PERCONA

InnoDB performance optimization

By Muhammad Irfan, Fernando Laudares, Nickolay Ihalainen and Michael Benshoof

Table of Contents

About Percona

Percona was founded in August 2006 by Peter Zaitsev and Vadim Tkachenko and now employs a global network of experts with a staff of more than 100 people. Our customer list is large and diverse, including Fortune 50 firms, popular websites, and small startups. We have over 1,800 customers and, although we do not reveal all of their names, chances are we're working with every large MySQL user you've heard about. To put Percona's MySQL expertise to work for you, please contact us.

Contact Us 24 Hours A Day

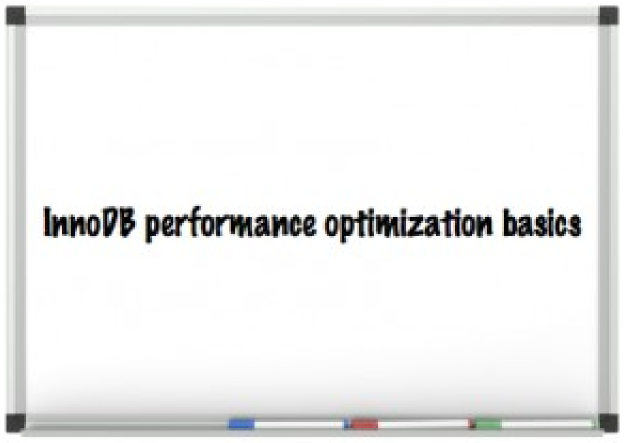
Is this an emergency? Get immediate assistance from Percona Support 24/7. Click here

Skype: oncall.percona
GTalk: oncall@percona.com
AIM (AOL Instant Messenger): oncallpercona
Telephone direct-to-engineer: +1-877-862-4316 or
UK Toll Free: +44-800-088-5561
Telephone to live operator: +1-888-488-8556
Customer portal: https://customers.percona.com/

Sales North America	(888) 316-9775 or (208) 473-2904
Sales Europe	+44-208-133-0309 (UK) 0-800-051-8984 (UK) 0-800-181-0665 (GER)
Training	(855) 55TRAIN or (925) 271-5054

InnoDB performance optimization basics

By Muhammad Irfan



I recently stumbled upon a post in the MySQL Performance Blog that Peter Zaitsev published back in 2007 titled "Innodb Performance Optimization Basics." It's a great post and reading it inspired me to examine what's changed in the nearly six years that have followed in terms of MySQL, Percona Server – as well as in all of the other now-available infrastructures.

And a lot has in fact changed! In this post I am going to highlight most of the InnoDB parameters critical for InnoDB – specifically from a performance perspective. I'm a support engineer and I can tell you that Percona Support gets many questions related to the right sizing of basic InnoDB parameters. So hopefully this post will help others with similar questions and issues.

Hardware:
For larger datasets, nowadays memory counted in hundreds of giga- and even in terabytes is not surprising. MySQL requires significant memory amounts in order to provide optimal performance. By caching hot datasets, indexes, and ongoing changes, InnoDB is able to provide faster response times and utilize disk IO in a much more optimal way. From a CPU standpoint, faster processors with many cores provide better throughput. CPUs with 32/64 cores or more are becoming common now, and the latest MySQL versions are able to utilize them much better then before. In terms of storage, SSD disks are replacing traditional spindles with great success, offering the best performance for the money. RAID 10 is still the most recommended level for most workloads, but first make sure your RAID controller is able to utilize the SSD drive's performance and will not become the actual bottleneck. There are also many PCI-e Flash drives out there if you need even more IOPS.

Operating System:

Linux is the most common operating system for high performance MySQL servers. Make sure to use modern filesystems, like EXT4 or XFS on Linux, combined with the most recent kernel. Each of them has it's own limits and advantages: for example XFS is fast in deleting large files, while EXT4 can provide better performance on fast SSD drives. Benchmark before you decide. Check this blog post to see how EXT4 can outperform XFS. You can use noatime and nodiratime options if you're using innodb_file_per_table and a lot of tables though benefit of these is minor. The default I/O scheduler in Linux is Completely Fair Queuing (CFQ), while Noop/Deadline will be much better in most cases.. Setting swappiness to zero is generally recommended for the MySQL dedicated host, which will lower the tendency of swapping. Make sure the MySQL host does not run out of memory. Swapping is bad for MySQL and defeats the purpose of caching in memory. To learn more about swapping, check this blog post.

