MySQL Configuration Tuning Handbook

Covers MySQL 5.7

SPEEDEMY

MYSQL PERFORMANCE TUNING SERIES

About the Author

Aurimas Mikalauskas is a data performance and scalability expert with over 12 years of experience with MySQL and a number of related technologies. Formerly a Percona architect for over 9 years, Aurimas is currently teaching performance optimization at speedemy.com.

While at Percona, Aurimas worked with companies such as BBC, EngineYard, Groupon, Boardreader, well known social networks and over a hundred of other companies large and small. While his focus was MySQL performance, scalability and high availability, he has also implemented a number of "one billion documents" full text search systems, large scale applications and he has published a number of articles on **The MySQL Performance Blog**.

To keep his skills sharp and up to date, Aurimas is still taking on challenging (read exciting) performance optimization or scalability cases. Feel free to reach out to him if you feel like you have an interesting challenge yourself.

Table of Contents

Introduction

With version 5.7, MySQL now has over 450 settings! That gives users like you and me a lot of power and flexibility to fine-tune it to perfection. But at the same time, it leaves a lot of room for errors.

Having spent over a decade tuning MySQL for best performance, running large scale as well as micro benchmarks, I'm pretty confident there are very few settings that actually matter for MySQL performance. And this e-book is here to guide you through them. Take a few minutes to study them with your environment in mind and you will never look at MySQL configuration the same way.

Before we get to configuration though, let's first discuss a question of MySQL lifetime, different MySQL distributions that are available on the market today, and the essentials of MySQL configuration tuning.

A quick note on Oracle killing MySQL

If you were at any of the MySQL conferences during past 6 years, you've probably heard this over and over again:

Oracle is going to kill MySQL

I want to acknowledge this FUD that has been spreading since April 2009, when Oracle acquired Sun (and consequently, MySQL).

First of all, Oracle was never planning to kill MySQL and very wisely so, given the number of other open source products that are already available on the market. They declared officially that they are going to keep MySQL Open to the world and they will keep improving it.

More importantly though, Oracle proved over and over again that they have learned the open source model and they keep doing a great job developing MySQL - the amount of improvements MySQL got during last 6 years is just amazing. Just take a look at MySQL 5.7 and you will realise that MySQL has never been better.

There are some people saying MySQL is now getting slower, which is only somewhat true if you look at isolated cases with zero-concurrency workload. However, when you look at the throughput, concurrency levels that MySQL is now able to withstand not loosing any stability, you will see it scales better than ever before. And beats most other products on the market, if you do a fair comparison.

Oh and let me name a few features that would have likely not seen the light of day for a number of years if not Oracle:

- InnoDB buffer pool instances
- InnoDB full text search
- JSON support
- Group Replication
- Multi-threaded replication
- Online buffer pool resize
- Native InnoDB partitions

Of course, they were able to monetise MySQL by creating some special Enterprise features, like Enterprise Monitor, Audit Plugin or Encryption. However when you think of it, that is a good thing, because it means this way they can keep MySQL free for the rest of us.

And MySQL ecosystem was quick to respond to this by creating Open Source alternatives for most of these products and are free to continue to do so.

Different MySQL distributions

Why don't we now spend a few minutes talking about different MySQL distributions that are available on the market these days. There's the *Oracle's Community MySQL*, *Percona Server*, *MariaDB* and *WebScaleSQL*. Let's quickly discuss what they have in common and what are the key differences so you know when to use which.

It's very important to note that, as long as you're using the same major version, you can use most of them interchangeably - you don't need to do migration, or even upgrades for your tables. You can just stop the server, switch the binaries and run the different version using the same data directory.

Your application won't notice the difference. Well at least from the functionality point of view. There can be some exceptions though. Let me walk you through it.

Community MySQL

This is the Oracle developed MySQL that, in my view, receives the most development efforts. It is also the version that is used as an upstream for all of the other MySQL versions - we'll talk about how each of them is using the upstream in each section separately.

In terms of performance, there are few important things to know:

1. **Community MySQL 5.1** does not have InnoDB plugin enabled by default, so if you run with default configuration, you will have a very poor performance at scale and also you will be missing out on a lot of features. That's not to say you should be using 5.1 at all, but if you are currently, beware that the very first thing you can do to take its performance to the next level is enable the innodb plugin.

2. **Community MySQL 5.5** already has strengthened InnoDB at its core, however there's still lots of performance improvements as well as features missing that you can find in an appropriate Percona Server with XtraDB and MariaDB with XtraDB versions, especially when it comes to stability at high write rates. Other than that, you can get pretty far with MySQL 5.5 as your base server.

3. **Community MySQL 5.6** got a lot of new features and performance improvements - including optimizer improvements, a number of new mutexes that were introduced to alleviate contention on the kernel mutex, performance schema improvements, multi-threaded purge threads, separate flush thread and so on and so forth. At this release, there are even fewer reasons to use an alternative version such as Percona or MariaDB and we'll talk about them in the appropriate sections.

4. **Community MySQL 5.7** - the newest MySQL GA release yet is currently the server with most features - including such new features like multi-source replication, JSON support, proper multi-threaded replication, online buffer pool resize, spatial data types for InnoDB, sys schema, etc. etc. Also, according to Oracle's benchmarks, it is the best performing MySQL server that's currently available on the market. (Beware that some

vendors have attempted to manipulate benchmark results showing better performance on their versions by using a different default configuration, so do not be confused by that).

I took a bit more time talking about Community MySQL, because it's used at the core of most of the other server variants. To explain what I mean by that, let me now switch to Percona Server.

Percona Server

Percona Server was originally launched around July 2008, when we stared releasing our builds of MySQL 5.0 and 5.1 with few additional patches - microsecond resolution slow query log, execution plan details, additional innodb statistics and user statistics patch from Google. Although it wasn't called Percona Server back then yet - it was called simply *Percona patches for MySQL*.

Unofficially however Percona has been building patched versions of MySQL for their customers even earlier - in addition to slow query log details that helped us as consultants understand better what is happening inside the server, those were also various performance fixes.

We always followed one simple rule - use latest upstream Community MySQL, remove any redundant features that are now implemented properly in MySQL and never break backwards compatibility. Meaning that if you run Percona Server 5.1.73, you should be able to switch back to MySQL 5.1.73 by simply swapping the binaries.

There were few exceptions to that, like with additional rollback segments for extremely write-heavy databases which, if enabled, didn't allow downgrade. However in all of these cases Percona made it very clear - I mean "DANGER in capital red" clear - that these changes are not backwards compatible and they were never enabled by default. And none of these incompatible changes are available in either 5.5, 5.6 or 5.7 versions of Percona Server.

So in a nutshell, if you are running MySQL 5.6.28 and you want to try out Percona Server, just remove Community MySQL 5.6.28, install Percona Server 5.6.28 and if you don't like something, you can switch back to Community edition the same way - it's just a matter of stopping one version and starting the other.

Now in terms of performance, here's few interesting things:

1. If you are running MySQL 5.1, do upgrade to **Percona Server 5.1** - you will be amazed how much better it is. On Percona Server you don't need to enable the innodb plugin, it will be enabled by default, but it has an enormous amount of performance and especially scalability improvements. Plus you will have all the additional features that showed up in MySQL only in versions 5.5 and 5.6.

2. **Percona Server 5.5** still has a great advantage over MySQL 5.5 - adaptive hash index partitions, faster checksums, buffer pool scalability improvements, much better

adaptive flushing algorithm plus a number of additional plugins that are otherwise only available in Enterprise MySQL, such as PAM authentication, InnoDB table import, Audit log and Thread pool. So it's both more stable and more scalable.

3. **Percona Server 5.6** has the same benefits of version 5.5 and it has a few performance improvements, but the difference over MySQL 5.6 is now only visible with high end hardware - super fast flash storage, hundreds of gigabytes of RAM and tens of CPU cores.

4. At this time, **Percona Server 5.7** is not yet generally available, however I would only expect Percona Server to make a difference in edge cases and for most common even high performance workloads, MySQL 5.7 might be good enough.

There's one big advantage of Percona Server I didn't talk much about. That is the *Enhanced Slow Query Log*. For me, it is the top reason why I run Percona Server everywhere I can. Just by using slow query log I can get such an enormous amount of information about the queries that I pretty much don't need any other statistics from MySQL.

But since this is not a query optimization course, let's now talk MariaDB.

MariaDB

MariaDB is a fork of MySQL created by Monty Widenius - the main author of the original MySQL. It is now adopted by many Linux distributions and while it's a good MySQL replacement in many aspects, in my view it's slightly over-appreciated.

