | rows |
Extra |
+----+-------------+-------+--------+---------------+---------+---------+-------------+------+-----
--+
|  1 | SIMPLE      | a     | ref    | PRIMARY,k     | k       | 4       | const       |    1
|       |
|  1 | SIMPLE      | b     | eq_ref | PRIMARY       | PRIMARY | 4       | sbtest.a.id |    1
|       |
+----+-------------+-------+--------+---------------+---------+---------+-------------+------+-----
--+
2 rows in set (0.00 sec)
```

In this case we have 2 rows analyzed for each row set which is expected as we have 2 (logical) tables used in the query. This rule also does not work if you have any group by in the query:

```
# Time: 120911 17:31:48
# User@Host: root[root] @ localhost □
# Thread_id: 65144  Schema: sbtest  Last_errno: 0  Killed: 0
# Query_time: 5.391612  Lock_time: 0.000121  Rows_sent: 2  Rows_examined: 10000000  Rows_affected:
0  Rows_read: 2
# Bytes_sent: 75  Tmp_tables: 0  Tmp_disk_tables: 0  Tmp_table_sizes: 0
# InnoDB_trx_id: 12F24
SET timestamp=1347399108;
select count(*) from sbtest group by k;
```

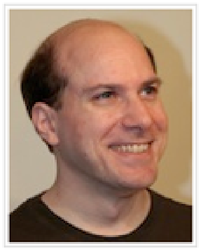
This only sends 2 rows while scanning 10 million, while we can't really optimize this query in a simple way because scanning all that rows are actually needed to produce group by results. What you can think about in this case is removing group by and aggregate functions. Then query would become "select * from sbtest" which would send all 10M rows and hence there is no ways to simply optimize it.

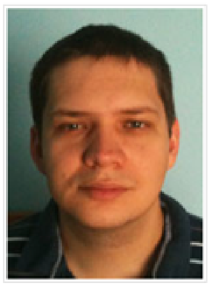
This method does not only provide you with "yes or no" answer but rather helps to understand how much optimization is possible. For example I might have query which uses some index scans 1000 rows and sends 10… I still might have opportunity to reduce amount of rows it scans 100x, for example by adding combined index.

So what is the **easy way to see if query is worth optimizing** ?
- see how many rows query sends after group by, distinct and aggregate functions are removed (A)
- look at number of rows examined divided by number of tables in join (B)
- if B is less or equals to A your query is "perfect"
- if B/A is 10x or more this query is a very serious candidate for optimization.

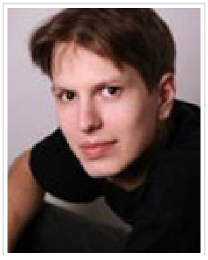
This is simple method and it can be used with **pt-query-digest** very well as it reports not only average numbers but also the outliers.

About the authors


Bill Karwin, Percona's Senior Knowledge Manager, has been a software engineer for over 20 years, developing and supporting applications, libraries and servers such as Zend Framework for PHP 5, the InterBase relational database, and the Enhydra Java application server. Throughout his career, Bill has shared his knowledge to help other programmers achieve success and productivity. Bill has answered thousands of questions, giving him a unique perspective on SQL mistakes that most commonly cause problems. He authored the book "SQL Antipatterns," collecting frequent blunders and showing better solutions.


Peter Boros joined Percona's European consulting team in May 2012. Before joining Percona, among many other things, he worked at Sun Microsystems, specialized there in performance tuning and was a DBA at Hungary's largest social networking site. He also taught many Oracle University MySQL courses. He has been using and working with open source software from early 2000s. Peter's first and foremost professional interest is performance tuning. He currently lives in Budapest, Hungary with his wife and son.

Aleksandr Kuzminsky is highly experienced in a variety of different Unix variants and network services, as well as in Operational Support Systems and Business Support Systems (OSS/ BSS) development. Prior to joining Percona, he taught Cisco courses and built multi-service Multiprotocol Label Switching (MPLS) networks in Ukraine and in the Caucasian countries. At Percona, he specializes in InnoDB and MyISAM data recovery. Aleksandr lives with his family in the suburbs of Kiev. He enjoys playing with his young son and gardening.


Peter Zaitsev, Percona's CEO and founder, is arguably the world's foremost expert in MySQL performance and scaling, with a special expertise in hardware and database internals. Peter's work has contributed to dozens of MySQL appliances, storage engines, replication systems, and other technologies. Peter co-authored High Performance MySQL along with two other Percona experts. He is a frequently invited guest at open source conferences, and has been a sell-out speaker at the yearly MySQL User Conference since its inception. Peter currently lives in North Carolina with his wife and their two children.