MySQL Innodb Settings

From 5.5 InnoDB is the default engine, so these parameters are even more important for performance than before. The most important ones are:

- **innodb_buffer_pool_size:** InnoDB relies heavily on the buffer pool and should be set correctly, so be sure to allocate enough memory to it. Typically a good value is 70%-80% of available memory. More precisely, if you have RAM bigger than your dataset setting it bit larger should be appropriate with that keep in account of your database growth and re-adjust innodb buffer pool size accordingly. Further, there is improvement in code for InnoDB buffer scalability if you are using Percona Server 5.1 or Percona Server 5.5 You can read more about it here.

- **innodb_buffer_pool_instances:** Multiple innodb buffer pools introduced in InnoDB 1.1 and MySQL 5.5. In MySQL 5.5 the default value for it was 1 which is changed to 8 as new default value in MySQL 5.6. Minimum innodb_buffer_pool_instances should be lie between 1 (minimum) & 64 (maximum). Enabling innodb_buffer_pool_instances is useful in highly concurrent workload as it may reduce contention of the global mutexes.

- **Dump/Restore Buffer Pool:** This feature speed up restarts by saving and restoring the contents of the buffer pool. This feature is first introduced in Percona Server 5.5 you can read about it here. Also Vadim benchmark this feature You can read more about it in this post. Oracle MySQL also introduced it in version 5.6, To automatically dump the database at startup and shutdown set innodb_buffer_pool_dump_at_shutdown & innodb_buffer_pool_load_at_startup parameters to ON.

- **innodb_log_file_size:** Large enough InnoDB transaction logs are crucial for good, stable write performance. But also larger log files means that recovery process will slower in case of crash. However this is not such big issue since great improvements in 5.5. Default value has been changed in MySQL 5.6 to 50 MB from 5 MB (old default), but it's still too small size for many workloads. Also, in MySQL 5.6, if innodb_log_file_size is changed between restarts then MySQL will automatically resize the logs to match the new desired size during the startup process. Combined log file size is increased to almost 512 GB in MySQL 5.6 from 4 GB. To get the optimal logfile size please check this blog post.

- **innodb_log_buffer_size:** Innodb writes changed data record into It's log buffer, which kept in memory and it saves disk I/O for large transactions as it not need to write the log of changes to disk before transaction commit. 4 MB – 8 MB is good start unless you write a lot of huge blobs.

- **innodb_flush_log_at_trx_commit:** When innodb_flush_log_at_trx_commit is set to 1 the log buffer is flushed on every transaction commit to the log file on disk and provides maximum data integrity but it also has performance impact. Setting it to 2 means log buffer is flushed to OS file cache on every transaction commit. The implication of 2 is optimal and improve performance if you are not concerning ACID and can lose transactions for last second or two in case of OS crashes.

- **innodb_thread_concurrency:** With improvements to the InnoDB engine, it is recommended to allow the engine to control the concurrency by keeping it to default value (which is zero). If you see concurrency issues, you can tune this variable. A recommended value is 2 times the number of CPUs plus the number of disks. It's dynamic variable means it can set without restarting MySQL server.

- **innodb_flush_method:** DIRECT_IO relieves I/O pressure. Direct I/O is not cached, If it set to O_DIRECT avoids double buffering with buffer pool and filesystem cache. Given that you have hardware RAID controller and battery-backed write cache.

- **innodb_file_per_table:** innodb_file_per_table is ON by default from MySQL 5.6. This is usually recommended as it avoids having a huge shared tablespace and as it allows you to reclaim space when you drop or truncate a table. Separate tablespace also benefits for Xtrabackup partial backup scheme.