If you have ever been to a Monty's talk, you know he is focusing his marketing efforts on making Oracle look bad while many of the things he has been manipulating never became true.

Anyways, politics aside, let's talk about two versions of MariaDB - MariaDB 5.5 and MariaDB 10.

1. **MariaDB 5.5** is using MySQL 5.5 at its core, plus it adds the XtraDB storage engine - a replacement for InnoDB adapted from Percona Server, and some code from MariaDB 5.3 which has a number of query optimizer improvements, multi-master replication, group commit fix and few other features. So if you are using 5.5 branch, with some specific query types you will see benefits by upgrading to MariaDB 5.5.

2. **MariaDB 10**, on the other hand, is very different from all of the other variants of MySQL in that it has started diverging from the upstream MySQL. It is what would be called a real fork (rather than a spoon). MariaDB used MySQL 5.6 at its foundation, but you should expect that MariaDB will not be backwards compatible with MySQL 5.6, 5.7 or any later MySQL versions.

Most noticeable Mariadb new features are:

- Parallel replication

- Multi-source replication
- Cassandra, Spider and TokuDB storage engines

The adaption of MariaDB is very limited so far and there aren't any good quality benchmarks comparing MySQL 5.6 or 5.7 to MariaDB 10, except for the ones produced by MariaDB (which are potentially biased), so it's too early to say if it's better to switch to MariaDB 10 or stay on MySQL. However given the amount of development resources MySQL is getting and the vast improvements in MySQL 5.7, my recommendation is to stay on MySQL for now.

Finally, let's discuss the last player in the field - WebScaleSQL.

WebScaleSQL

WebScaleSQL is a collaboration among engineers from several companies that face similar challenges in running MySQL at scale, namely *Alibaba*, *Facebook*, *Google*, *LinkedIn*, and *Twitter*.

It is a MySQL fork using MySQL 5.6 at its core and as far as I'm aware, there are no plans to use MySQL 5.7 as they consider 5.6 a good enough version to use as a basis. However it is also not planning to diverge from MySQL upstream, even though some features might make your data files not MySQL 5.6 backwards compatible. Oh and also, the coolest stuff is being back ported from MySQL 5.7 when needed.

It's very important to mention that, unlike other variants mentioned earlier, WebScaleSQL serves a very special purpose - it's not meant to be used as a general purpose MySQL server. Rather, it addresses very specific needs of the named companies running MySQL at scale. Well SCALE with the capital S. So unless you are running at least few dozen MySQL servers, you may wait a little on optimizing to that degree.

Here's just a few things that are different on WebScaleSQL - things that other variants don't and may not even have plans to have for quite obvious reasons:

- Ability to specify millisecond timeouts for various network operations
- Super read-only mode
- Ability to disable deadlock detection (not something you want to run on a typical transactional database server)
- Prefix index query optimization
- Absence of Performance Schema by default (Apparently Performance Schema may have a noticeable overhead even when not enabled)
- InnoDB flushing performance fixes, most of which are also available in Percona Server, MariaDB and now MySQL 5.7.

You can find a full list with extensive details in the following blog post by my former colleague Laurynas on Percona blog.

As you can see, unless you know exactly what you're doing, some features are potentially dangerous to run with. Therefore if you're in doubt, stick to Percona Server, MariaDB or stock MySQL 5.7 until you are absolutely sure WebScaleSQL is what you need.

The Essentials of MySQL Performance Tuning

Now that you have hopefully made a decision which MySQL distribution best suites your needs, let's talk about the process of MySQL performance tuning.

Here's a fact: *MySQL defaults are poor*.

Yes, you can start MySQL with no configuration and you can start using it for development right away. However, you can't just put MySQL with default configuration to production and expect that it will handle the increasing workload with ease - you have to prepare your server for that.

There's two good news though:

1. MySQL 5.7 has better defaults than ever before
2. MySQL is very easy to configure. It's just one configuration file you have to deal with, my.cnf, and it has one option per line, so the format is very convenient.

Enter MySQL Configuration File

The location of MySQL configuration file may be different across different operating systems and distributions, however on Linux it's typically either `/etc/my.cnf` (Redhat style) or `/etc/mysql/my.cnf` (Debian style). On Windows, you can put it in a number of locations, however I suggest you use the data directory and create my.cnf file there.

Open that file as we'll need it real soon. Or create it if it's not there yet.

By the way, if you already have a default configuration file (e.g. Debian/Ubuntu default files usually contain a lot of options), you'll probably want to do some merging in the editing process. Don't hesitate to remove any performance related options you're not sure about, or that are not mentioned in this e-book (but do keep a backup just in case).

Most Common MySQL Configuration Mistakes

Before we get to the key MySQL configuration variables, I'd like to quickly go over the most common mistakes I've seen being done when configuring MySQL.

Using Trial and Error approach

Here's how I've seen many customers approach MySQL configuration: they change a few things and check if it *feels* better.

And this is the biggest mistake that I see being done when approaching MySQL configuration. Problem is that by the time you're measuring how your application *feels*, situation may have changed already. Plus you have to consider the warm up time and disregard any overhead you see due to it. And most importantly, you can't actually base your optimization efforts on feelings – it's either faster, more scalable (and you understand why) or it's not.

After such a "tuning" session one may end up with a worse configuration than what they have started with. In fact, more often than not, that's exactly what I would find when a customer would first contact me.

Don't do it. Instead, understand exactly what it is that you're changing and only change it when you know that is what you need. Often running a micro-benchmark in a controlled environment to verify such change is a very good idea - that way you can confirm that it is working the way it's supposed to according to your understanding.

Most important variables to performance will be discussed here, so hopefully that will be a good start for you.

Now here's a few other common errors:

Using Google for performance advice

Never trust the first response you find on Google when searching for a performance advice, or value for your specific variable. A lot of the advice on the internet is very generic and often lacks context. For example, you may find a lot of configuration files on the internet that were used for benchmarks, however benchmarks often intentionally do certain things that should not be done on a production server, such as disabling double-write buffer, setting `innodb_thread_concurrency` to 0 and similar.

Also, a lot of the settings are hardware dependant, which is why it's even more so important to understand what it is that a specific variable is doing exactly.

Obsessing about fine-tuning the my.cnf

Don't get obsessed about fine-tuning the configuration – usually 10-15 variables will give you the most impact, and fine-tuning the variables is highly unlikely to have any additional benefits. It can do harm though. If you still have a performance problem even after you have applied all the recommendations (and gotten rid of everything that you shouldn't have touched in the first place), the problem is probably somewhere else – bad queries, lack of resources, etc.

Changing many things at once

When working with configuration, change only one thing at a time. Especially if you already have a solid configuration. Otherwise when things go bad, it maybe very hard or even impossible to figure out which setting could have caused the issue, so you will have to roll back all of the changes and then start one by one anyway.

If it's a new setup, or you were running with default configuration until now, feel free to go wild and implement all of the changes recommended here at once. Otherwise, change one thing and take some time to monitor the server after the change.

Not keeping my.cnf in sync with the changes you make

It's no secret that many things can now be changed online without even touching my.cnf. Even the innodb buffer pool size can be changed online in MySQL 5.7. That's very convenient, but. Make sure you update my.cnf after you are done with the changes or you will lose all these changes when MySQL is restarted and you'll have to start over.

Redundant entries in my.cnf

If you use the same variable twice, MySQL will not complain about it. In most cases, it will just use the last value found for the same variable, so be sure you don't add the same variable twice, otherwise you may end up not seeing the impact. Also note that a dash "-" and an underscore "_" can be used interchangeably, so `innodb-log-file-size` and `innodb_log_file_size` are both referring to the same server setting.

Multiplying buffer sizes

When you add more memory to the server, don't just multiply the size of all buffers in effect. Some buffers are local, some global. Some are storage engine related, some are server wide. In fact, there are very few variables that you need to increase in size as you add more memory. Yes, these are crucial to update, or you won't have the desired effect, but only these and no other. I will talk about these variables as we progress.

Using the wrong my.cnf section

While MySQL configuration file is simple, it's important to mention that it does have sections and these are important.

For example, there's such sections as `[mysql]`, `[client]`, `[mysqld_safe]`. And there's also a **`[mysqld]`** section, which is exactly the section you must use if you want server configuration to take effect. So all of the variables for server configuration should be placed after `[mysqld]`.

The only exception here is if you're using the `mysqld_multi` script, in which case you'll be working with several sections rather than just one.

Changing configuration online

Like I mentioned earlier, there is a way to change some parameters online. And in fact it's safe to try, even if you're not sure if you can change it online - MySQL will just tell you that the variable is read-only, meaning that you should be changing the my.cnf file instead and restart the server.

Here's how I would change innodb buffer pool size to 128MB online:

```
mysql> set global innodb_buffer_pool_size = 128*1024*1024;
ERROR 1238 (HY000): Variable 'innodb_buffer_pool_size' is a read only
variable
```

Except that this is MySQL 5.6 and not 5.7, which is why I can't do it online. How about innodb thread concurrency. Let's change that, but first - let's check the current value for it:

```
mysql> show global variables like 'innodb%';
+---------------------------+---------+
| Variable_name             | Value   |
+---------------------------+---------+
...
| innodb_thread_concurrency | 0       |
...
```

It's zero. Let's change it to 8:

```
mysql> set global innodb_thread_concurrency = 8;
Query OK, 0 rows affected (0.01 sec)

mysql> select @@global.innodb_thread_concurrency;
+------------------------------------+
| @@global.innodb_thread_concurrency |
+------------------------------------+
|                                  8 |
+------------------------------------+
1 row in set (0.00 sec)
```

This time we succeed. And in addition to the syntax of changing the configuration, you can see how you can access a value of a single global configuration variable. Which brings me to the next section.

Global -vs- Local scope

We have just seen a way to alter global MySQL configuration, but here's an interesting thing - quite often you don't need to update the global configuration just to get that single query work properly. In fact, very often it's better that you leave the global configuration untouched, because an optimization for one query may affect all of the other queries in the negative way.

For example, one of the things I recommend is to keep the `sort_buffer_size` at its default value, because otherwise a full buffer is allocated for any session where sorting is done and that may end up wasting a lot of memory and time allocating it.

But what do you do if you have a query that is constantly sorting large amounts of data and you can't add an efficient index, but you also want to avoid disk based sorts? Or what if you have a query that's consistently using `index_merge_intersection` optimization and you know it would be much better off not using it?

In that case, just set a session variable prior to running that query by issuing a statement similar to this one:

```
mysql> set optimizer_switch = 'index_merge_intersection=off';
Query OK, 0 rows affected (0.01 sec)
```

This will only change the value (and behaviour) for this session, so the day is saved.

17 Key MySQL variables

Now let's get down to business. I want to introduce you to 17 my.cnf settings that are key for optimal MySQL performance.

Some of these settings are self-evident but often forgotten, others require explanation so you know which setting is the right for you. And there are also a few settings that people tend to spend hours on, even though they aren't actually doing anything at all. I didn't list the settings in this last category here, but you will find them discussed in the comments in special my.cnf that I have prepared for your production needs. The file has short explanations and links to appropriate sections on my blog with more verbose explanations. If you haven't downloaded it yet, you can download it by going to:

http://www.speedemy.com/17

Basically, you can use this file as a starting point for your production server by tweaking one or two settings where appropriate.

Now here's the first MySQL setting we should talk about.

1. default_storage_engine – choose the right engine first

If all of your MySQL tables are using InnoDB and you don't need convincing that InnoDB is the way to go, you're all set with this one.

If you are unsure, however, bear we me. We have some ground to cover.

Storage engine what?

MySQL has supported pluggable storage engines since its inception over 20 years ago, but for a very long time MyISAM was the default storage engine and many people running MySQL didn't even know anything about the underlying storage engines. After all, MySQL was initially designed to be a practical database for small websites and many applications got into habit of using MyISAM storage engine explicitly.

This seemed like a good idea first, but here's the problem: MyISAM was *NOT* designed with highly concurrent workload, number of CPU cores and RAID arrays in mind. And it was never meant to be resilient either. So as websites kept attracting more traffic, they could no longer scale, because MySQL queries would spend seconds waiting on table level locks (the only locking mechanism that MyISAM supports). And they didn't want to be loosing their business critical data on each MySQL crash either.

Meet InnoDB

What many people don't know though is that virtually for as long as MySQL existed, MyISAM storage engine had a cousin called InnoDB. And highly concurrent workload, performance and resilience (also atomicity, consistency and isolation) was exactly where InnoDB shined.

Sure, it had a few bumps as it was growing up (most notably, scalability issues before version 5.0.30), but over the last 9 years InnoDB has been improved in all areas you can imagine, whereas MyISAM got virtually no attention at all.

Therefore, as of MySQL 5.5.5, InnoDB became the default storage engine and nowadays you will hardly find a sizeable MySQL installation that's still using MyISAM and not InnoDB.

Now before we go any further, let me show you how you can quickly get a count and a list of MyISAM tables on your system so you can start planning your migration.

Storage engines used by your database

Here's a really cool query that shows the storage engines you are using and a number of tables using each storage engine:

```
mysql> SELECT engine,
  count(*) as TABLES,
  concat(round(sum(table_rows)/1000000,2),'M') rows,
  concat(round(sum(data_length)/(1024*1024*1024),2),'G') DATA,
  concat(round(sum(index_length)/(1024*1024*1024),2),'G') idx,
  concat(round(sum(data_length+index_length)/
      (1024*1024*1024),2),'G') total_size,
  round(sum(index_length)/sum(data_length),2) idxfrac
 FROM information_schema.TABLES
WHERE table_schema not in
      ('mysql', 'performance_schema', 'information_schema')
GROUP BY engine
ORDER BY sum(data_length+index_length) DESC
LIMIT 10;
+--------+--------+---------+--------+--------+------------+---------+
| engine | TABLES | rows    | DATA   | idx    | total_size | idxfrac |
+--------+--------+---------+--------+--------+------------+---------+
| InnoDB |    181 | 457.58M | 92.34G | 54.58G | 146.92G    |    0.59 |
| MyISAM |     13 | 22.91M  | 7.85G  | 2.12G  | 9.97G      |    0.27 |
+--------+--------+---------+--------+--------+------------+---------+
2 rows in set (0.22 sec)
```

You can see that this particular customer had 13 MyISAM tables holding over 7GB worth of data combined. If you find yourself in a similar situation, don't worry, we'll fix that soon.

To get a list of all MyISAM tables sorted by size, just run this query:

```
SELECT
    concat(table_schema, '.', table_name) tbl,
    engine,
    concat(round(table_rows/1000000,2),'M') rows,
    concat(round(data_length/(1024*1024*1024),2),'G') DATA,
    concat(round(index_length/(1024*1024*1024),2),'G') idx,
    concat(round((data_length+index_length)/
        (1024*1024*1024),2),'G') total_size,
    round(index_length/data_length,2) idxfrac
 FROM information_schema.TABLES
WHERE table_schema not in
        ('mysql', 'performance_schema', 'information_schema')
  AND engine = 'MyISAM'
ORDER BY data_length+index_length DESC;
```

Bear in mind that changing `default-storage-engine` setting to InnoDB or upgrading MySQL doesn't automagically convert all your tables to InnoDB. Far from it. You actually have to go and convert tables one by one (or have a script do it).

Now before you go on and convert "just the big ones" to InnoDB, here's something really important:

Sometimes as part of migration to InnoDB, DBAs start with the large MyISAM tables to see if things will get better. And ***sometimes*** *it helps, but in many cases it doesn't. Here's the problem: if at least one table in a join is MyISAM, the entire query is using table level locks. So having even a small MyISAM table in a large join can be very bad for concurrency.*

Therefore when you're ready to convert, make sure to convert all MyISAM tables to InnoDB, not just the big ones.

Converting to InnoDB

I recommend that you hold off with conversion for now until you understand InnoDB configuration better and prepare your server for that, but when you're ready, you can run the following query to get a list of queries that will convert all tables in a given schema from *MyISAM* to *InnoDB*:

```
SET @DB_NAME = 'your_database';

SELECT CONCAT('ALTER TABLE `',
   table_name, '` ENGINE=InnoDB;') AS sql_statements
 FROM information_schema.tables AS tb
WHERE table_schema = @DB_NAME
  AND ENGINE = 'MyISAM'
  AND TABLE_TYPE = 'BASE TABLE'
ORDER BY table_name DESC;
```

Why my tables are created as MyISAM?

In the beginning I mentioned that many applications are using MyISAM explicitly. What I mean by that is that during the initialisation of the database, when all tables are created, CREATE TABLE statements often have ENGINE=MyISAM set at the end, so tables are created as MyISAM regardless of your `default-storage-engine setting`.