Along with that, there are lot of enhancements for InnoDB, specifically in Percona Server 5.5 and in Oracle MySQL 5.6. Persistent optimizer statistics is one of the features first introduced in Percona Server 5.5 that requires the enabling of the innodb_use_sys_stats_table in XtraDB. You can read more about it here. This feature is now included in Oracle MySQL 5.6, too. In MySQL 5.6 persistent stats are stored in two new tables: mysql.innodb_index_stats and mysql.innodb_table_stats. Through this query plans are much more accurate and consistent. You can read more about it in documentation. Also Percona Server 5.5 introduced a Thread Pool feature which is ported from MariaDB. You can read more about it in this documentation. On a related note, I recommend reading this blog post from Vadim on the Thread Pool feature.

Percona Server free and open source. An enhanced drop in Oracle MySQL replacement and some of the mentioned features are only applicable to Percona Server.

There are bunch of other options which you may want to tune but in this post we focus only InnoDB specifically.

Application tuning for Innodb:
Especially when coming from a MyISAM background, there will be some changes you would like to make with your application. First make sure you're using transactions when doing updates, both for sake of consistency and to get better performance. Next if your application has any writes be prepared to handle deadlocks which may happen. Third you should review your table structure and see how you can get advantage of Innodb properties – clustering by primary key, having primary key in all indexes (so keep primary key short), fast lookups by primary keys (try to use it in joins), large unpacked indexes (try to be easy on indexes).

Conclusion:
We covered almost all basic and important InnoDB parameters, OS related tweaking and hardware for optimal MySQL server performance. By setting all mentioned variables appropriately certainly help to boost overall MySQL server performance.

InnoDB file formats: Here is one pitfall to avoid

By Fernando Laudares

Compressed tables is an example of an InnoDB feature that became available with the **Barracuda** file format, introduced in the InnoDB plugin. They can bring significant gains in raw performance and scalability: given the data is stored in a compressed format the amount of memory and disk space necessary to hold it and move it around (disk/memory) is lower, thus making them attractive for servers equipped with SSD drives of smaller capacity.

The notion of "file formats" (defined by the variable innodb_file_format) was first introduced when InnoDB was still a plugin. The evolution of InnoDB has lead to the development of new features and some of them required the support of new on-disk data structures. That means those particular features (like compressed tables) will only work with the newer file format. To make things clear and help manage compatibility issues when upgrading and (specially) downgrading MySQL the original file format started being referred to as **Antelope**.

The default file format in MySQL 5.6 and the latest 5.5 releases is Antelope. Note this can be a bit confusing as the first releases of 5.5 (until 5.5.7) introduced the new file format as being the default one, a decision that was later reversed to assure maximum compatibility in replication configurations comprised of servers running different versions of MySQL. To be sure about which file format is the one set as default in your server you can issue:

```
mysql> SHOW VARIABLES LIKE 'innodb_file_format';
```

The important lesson here that motivated me to write this post is that the file format can only be defined for tablespaces – not tables, in general. This is documented in the manual but maybe not entirely clear:

> innodb_file_format: The file format to use for new InnoDB tables. Currently, Antelope and Barracuda are supported. This applies only for tables that have their own tablespace, so for it to have an effect, innodb_file_per_table must be enabled. The Barracuda file format is required for certain InnoDB features such as table compression.

Even if you configure your server with innodb_file_format=Barracuda and recreate the datadir and basic tables with the script mysql_install_db, the common tablespace will always use Antelope. So, to create tables under the new file format it is imperative you use innodb_file_per_table. Although this requirements is documented what might be misleading here is the fact there's no error being issued if you set the file format to Barracuda and create a new compressed table without having innodb_one_file_per_table set – only a couple of warnings, if you pay close attention. Here's an example:

```
mysql> SET GLOBAL innodb_file_format=Barracuda;
Query OK, 0 rows affected (0.00 sec)

mysql> create table test.testA (id int) row_format=Compressed;
Query OK, 0 rows affected, 2 warnings (0.96 sec)
```

If you do choose to check the warnings, you'll find:

```
mysql> show WARNINGS;
+---------+------+--------------------------------------------------------------+
| Level | Code | Message |
+---------+------+--------------------------------------------------------------+
| Warning | 1478 | InnoDB: ROW_FORMAT=COMPRESSED requires innodb_file_per_table. |
| Warning | 1478 | InnoDB: assuming ROW_FORMAT=COMPACT. |
+---------+------+--------------------------------------------------------------+
2 rows in set (0.00 sec)
```

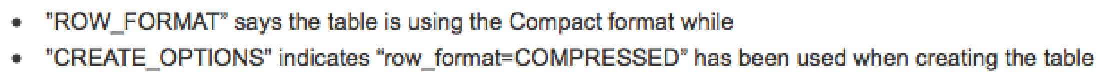
This happens when *innodb_strict_mode* is turned **OFF**, as it usually is. If it was turned ON the table creation would fail with the following error:

```
mysql> create table test.testA (id int) row_format=Compressed;
ERROR 1031 (HY000): Table storage engine for 'testA' doesn't have this option
```

Now, let's take a look at what the INFORMATION_SCHEMA tell us about this table:

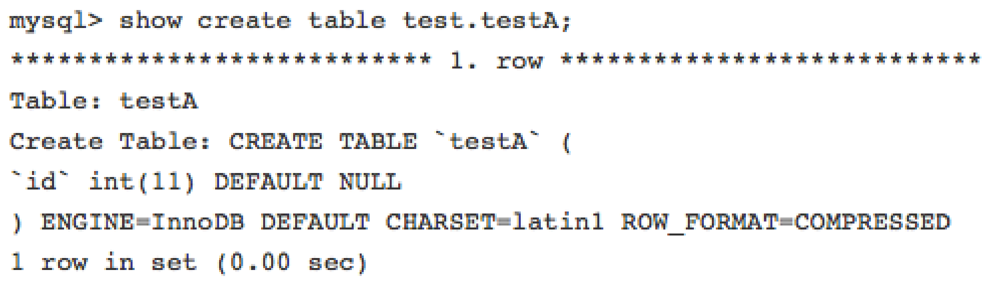
```
  mysql> SELECT * FROM information_schema.tables WHERE table_schema='test' and
  table_name='testA'\G
  *************************** 1. row ***************************
  TABLE_CATALOG: def
  TABLE_SCHEMA: test
  TABLE_NAME: testA
  TABLE_TYPE: BASE TABLE
  ENGINE: InnoDB
  VERSION: 10
  ROW_FORMAT: Compact
  TABLE_ROWS: 0
  AVG_ROW_LENGTH: 0
  DATA_LENGTH: 16384
  MAX_DATA_LENGTH: 0
  INDEX_LENGTH: 0
  DATA_FREE: 0
  AUTO_INCREMENT: NULL
  CREATE_TIME: 2014-01-07 14:21:05
  UPDATE_TIME: NULL
  CHECK_TIME: NULL
  TABLE_COLLATION: latin1_swedish_ci
  CHECKSUM: NULL
  CREATE_OPTIONS: row_format=COMPRESSED
  TABLE_COMMENT:
  1 row in set (0.00 sec)
```

There’s two at-first-look “contradictory” fields here:



- "ROW_FORMAT" says the table is using the Compact format while
- "CREATE_OPTIONS" indicates “row_format=COMPRESSED” has been used when creating the table

The one to consider is ROW_FORMAT: CREATE_OPTION is used to store the options that were used at the moment the table was created and is evoked by the SHOW CREATE TABLE statement to “reconstruct” it:



```
mysql> show create table test.testA;
*************************** 1. row ***************************
Table: testA
Create Table: CREATE TABLE `testA` (
`id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1 ROW_FORMAT=COMPRESSED
1 row in set (0.00 sec)
```

Conclusion

A customer contacted us asking how he could get a list of the tables using the compression format, which we can obtain by interrogating INFORMATION_SCHEMA:

mysql> SELECT TABLE_NAME, ROW_FORMAT FROM INFORMATION_SCHEMA.TABLES WHERE ROW_FORMAT=Compressed’;

To their surprise this statement returned an empty set. We verified that the tables created by them specified ROW_FORMAT=Compressed but as shown in this article this method is not to be trusted – ask the INFORMATION_SCHEMA instead.

How to move the InnoDB log sequence number (LSN) forward

By Nickolay Ihalainen

This chapter focuses on the problem of the InnoDB log sequence number being in the future.

Preface: What is an InnoDB log sequence number?

The Log sequence number (LSN) is an important database parameter used by InnoDB in many places.The most important use is for crash recovery and buffer pool purge control.