Thus it's a good idea to run the query above to check for any MyISAM tables every now and then. And, if you're using Percona Server, you may also want to set the following in my.cnf:

```
enforce_storage_engine = InnoDB
```

2. innodb_buffer_pool_size – get the best out of your memory

This is the most important InnoDB variable. Actually, if you're using InnoDB as your main storage engine, for you – it's THE most important variable for the entire MySQL server.

What is InnoDB Buffer Pool?

Computers use most of their memory to improve access to most commonly used data. This is known as caching and it is a very important part of computing, because accessing data on a disk can be 100 to 100,000 times slower, depending on the amount of data being accessed.

Just think of it, a report that takes 1 second to generate when all data is in memory could take over a day to generate if all data had to be read from disk every single time (assuming also random I/O).

MyISAM is using OS file system cache to cache the data that queries are reading over and over again. Whereas InnoDB uses a very different approach.

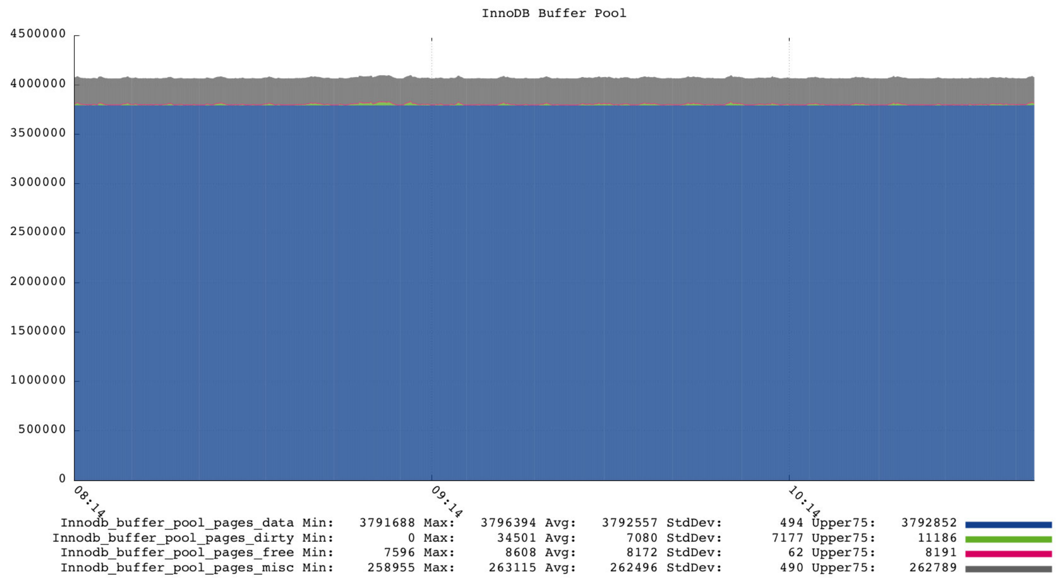
Instead of relying on OS to do the "right thing", InnoDB handles caching itself – within the InnoDB Buffer Pool – and you will soon learn how it works and why it was a good idea to implement it that way.

InnoDB Buffer Pool is more than just a cache

InnoDB buffer pool actually serves multiple purposes. It's used for:

- **Data caching** – this is definitely a big part of it
- **Indices caching** – yes, these share the same buffer pool
- **Buffering** – modified data (often called "dirty data") lives here before it's flushed
- **Storing internal structures** – some structures such as Adaptive Hash Index (we'll get to it) or row locks are also stored inside the InnoDB Buffer Pool

Here’s a very typical InnoDB Buffer Pool page distribution from a customer machine with `innodb-buffer-pool-size` set to 62G:



```
 Innodb_buffer_pool_pages_data Min:   3791688 Max:   3796394 Avg:   3792557 StdDev:       494 Upper75:   3792852
Innodb_buffer_pool_pages_dirty Min:         0 Max:     34501 Avg:      7080 StdDev:      7177 Upper75:     11186
 Innodb_buffer_pool_pages_free Min:      7596 Max:      8608 Avg:      8172 StdDev:        62 Upper75:      8191
 Innodb_buffer_pool_pages_misc Min:    258955 Max:    263115 Avg:    262496 StdDev:       490 Upper75:    262789
```

As you can see, Buffer Pool is mostly filled with regular InnoDB pages, but about 10% of it is used for other purposes.

Oh and in case you’re wondering what units are used in this graph, that’s InnoDB pages. A single page is typically 16 kilobytes in size, so you can multiply these numbers by 16,384 to get a sense of usage in bytes.

Sizing InnoDB Buffer Pool

So what should `innodb-buffer-pool-size` be set to? Well, it depends. And mostly, it depends on how important of a role is InnoDB playing.

DEDICATED SERVER

On a dedicated MySQL server running fully on InnoDB, as a rule of thumb, recommendation is to set the `innodb-buffer-pool-size` to 80% of the total available memory on the server.

Why not 90% or 100%?

Because other things need memory too:

- Every query needs at least few kilobytes of memory (and sometimes – few megabytes!)
- There’s various other internal MySQL structures and caches
- InnoDB has a number of structures using memory beyond the buffer pool (Dictionary cache, File system, Lock system and Page hash tables, etc.)
- There’s also some MySQL files that must be in OS cache (binary logs, relay logs, innodb transaction logs).
- Plus, you want to leave some room for the operating system memory structures.

By the way, this number is NOT pulled out of the hat – we've seen hundreds of systems large and small, and even on the servers with 512GB of RAM, we found 80% to be about the right size for sustainable operation.

If you see a lot of free memory, sure you can bump it up a bit (especially as MySQL 5.7 makes this much easier), but do not let MySQL consume all memory, or you will face problems. BIG problems. Namely, swapping.

Don't let your database server swap!

Swapping is the worst thing that can happen to a database server – it's much worse than having buffer pool size that's not large enough. One example how this may go wrong is InnoDB using a lock that it typically uses when accessing a page in memory (100ns access time) to access the page that is swapped out (10ms access time). This would cause all currently running queries to stall and as these things usually don't walk alone, you can guess where this is going…

SHARED SERVER

If your MySQL server shares resources with application, rules of thumb no longer work.

In such an environment it's much harder to find the right number. On the other hand, it's usually less so important to find the *right* size and a *good enough* number is often good enough in a shared environment.

In any case, first I check the actual size of the InnoDB tables. Chances are, you don't really need much memory. Remember this query from the earlier section:

```
SELECT engine,
  count(*) as TABLES,
  concat(round(sum(table_rows)/1000000,2),'M') rows,
  concat(round(sum(data_length)/(1024*1024*1024),2),'G') DATA,
  concat(round(sum(index_length)/(1024*1024*1024),2),'G') idx,
  concat(round(sum(data_length+index_length)/
      (1024*1024*1024),2),'G') total_size,
  round(sum(index_length)/sum(data_length),2) idxfrac
FROM information_schema.TABLES
WHERE table_schema not in
      ('mysql', 'performance_schema', 'information_schema')
GROUP BY engine
ORDER BY sum(data_length+index_length) DESC LIMIT 10;
```

This will give you an idea how much memory you'd need for InnoDB buffer pool if you wanted to cache the entire dataset. And note that in many cases you don't need that, you only want to cache your working set (actively used data).

If it all fits in, say, half the memory on the server, great – set it to the size of all InnoDB tables combined and forget it (for some time). If not, let me teach you a very simple trick to determine if the InnoDB buffer pool is well sized.

Using server command line, run the following (it will keep running until you hit ^C):

```
$ mysqladmin ext -ri1 | grep Innodb_buffer_pool_reads
| Innodb_buffer_pool_reads     | 1832098003   |
| Innodb_buffer_pool_reads     | 595          |
| Innodb_buffer_pool_reads     | 915          |
| Innodb_buffer_pool_reads     | 734          |
| Innodb_buffer_pool_reads     | 622          |
| Innodb_buffer_pool_reads     | 710          |
| Innodb_buffer_pool_reads     | 664          |
| Innodb_buffer_pool_reads     | 987          |
| Innodb_buffer_pool_reads     | 1287         |
| Innodb_buffer_pool_reads     | 967          |
| Innodb_buffer_pool_reads     | 1181         |
| Innodb_buffer_pool_reads     | 949          |
```

What you see here is the number of reads from disk into the buffer pool (per second). These numbers above are pretty darn high (luckily, this server has an IO device that can handle around 4000 random IO operations per second) and if this was an OLTP system, I would highly recommend to increase the innodb buffer pool size and add more memory to the server if needed.

If you don't have access to the command line, I suggest you get one as you're a lot more flexible there. But if you want a GUI alternative, MySQL Workbench is your friend. Under **PERFORMANCE**, open the **Dashboard** and you will see both "InnoDB buffer pool disk reads" and also "InnoDB Disk Reads" (most of the time they go hand in hand).

Ideally, there would be no reads from disk here. If that is the case, all data fits in the buffer pool, hence buffer pool is well sized. If the number is higher than the number of random I/O operations your disks can handle, then it's a good indication you need larger innodb buffer pool.

Anything in between shows that buffer pool is too small, but if you're happy with query response time (as they will be reading from disk occasionally), you may not need to do anything just yet.

Changing InnoDB Buffer Pool Size

Finally, here's how you actually change the `innodb-buffer-pool-size`.

If you're already on MySQL 5.7, you are extremely lucky, because that means you can change it online! Just run the following command as a SUPER user (i.e. root) and you're done:

```
mysql> SET GLOBAL innodb_buffer_pool_size = size_in_bytes;
```

Well not exactly done – you still need to change it in my.cnf file too, but at least you will not need to restart the server as the innodb buffer pool will be resized online with something along these lines in the error log:

```
[Note] InnoDB: Resizing buffer pool from 134217728 to 21474836480.
(unit=134217728)
[Note] InnoDB: disabled adaptive hash index.
[Note] InnoDB: buffer pool 0 : 159 chunks (1302369 blocks) were added.
[Note] InnoDB: buffer pool 0 : hash tables were resized.
[Note] InnoDB: Resized hash tables at lock_sys, adaptive hash index,
dictionary.
[Note] InnoDB: Completed to resize buffer pool from 134217728 to 21474836480.
[Note] InnoDB: Re-enabled adaptive hash index.
```

All earlier versions of MySQL do require a restart, so:

1. Set an appropriate `innodb_buffer_pool_size` in my.cnf
2. Restart MySQL server
3. Celebrate improved MySQL performance

Yay!

3. innodb_log_file_size – room for MySQL's redo log

This sets the size of the InnoDB's redo log files which, in MySQL world, are often called simply transaction logs. And right until MySQL 5.6.8 the default value of `innodb_log_file_size=5M` was the single biggest InnoDB performance killer. Starting with MySQL 5.6.8, the default was raised to 48M which, for many intensive systems, is still way too low.

Anyways, let's talk a little about the InnoDB Log Files so you have a better understanding of what it is, how MySQL is using it and how you would have to tune it.

What is InnoDB Redo Log?

Have you ever used an Undo or Redo function in a word processor, image editor or virtually any editor for that matter? I'm sure you have. Guess what, transactional databases have exactly the same thing! Well not exactly, but the principles are the same.

And just like it's important for you to always have the ability to go back a few steps in your editing process, so are the undo and redo functions important for a transactional database system. Why? Two reasons primarily:

1. Rolling back a transaction (that's the Undo)

2. Replaying committed transactions in case of a database crash (and that's Redo)

Here's how it works.

Undo

When you are using a transactional storage engine (let's stick to InnoDB for now), and you modify a record, the changes are not written to the data file directly.

First, they are written to a special file on disk called transaction log. And at the same time, they are also modified in memory – the InnoDB's buffer pool. This new InnoDB page that contains your modifications is now called *dirty*.

The original *unmodified* page is copied to a special area on disk called *rollback segment*.

So far so good?

Now, if someone (or something) interrupts a transaction with a ROLLBACK before it's committed, *Undo* operation needs to occur – your record has to be restored to its original state.

As the changes weren't written to data file yet, this is pretty trivial – InnoDB just removes the old copy of the page from the rollback segment, wipes the dirty page from memory and marks in a transaction log that the transaction was rolled back.

So there you go. Data file was never modified and it's good that it wasn't, because you have cancelled any changes you made before even issuing a random write operation to flush that dirty page to disk.

Redo

When you COMMIT the transaction, however, and InnoDB approves your commit (i.e. it returns from the COMMIT call), changes are ready to be written to the actual data files.

You'd think they are written to data files immediately at this point, but that's not what happens. Why? Because doing so would be very inefficient. Instead, the changes are only written to the transaction log (this is very efficient sequential activity called Redo logging), while the modified record still lives in memory – in the InnoDB buffer pool as a dirty page, for as long as time comes to flush it.

So what happens now if MySQL crashes at this point?

Well, if InnoDB had no redo log and it only kept dirty pages in memory – all of the committed transactions that were not flushed to disk yet would be gone forever. Quite a disaster if you consider that one of these transactions may have been your salary transfer from company account to yours.

Luckily, the changes ARE always written to the transaction log (a.k.a. REDO log) before the operations return from the call, so all InnoDB needs to do is find the last checkpoint in the Redo log (position that's been synchronised to disk) and Redo all of the modifications by re-reading the "to-be-modified" data from disk and re-running the same changes.

Easy peasy, right?

Well, right. But only on the surface. Underneath, there's a lot of really complex stuff happening that you probably don't want to know about right now. We can talk about it sometime later.

Size matters

One thing you may want to know about though is how to size the `innodb_log_file_size` properly. The rules are actually pretty simple:

- Small log files make writes slower and crash recovery faster
- Large log files make writes faster and crash recovery slower

Writes become slow with small Redo log because the transaction logs act as a buffer for writes. And if you have a lot of writes, MySQL may be unable to flush the data fast enough, so write performance degrades.

Large log files, on the other hand, give you a lot of room that can be used before flushing needs to happen. That in turn allows InnoDB to fill the pages more fully (for example, when you modify few records that are on the same page, or in fact, modify same record several times) and also, in case of Magnetic drives, flush the dirty pages in a more sequential order.

As for the crash recovery – larger Redo log files means more data to be read and more changes to be redone before the server can start, which is why it makes crash recovery slower.

Sizing the Redo log

Finally, let's talk how you can figure out the right size for the Redo logs.

Luckily, you don't have to come up with a size that's exactly right. Here's a rule of thumb that we found to work like magic:

Rule of Thumb: *Check that total size of your Redo logs fits in 1-2h worth of writes during your busy period.*

How do you know how much InnoDB is writing? Here's one way to do it:

```
mysql> pager grep seq
mysql> show engine innodb status\G
        select sleep(60); show engine innodb status\G
Log sequence number 1777308180429
...
Log sequence number 1777354541591

mysql> nopager
mysql> select (1777354541591-1777308180429)*60/1024/1024;
```

```
+-----------------+
| (17773.../1024 |
+-----------------+
|  2652.80696869 |
+-----------------+
1 row in set (0.00 sec)
```

In this case, based on 60s sample, InnoDB is writing 2.6GB per hour. So, if `innodb_log_files_in_group` was not modified (and by default it is 2 - minimum number of Redo log files that InnoDB needs), then by setting `innodb_log_file_size` to, say, 2560M, you will have exactly 5GB of Redo log storage across the two Redo log files.

Changing the Redo log size

How hard it will be to change the `innodb_log_file_size` AND how large you can set it to, depends greatly on the version of MySQL (or Percona, or MariaDB) server that you are using.

Specifically, if you are using version prior to 5.6, you can't simply change the variable and expect that the server will restart. In fact it will stop, but won't start.

So here's how you have to do it – I have described the process on Percona blog few years ago. Basically, it's:

1. Change `innodb_log_file_size` in my.cnf
2. Stop MySQL server
3. Ensure MySQL had a clean shutdown (mysql log is your friend)
4. Remove old log files, usually by running the following command: `rm -f /var/lib/mysql/ib_logfile*`
5. Start MySQL server – it should take a bit longer to start because it is going to be creating new transaction log files.

Final thing you should be aware of is that until quite recently (i.e. until MySQL version 5.6.2), the total Redo log size (across all Redo log files) was limited to 4GB, which was quite a significant performance bottle-neck for write-intensive SSD backed MySQL servers. (Percona Server, on the other hand, supports 512GB since like Percona Server five ou something) In other words, before you set `innodb_log_file_size` to 2G or more, check if the version of MySQL you are running actually supports it.

4. innodb_flush_log_at_trx_commit – durable or not? That is the question!

By default, `innodb_flush_log_at_trx_commit` is set to 1 which instructs InnoDB to flush AND sync after EVERY transaction commit. And if you are using autocommit (by default you are), then every single INSERT, UPDATE or DELETE statement is a transaction commit.

Sync is an expensive operation (especially when you don't have a non-volatile write-back cache) as it involves the actual synchronous physical write to the disk, so whenever possible, I would recommend to avoid using this default configuration.