Internally, the InnoDB LSN counter never goes backward. And, when InnoDB writes 50 bytes to the redo logs, the LSN increases by 50 bytes. As such we can count LSN in megabytes, gigabytes and etc.

Now for the problem: LSN being in the future!

When you have set innodb_force_recovery like this:

```
innodb_force_recovery=6
```

... and then issue a data affecting query.

For example, if you are dropping a corrupted table after doing a mysqldump for backup purposes, InnoDB will save an incorrect LSN to ibdata1 and you will have an error message in the mysqld error log after each server restart:

```
120323 4:38:52 InnoDB: Error: page 0 log sequence number 6094071743825
InnoDB: is in the future! Current system log sequence number 10000.
```

The solution: some methods to change the LSN

Usually the safest method to fix the LSN is to insert/delete the required amount of data.

But what if an old LSN was several TB? Several options are available

a. Use your backup
b. Convert all tables to myisam, remove ibdata1 & ib_logfile*, after server restart, convert all tables back to InnoDB
c. mysqldump/restore
d. Black magic if you have a huge database

If you can’t use methods a-c the only way to get correct LSN is make some unsafe step, like change innodb files or modify mysqld memory:

1. Make sure that you have the debuginfo package installed
2. No queries should be executed (at all!) during operation, otherwise the LSN may be updated
3. gdb -p `pgrep -x mysqld`
   ```
   gdb) p log_sys->lsn
   $1 = 12300
   (gdb) set log_sys->lsn = 12300000;
   Invalid character ';' in expression.
   (gdb) set log_sys->lsn = 12300000
   (gdb) c
   ```
4. Shutdown mysqld, this should be a clean normal shutdown
5. Check if the correct LSN us shown in the error log
6. Start mysqld and check if the correct LSN is shown
   ```
   LOG
   ---
   Log sequence number 12300000
   Log flushed up to 12300000
   Last checkpoint at 12300000
   ```
7. Insert something, check that the LSN is changing

Possible issues: How to avoid database corruption after you change the LSN

- Of course, because this insider method relies on the internal mysql structure it could fail with future versions of InnoDB.
  I have modified LSN in memory for 5.5.32-rel31.0-549.precise during preparations for this blog post.
  Please check the working of this method on the version you are using on a staging system first.
- Also it’s a really bad idea to update the LSN online on a production server, because it will mean recovery will fail if your server ends up crashing.
- Server should be completely idle with the same LSN value for a while.
  ‘Log sequence number’, ‘Log flushed up to’, ‘Last checkpoint at’ are all the same on idle server.
  If the server is not idle enough, and you don’t see changes in SHOW MASTER STATUS\G output,
  try to SET GLOBAL innodb_fast_shutdown=0 and restart the server.
- A server restart is required to write the LSN changes to transaction log.
- The system should be stable before change: A mysqld crash during server restart could cause data corruption. Please check that restart procedure is fast, and that “recovery” is not in progress in the mysqld error log

How could corruption happen to start with?

If mysqld crashes, InnoDB will do a crash recovery on mysqld restart.
InnoDB crash recovery applies the transaction log from the last on-disk checkpoint until the ‘Log flushed up to’ position.
If the ‘Log flushed up to’ position is equal to a non-existing position, InnoDB will try to apply old events, because transaction logs are organized in a ring buffer manner. There you can additionally enforce the change if you will re-create transaction logs right before the change. At maximum you will have a mysqld server crash without significant data corruption.

Last warning: ALWAYS have a backup before modifying memory with gdb, especially if you are doing something untested with your particular version of MySQL for the first time.

MySQL 5.6 - InnoDB Memcached Plugin as a caching layer

By Michael Benshoof

A common practice to offload traffic from MySQL 5.6 is to use a caching layer to store expensive result sets or objects. Some typical use cases include:

- Complicated query result set (search results, recent users, recent posts, etc)
- Full page output (relatively static pages)
- Full objects (user or cart object built from several queries)
- Infrequently changing data (configurations, etc)

In pseudo-code, here is the basic approach:

```
data = fetchCache(key)

if (data) {
  return data
}

data = callExpensiveFunction(params)
storeCache(data, key)
return data
```

Memcached is a very popular *(and proven)* option used in production as a caching layer. While very fast, one major potential shortcoming of memcached is that it is not persistent. While a common design consideration when using a cache layer is that "data in cache may go away at any point", this can result in painful warmup time and/or costly cache stampedes.