Two alternative values for `innodb_flush_log_at_trx_commit` are 0 and 2:

- 0 means FLUSH to disk, but DO NOT SYNC (no actual IO is performed on commit),
- 2 means DON'T FLUSH and DON'T SYNC (again no actual IO is performed on commit).

So if you have it set to 0 or 2, sync is performed once a second instead. And the obvious disadvantage is that you may lose last second worth of committed data. Yes, you read that right – those transactions would have committed, but if server loses power, those changes never happened.

Obviously for financial institutions, such as banks. it's a huge no-go. Most websites, however, can (and do) run with `innodb_flush_log_at_trx_commit=0|2` and have no issues even if the server crashes eventually. After all, just few years ago many websites were using MyISAM, which would lose up to 30s worth of written data in case of a crash. (not to mention the crazy slow table repair process).

Finally, what's the practical difference between 0 and 2? Well, performance wise the difference is negligible really, because a flush to OS cache is cheap (read fast). So it kind of makes sense to have it set to 0, in which case if MySQL (but not the whole machine) crashes, you do NOT lose any data as it will be in OS cache and synced to disk from there eventually.

BTW, if you prefer durability over performance and have `innodb_flush_log_at_trx_commit` set to 1, let me draw your attention to the next variable which is closely related:

5. sync_binlog – that's for durable binlog

A lot has been written about `sync_binlog` and it's relationship with `innodb_flush_log_at_trx_commit`, but let me simplify it for you for now:

a. If your MySQL server has no slaves and you don't do backups, set `sync_binlog=0` and be happy with a good performance

b. If you do have slaves and you do backups, but you don't mind losing a few events from the binary logs in case of a master crash, you may still want to use `sync_binlog=0` for the sake of a better performance.

c. If you do have slaves and/or backups, and you do care about slaves consistency and/or point in time recovery (ability to restore your database to a specific point in time by using your latest consistent backup and binary logs) and you are also running `innodb_flush_log_at_trx_commit=1`, then you should seriously consider using `sync_binlog=1`.

Problem is of course that `sync_binlog=1` has a pretty significant price – now every single transaction is again synced to disk – to the binary logs.

You'd think why not do both sync operations at once, and you'd be right – new versions of MySQL (both 5.6 and 5.7, MariaDB 5.5 and Percona Server 5.6+) already have a properly working binlog group commit, in which case the price for running with `sync_binlog=1` gets really small (assuming you are running with `innodb_flush_log_at_trx_commit=1` anyway), but older versions of MySQL have a really significant performance penalty, so make sure you watch your disk writes.

So far so good? Good. Next.

6. innodb_flush_method – your chance to avoid double buffering

Set `innodb_flush_method` to `O_DIRECT` to avoid double-buffering. The only case you should NOT use `O_DIRECT` is when it is not supported by your operating system. But if you're on Linux, use `O_DIRECT` to enable direct IO.

Direct I/O means that InnoDB's read and write calls are bypassing OS cache and are going directly to the I/O scheduler to get sent to the disks.

Without direct IO, double-buffering happens because all database changes are first written to OS cache and then they are synced to disk – so you end up with the same data in InnoDB buffer pool AND in OS cache. Yes, that means in a write-intensive environment you could be losing up to almost 50% of memory, especially if your buffer pool is capped at 50% of total memory. And if not, server could end up swapping due to high pressure on the OS cache.

In other words, do set `innodb_flush_method=O_DIRECT`, please.

7. innodb_buffer_pool_instances – reduce global mutex contention

MySQL 5.5 introduced buffer pool instances as a means to reduce internal locking contention and improve MySQL scalability. In version 5.5 this was known to improve the throughput to some degree only, however MySQL 5.6 was a big step up, so while in MySQL

5.5 you may want to be more conservative and have `innodb_buffer_pool_instances=4`, on MySQL 5.6 and 5.7 feel free to go with 8-16 buffer pool instances.

Your mileage may vary of course, but with most workloads, that should give you best results.

Oh and obviously do not expect this to make any difference to a single query response time. The difference will only show with highly concurrent workloads i.e. those with many threads doing many things in MySQL at the same time.

8. innodb_thread_concurrency – have better control over your threads

You may hear very often that you should just set `innodb_thread_concurrency=0` and forget it. Well, that's only true if you have a light or moderate workload. However, if you're approaching the saturation point of your CPU or IO subsystem, and especially if you have occasional spikes when the system needs to operate properly when overloaded, then I would highly recommend to tackle `innodb_thread_concurrency`.

Here's the thing – InnoDB has a way to control how many threads are executing in parallel – let's call it a concurrency control mechanism. And for the most part it is controlled by `innodb_thread_concurrency`. If you set it to zero, concurrency control is off, therefore InnoDB will be processing all requests immediately as they come in (and as many as it needs to).

This is fine if you have 32 CPU cores and 4 requests. But imagine you have 4 CPU cores and 32 CPU intensive requests – if you let the 32 requests run in parallel, you're asking for trouble. Because these 32 requests will only have 4 CPU cores, they will obviously be at least 8 times slower than usually (in reality, more than 8 times slower of course), but each of those requests will have its own external and internal locks which leaves great opportunities for all the queries to pile up.

To solve that, you can control how many threads InnoDB allows to execute at any given point in time. Other threads wait in a lined up queue. But it's NOT a simple First In, First Out queue (a.k.a. FIFO), it's much more interesting that that. Here's how it works.

A thread entering an InnoDB queue is given a certain number of tickets, equal to the value of `innodb_concurrency_tickets` - 500 by default on MySQL 5.5 or earlier, 5000 starting MySQL 5.6.

Then it waits in a queue until a slot becomes available for it. Finally, it starts executing. And every time the thread does a certain operation, it loses one ticket and goes through a check for the number of remaining tickets. For example, a thread would lose a ticket every time a row is read, inserted or updated (and it would lose 300 tickets for a single read of 300 rows).

Now here's where it gets interesting - as soon as the thread runs out of tickets, i.e. the ticket count decreases to 0, the thread is placed at the back of the queue and needs to wait until the slot becomes available again, assuming the current number of running threads is above the value of `innodb_thread_concurrency`. That way the number of queries that are executing at any given point is limited, yet long running queries don't prevent quick queries from entering a queue for a very long time.

So you see we already have two variables:

- `innodb_thread_concurrency`, which controls how many threads are allowed to run at the same time,
- `innodb_concurrency_tickets`, which controls how many tickets a thread is given every time it re-enters the queue.

There's one more variable related to this - `innodb_thread_sleep_delay`, which has a default value of 10,000us. This sets how long innodb threads sleep before joining the InnoDB queue, thus controlling the concurrency to a certain degree.

If you want to change the value for `innodb_thread_concurrency`, you can do it online by running the following command:

```
SET global innodb_thread_concurrency=X;
```

For most workloads and servers, 8 is a good start and then you should only increase it gradually if you see that the server keeps hitting that limit while hardware is under-utilised. To see where the Threads stand at any given moment, run `show engine innodb status\G` and look for a similar line in the ROW OPERATIONS section:

```
22 queries inside InnoDB, 104 queries in queue
```

Both `innodb_concurrency_tickets` and `innodb_thread_sleep_delay` can also be changed online if you'd like to try out different configuration for these.

9. skip_name_resolve – do skip that reverse IP lookup

This is a funny one, but I couldn't not mention it as it's still quite common to see it not being used.

Essentially, you want to add `skip_name_resolve` to avoid DNS resolution on connection. Most of the time you will feel no impact when you change this, because most of the time DNS servers work, they work well and they also tend to cache results.

But when a DNS server will fail, it could be really time-consuming to figure out what are those "unauthenticated connections" doing on your server and why things all of the sudden seem slow…

So. Don't wait until this happens to you. Add this variable now and get rid of any hostname based grants. The only exception here is if you're using hosts file based name resolution. Or if your DNS servers will never fail (ha ha).

10. innodb_io_capacity, innodb_io_capacity_max – cap InnoDB IO usage

Here's what these two IO capacity settings control in a nutshell:

- `innodb_io_capacity` controls how many write IO requests per second (IOPS) will MySQL do when flushing the dirty data,
- `innodb_io_capacity_max` controls how many write IOPS will MySQL do flushing the dirty data when it's under stress.

So, first of all, this has nothing to do with foreground reads – ones performed by SELECT queries. For reads, MySQL will do the maximum number of IOPS possible to return the result as soon as possible. As for writes, MySQL has background fuzzy flushing cycles and during each cycle it checks how much data needs to be flushed. Then, it will use no more than `innodb_io_capacity` IOPS to do the flushing. That also includes change buffer merges (change buffer is a where secondary keys dirty pages are stored until they are flushed to disk).