Cache stampedes can be mitigated through application approaches (semaphores, pre-expiring and populating, etc), but those approaches are more geared towards single key expiration or eviction. However, they can't help overall warmup time when the entire cache is cleared (think restarting a memcache node). This is where a persistent cache can be invaluable.

Enter **MySQL 5.6** with the **memcached plugin**…

As part of the standard MySQL 5.6 GA distribution, there is a memcached plugin included in the base plugin directory (**/usr/lib64/mysql/plugin/libmemcached.so**) that can be stopped and started at runtime. In a nutshell, here is how one would start the memcached plugin:

```
mysql> install plugin daemon_memcached soname 'libmemcached.so';
```

In an effort to not re-invent the wheel, here is a link to the full documentation for setting up the plugin:

http://dev.mysql.com/doc/refman/5.6/en/innodb-memcached-setup.html

As a quick benchmark, I ran some batches of **fetch** and **store** against both a standard memcached instance and a minimally tuned MySQL 5.6 instance running the memcached plugin. Here are some details about the test:

- Minimal hardware (vBox instances on MacBook Pro)
    - Centos 6.4
    - Single core VM
    - 528M RAM
    - Host-Only network
    - 1 Box with http/php, 1 box with memcache or mysql started
- PHP script
    - Zend framework
    - libmemcached PECL module
    - Zend_Cache_Backend_Libmemcached

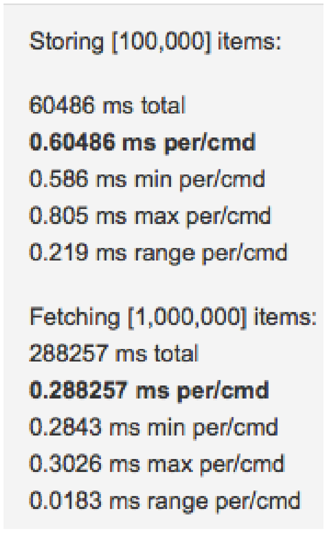
Here is the rough code for this benchmark:

```
// Identical config/code for memcached vs InnoDB
$frontendOpts = array(
  'caching' => true,
  'lifetime' => 3600,
  'automatic_serialization' => true
);

$memcacheOpts = array(
  'servers' =>array(
    array(
      'host'   => '192.168.57.51',
      'port'   => 11211,
      'weight' => 1,
    )
  ),
  'client' => array(
    'compression' => true,
  ),
);

$cache = Zend_Cache::factory('Core', 'Libmemcached', $frontendOpts, $memcacheOpts);
$timer->start();

for ($i = 0; $i < 100000; $i++) {
  $cache->save(new Object(), "key_$i");
}

$totalTimeStore = $timer->stop();
$avgTimeStore = $totalTimeStore / 100000;
$timer->start();

for ($i = 0; $i < 10; $i++) {
  for ($i = 0; $i < 100000; $i++) {
    $obj = $cache->load("key_$i");
  }
}

$totalTimeFetch = $timer->stop();
$avgTimeFetch = $totalTimeFetch / 1000000;
```

While this benchmark doesn't show any multi-threading or other advanced operation, it is using identical code to eliminate variation due to client libraries. The only change between runs is on the remote server (stop/start memcached, stop/start plugin).

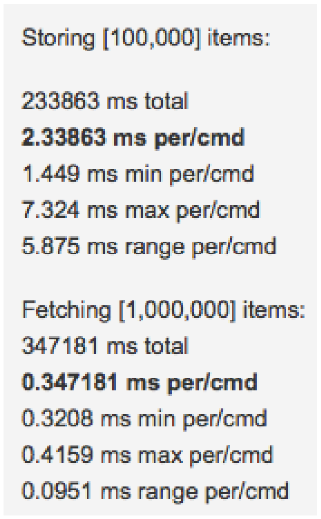
As expected, there is a slowdown for write operations when using the InnoDB version. But there is also a slight increase in the average fetch time. Here are the raw results from this test run (**100,000 store operations, 1,000,000 fetch operations**):

Standard Memcache:



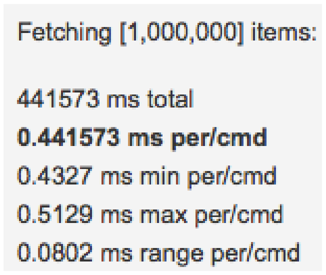
```
Storing [100,000] items:

60486 ms total
0.60486 ms per/cmd
0.586 ms min per/cmd
0.805 ms max per/cmd
0.219 ms range per/cmd

Fetching [1,000,000] items:
288257 ms total
0.288257 ms per/cmd
0.2843 ms min per/cmd
0.3026 ms max per/cmd
0.0183 ms range per/cmd
```

InnoDB Memcache:



```
Storing [100,000] items:

233863 ms total
2.33863 ms per/cmd
1.449 ms min per/cmd
7.324 ms max per/cmd
5.875 ms range per/cmd

Fetching [1,000,000] items:
347181 ms total
0.347181 ms per/cmd
0.3208 ms min per/cmd
0.4159 ms max per/cmd
0.0951 ms range per/cmd
```

InnoDB MySQL Select (same table):



```
Fetching [1,000,000] items:

441573 ms total
0.441573 ms per/cmd
0.4327 ms min per/cmd
0.5129 ms max per/cmd
0.0802 ms range per/cmd
```

Keep in mind that the entire data set fits into the buffer pool, so there are no reads from disk. However, there is write activity stemming from the fact that this is using InnoDB under the hood (redo logs, etc).

Based on the above numbers, here are the relative differences:

- InnoDB store operation was **280%** higher (~1.73 ms/op)
- InnoDB fetch operation was **20%** higher (~.06 ms/op)
- MySQL Select showed **27%** increase over InnoDB fetch (~.09 ms/op)
    - This replaced $cache->load() with $db->query("SELECT * FROM memcached.container WHERE id='key_id'");
    - **id** is PK of the container table

While there are increases in both operations, there are some tradeoffs to consider:

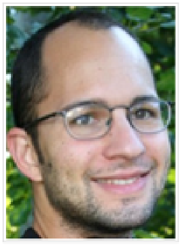
- Cost of additional memcached hardware
- Cost of operations time to maintain an additional system
- Impact of warmup time to application
- Cost of disk space on database server

Now, there are definitely other NoSQL options for persistent cache out there *(Redis, Couchbase, etc)*, but they are outside the scope of this investigation and would require different client libraries and benchmark methodology.

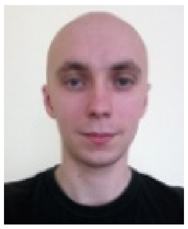
My goal here was to compare a transparent switch *(in terms of code)* and experiment with the memcache plugin. Even the use of HandlerSocket would require coding changes *(which is why it was also left out of the discussion).*

About the authors

Muhammad Irfan is a software engineer vastly experienced in the LAMP Stack. Prior to joining Percona, he worked in the role of MySQL DBA & LAMP Administrator, maintained high traffic websites, and worked as a Consultant. His professional interests focus on MySQL scalability and on performance optimization. In his spare time, he normally spends time with family and friends and loves to play and watch cricket.



Support Engineer Fernando Laudares focuses on the MySQL universe with a particular interest in understanding the intricacies of database systems. His work experience includes the architecture, deployment and maintenance of IT infrastructures based on Linux, open source software and a layer of server virtualization. From the basic services such as DHCP & DNS to identity management systems, but also including backup routines, configuration management tools and thin-clients.



Principal consultant Nickolay Ihalainen has a great deal of experience in both systems administration and programming. His experience includes extensive hands-on work with a broad range of technologies, including SQL, MySQL, PHP, C, C++, Python, Java, XML, OS parameter tuning (Linux, Solaris), caching techniques (e.g., memcached), RAID, file systems, SMTP, POP3, Apache, networking and network data formats, and many others. He is an expert in scalability, performance, and system reliability.

MySQL consultant Michael Benshoof enjoys designing extensible and flexible solutions to problems and has a strong background in HA systems. Prior to joining Percona, Michael spent several years in a DevOps role in a company that developed and maintained a SaaS application specializing in social networking. His experiences include application development and scaling, systems administration, along with database administration and design.