Second, I need to clarify what "under stress" means. This, what MySQL calls an "emergency", is a situation when MySQL is behind with flushing and it needs to flush some data in order to allow for new writes to come in. Then, MySQL will use the `innodb_io_capacity_max` amount of IOPS.

Now here's the $100 question: so what do you set these to?

Best solution – measure random write throughput of your storage and set `innodb_io_capacity_max` to the maximum you could achieve, and `innodb_io_capacity` to 50-75% of it, especially if your system is write-intensive.

Often you can predict how many IOPS your system can do, especially if it has magnetic drives. For example, 8 15k disks in RAID10 can do about 1000 random write IOPS, so you could have `innodb_io_capacity=600` and `innodb_io_capacity_max=1000`. Many cheap enterprise SSDs can do 4,000-10,000 IOPS.

Nothing bad will happen if you don't make it perfect. But. Beware that the default values of 200 and 400 respectively could be limiting your write throughput and consequently you may have occasional stalls for the flushing activity to catch up. If that's the case – you are either saturating your disks, or you don't have a high enough values for these variables.

11. innodb_stats_on_metadata – turn them OFF!

If you're running MySQL 5.6 or 5.7 and you didn't change the default `innodb_stats_on_metadata` value, you're all set.

On MySQL 5.5 or 5.1, however, I highly recommend to turn this OFF – that way commands like `show table status` and queries against `INFORMATION_SCHEMA` will be instantaneous instead of taking seconds to run and causing additional disk IO.

My former colleague Stephane Combaudon from Percona has written a very good blog post on this.

Oh and note that starting with 5.1.32, this is a dynamic variable, so you don't need to restart MySQL just to disable that.

12. innodb_buffer_pool_dump_at_shutdown & innodb_buffer_pool_load_at_startup

It may seem the two variables `innodb_buffer_pool_dump_at_shutdown` and `innodb_buffer_pool_load_at_startup` are not performance related, but if you are occasionally restarting your MySQL server (e.g. to apply configuration changes), they are.

When both are enabled, the contents of MySQL buffer pool (more specifically, cached pages) are saved into a file upon shutdown. And when you start MySQL, it starts a background thread that loads back the contents of buffer pool and improves the warm-up speed that way up to 3-5 times.

Couple of things:

First, it doesn't actually copy the contents of the buffer pool into a file upon shutdown, it merely copies the `tablespace IDs` and `page IDs` – enough information to locate the page on disk. Then, it can load those pages really fast with large sequential reads instead of hundreds of thousands of small random reads.

Second, loading of the contents on startup happens in the background, hence MySQL doesn't wait until the buffer pool contents are loaded and starts serving requests immediately (so it's not as intrusive as it may seem).

Third (I know, I know, three isn't really a couple anymore), starting with MySQL 5.7.7, only 25% of least recently used buffer pool pages are dumped on shutdown, but you have a total control over that – use `innodb_buffer_pool_dump_pct` to control how many pages (expressed in percents) would you like to be dumped (and restored). I vote 75-100.

And fourth (sorry!), while MySQL only supports this since MySQL 5.6, in Percona Server you had this for quite a while now – so if you want to stay on version 5.1 or 5.5, or if you are already using Percona Server, check this out (version 5.1 link).

13. innodb_adaptive_hash_index_parts – split the AHI mutex

If you're running a lot of SELECT queries (and everything else is perfectly in tune), then *Adaptive Hash Index* is likely going to be your next bottle-neck.

Adaptive Hash Indexes are dynamic indexes maintained internally by InnoDB that improve performance for your most commonly used query patterns. This feature can be turned off with a server restart, but by default it is ON in all versions of MySQL.

While it's quite a complex technology (more on it here), in most cases it gives a nice boost to many types of queries. That is, until you have too many queries hitting the database, at which point they start spending too much time waiting on the AHI locks and latches.

If you're on MySQL 5.7, you won't have any issues with that – `innodb_adaptive_hash_index_parts` variable is there and it's set to 8 by default, so the adaptive hash index is split into 8 partitions, therefore there's no global mutex and you're good to go.

All MySQL versions before 5.7, unfortunately, have no control over the number of AHI partitions. In other words, there's one global mutex protecting the AHI and with many select queries you're constantly hitting this wall.

So if you're running 5.5 or 5.6, and you have thousands of select queries per second, the easiest solution would be to switch to same version of Percona Server and enable AHI partitions. The variable in Percona Server is called `innodb_adaptive_hash_index_partitions`.

14. query_cache_type – ON? OFF? ON DEMAND?

A lot of people think Query cache is great and you should definitely use it. Well, sometimes it's true – sometimes it is useful. But it's only useful when you have a relatively light workload and especially if the workload is pretty much read-only with very few or virtually no writes.

If that's your workload, set `query_cache_type=ON` and `query_cache_size=256M` and you're done. Note, however, **you should never set the query cache size any higher**, or you will likely run into serious server stall issues due to query cache invalidation. I've seen this happen too many times and until someone figures out a way to split a global query cache mutex, this will not go away.

If you have an intensive workload, however, then you should not only set the `query_cache_size=0` but also it's very important that you set `query_cache_type=OFF` and restart the server when you do that – this way MySQL will stop using query cache mutex for all queries – even those that wouldn't use the query cache anyway.

BTW, this works with MySQL 5.5 or newer, you can't really disable the query cache mutex in the earlier versions of MySQL. In Percona Server, you can also disable it in version 5.1.

So if you're still using the query cache and you shouldn't, make these changes now because your queries are likely already suffering due to query cache mutex contention.

15. innodb_checksum_algorithm – the secret hardware acceleration trick

Most mainstream CPUs nowadays support native crc32 instructions and MySQL can finally make use of that to improve the speed of calculating the InnoDB checksums significantly.

To enable that, set the `innodb_checksum_algorithm=crc32`. This is available since MySQL 5.6. Starting with MySQL 5.7.7, `innodb_checksum_algorithm=crc32` is set by default.

Checksums are calculated every single time a page (or log entry) is read or written, so this is definitely YUUGE! (right, Donald?)

Oh and BTW, this is totally safe to change on a server that already has tables created with checksum type "innodb". In fact, you can change it dynamically, online, while the server is running (There, I said it three times, now it's fully redundant).

16. table_open_cache_instances – use it

Starting with MySQL 5.6.6, table cache can be split into multiple partitions and if you're running MySQL 5.6 or newer, you should definitely do that.

Table cache is where the list of currently opened tables is stored and the mutex is locked whenever a table is opened or closed (and, in fact, in a number of other cases) – even if that's an implicit temporary table.

And using multiple partitions definitely reduces potential contention here. I've seen this `LOCK_open` mutex issue in the wild too many times and it causes a great deal of trouble, especially with slower disk I/O or file systems.

Starting with MySQL 5.7.8, `table_open_cache_instances=16` is the default configuration, and I'd say it is definitely a good starting point both on MySQL 5.6 and 5.7.

17. innodb_read_io_threads & innodb_write_io_threads – last and, yes, least

I've placed this last because it's really not as important as it may seem.

First of all, your MySQL is likely already using Asynchronous IO (AIO) which on Linux is supported since MySQL 5.5 (on Windows, it's been supported for a while now).

Second, both `innodb_read_io_threads` and `innodb_write_io_threads` are only used for the background activity, such as checkpointing (flushing dirty pages to disk), change buffer merge operations and sometimes, read-ahead activity.

So, while it's not a key variable to tune, aligning it to the number of bearing disks is still a good idea. So for example on RAID10 with 8 disks, you may want to have `innodb_read_io_threads=8` and `innodb_write_io_threads=4`. If you have an SSD, well then just have it around 16-32, but do not expect much performance difference, unless your server is extremely write-heavy and disk IO is the bottle-neck.

The End (Almost)

We’ve now covered all the different MySQL distributions, right and wrong ways to approach MySQL configuration and the key variables to look at. It’s quite amazing how much you can improve things with small changes over the vanilla MySQL configuration, isn’t it? But you know what - we haven only made first performance optimization steps.

I mean, improving MySQL configuration will definitely impact its performance greatly and it is surely the foundation of a well performing MySQL server.

However, if you have a server that is otherwise tuned pretty well, you shouldn’t expect a few configuration changes to improve things drastically. Most of the time, the devil is in the queries.

Occasionally I am running free webinars on query optimization and similar topics where you can learn more in-depth about MySQL Performance Optimization and get your questions answered. You can see a list of upcoming webinars and sign up at speedemy.com/webinars.

Also do not forget to download the my.cnf I have mentioned if you have’t done so yet.

Now let’s move to the final - bonus - part of this ebook.

Bonus: Understanding MySQL Status Counters

In the early days of MySQL, MySQL status was pretty much the only way to look at MySQL performance, therefore it was a pretty common practice amongst MySQL consultants and support engineers to look at MySQL status to understand what's happening with the server.

And even though now there are better ways to look at specific issues, for example analyze query performance using the extended slow query log in Percona Server or `PERFORMANCE_SCHEMA` starting with MySQL 5.6, MySQL status is still a good way to look at the general metrics of the server. However, it's really important to look at it correctly, which is what I want to show you now.

In this quick lesson:

- I am going to show you how to look at mysql status the right way
- We will identify few key status variables to look at
- And you will learn what free tools you can use to make monitoring MySQL status much easier

How to Look At MySQL Status

First, let me show you what is the wrong way to look at MySQL status. Run `show global status;` in MySQL console or `mysqladmin ext` in the command line and you will be overwhelmed with huge numbers.

These commands will show you the current count for each status variable. Oh and bear in mind that not all status variables are counters, some are gauges, displaying current value. You will have to look at these separately.

So, the right way to look at status variables is over a certain period of time. For example, if you run the following command, you will get total count for each counter now, followed by a total counter 60 seconds later:

```
$ mysqladmin ext -i60 -c3
```

`-c` specifies how many iterations to show, hence in this case counters will be displayed 3 times.

Now this is still not very useful, because for most variables you are interested in delta rather than the totals and I will show you two ways to get the relative numbers:

- Add `-r` to mysqladmin and that will show you delta right away:

  ```
  $ mysqladmin ext -ri60 -c3
  ```

- Another, more elegant way to get the same data is by using pt-mext:

  ```
  $ pt-mext -r -- mysqladmin ext -i60 -c3
  ```

Although I should admit, I never use it that way. Instead, I would first write `mysqladmin` output into a file and then I would read that file with `pt-mext`:

```
$ mysqladmin ext -i60 -c3 > mysqladmin.txt
$ pt-mext -r -- cat mysqladmin.txt
```

This allows me to get a different cut for the same data in `mysqladmin.txt`. For example, I can collect 100 samples every 1s and then analyze the file with grep to look at gauges, and use pt-mext to analyze counters.

Key MySQL Status Variables To Look At

Now let's take a look at a few variables that are more interesting. Beware that we're focusing on performance only, so we will not look at all 400 counters.

Com_* Counters

I always look at these to understand the workload that server is currently dealing with:

```
Com_begin
Com_commit
Com_delete
Com_insert
Com_select
Com_update
```

This will give you an idea of how many commands per second the server is running at any given point (so you can correlate these to any other counters)

Temporary tables

It's always a good idea to see how many temporary tables you have on disk -vs- total number of temporary tables, so you have to look at the following variables:

```
Created_tmp_disk_tables
Created_tmp_tables
```

Note that very often the reason on-disk temporary tables are created is not a too small setting for `tmp_table_size` or `max_heap_table_size`, rather it's variable size columns that are used in the queries, e.g. text or blob columns that can't be used in the fixed size temporary tables. For these, temporary table on disk will be used even if the temporary table will contain a single record.

Otherwise, many temporary tables can be avoided by reviewing the queries that create them and fixing inefficient execution plans by either indexes or changes in the queries.

Handler_* Counters

These are internal operation counters, often accounting for every single record access. The most interesting ones you want to look at are:

```
Handler_read_first
Handler_read_key
Handler_read_next
Handler_read_prev
Handler_read_rnd_next
```

For what they stand for, please have a look at MySQL manual over here. However, these counters are much more interesting to look at when you're analysing a behaviour of a specific query. For that, you want to use session status counters in the following fashion:

```
mysql> flush status;
mysql> RUN YOUR QUERY;
mysql> show status;
```

That will give you a much better understanding of what a specific query is doing internally and potentially how you can optimize it.

Innodb_* Counters

InnoDB has plenty of counters to look at. Additionally, you can get a whole lot deeper with `show engine innodb status\G` command. Here's the most typical things I would look at and why:

- `Innodb_buffer_pool_pages_flushed` - number of pages flushed from buffer pool - good way to monitor page flush activity.
- `Innodb_buffer_pool_reads` - number of Disk IO calls to read into the buffer pool - see how close that is to how many random reads can your disks actually deliver.
- `Innodb_data_fsyncs` - number of `fsync()` calls executed - see if it's not too high for your hardware.
- `Innodb_data_pending_*` - gauges showing a number of pending fsync, read or write calls - could potentially show saturated IO resources.
- `Innodb_data_reads/writes` - number of random read/write disk IO operations for data files specifically.
- `Innodb_history_list_length` - gauge showing a number of transactions that haven't been cleaned up after.
- `Innodb_ibuf_merges` - number of insert buffer merge operations. High numbers here could explain intense IO spikes.

- `Innodb_log_waits` - number of times log buffer was too small. Good indicator `innodb_log_buffer_size` needs a raise.
- `Innodb_lsn_current` - number of bytes written to the transaction log. Helps you tune innodb redo log files.
- `Innodb_mutex_os_waits` - if this is high, you could be having internal mutex contention.
- `Innodb_rows_*` - helps you understand internal activity - number of rows read, inserted, deleted or updated.
- `Innodb_row_lock_time*` - shows how much time is spent on logical locks.

Note that some of these variables are only available in Percona Server. If you can't find it on your server, check `show engine innodb status\G` - it's not as convenient, but that's where you will find everything about InnoDB operation.

Opened_* stuff

Check these to understand if your file caches are sized decently. Ideally, `Opened_tables` and `Opened_table_definitions` should not be increasing much or at all.

Query cache

You can monitor the query cache activity by looking at `Qcache_*` counters. Most important is to compare `Qcache_hits` to `Com_select` to understand how many SELECT queries out of total are served from the query cache. Although `Qcache_lowmem_prunes` and `Qcache_inserts` can be just as interesting to understand. Note however that even if you do see lowmem prunes, do not increase the size of the query cache above 256MB.

Select counters

There's a few interesting `Select_*` counters to look at, here's what they stand for:

- `Select_full_join` - shows a number of queries that made a table scan during a join (even if join buffer was used). These could be pretty bad queries in some cases, but sometimes it's also tiny tables that are very fast to join anyway.
- `Select_full_range_join` - number of joins that used a range search on a reference table. These aren't common, but they aren't always harmful either.
- `Select_range` - this is a very common range access pattern used by queries. FYI only.
- `Select_range_check` - number of joins without keys, with additional key usage check each time row is read. Good indicator of bad indexing. Rarely seen in the wild though.

- `Select_scan` - table scan on the first (or the only) table in the join. Also shows bad indexing.

So, if either `Select_full_join`, `Select_range_check` or `Select_scan` is relatively high, chances are you have a pretty bad indexing and I highly recommend doing a query optimization round.

Threads_* counters

One of these counters is the main indicator of any performance issues in MySQL and I use it in my troubleshooting practice all the time. Here's what the Threads counters show:

- `Threads_cached` - gauge, showing a current number of cached threads. Not super interesting.
- `Threads_connected` - number of threads currently connected to the server. A lot of these could be sleeping threads and sleeping threads are very cheap to have, so don't mind these too much either.
- `Threads_created` - counter, showing a number of threads created because the number of cached threads wasn't enough. If this is high, consider increasing the `thread_cache_size`.
- `Threads_running` - finally, this is the most interesting counter of them all. It is a gauge showing a number of threads that are currently executing inside MySQL. They could be doing anything from waiting in the InnoDB queue to committing data to disk, but the key thing is that if this number is occasionally abnormally high (i.e. larger than the number of CPU cores or disk spindles that you have), you are most likely having pretty serious MySQL Performance issues.

See the following blog post to learn how you can deal with those.

Free Tools To Make Monitoring MySQL Easier

Finally, let's look at some free tools that will make your life better when it comes to MySQL Status monitoring.

Cacti && Zabbix

If you're not using either of two, I would highly recommend to give it a try:

- Cacti
- Zabbix

However, they are not useful for MySQL monitoring in itself, you need to add some additional templates. Percona has a really nice package just for that:

- Percona Monitoring Plugins for MySQL

pt-mext

I have already mentioned pt-mext, what I did not mention however, is how you can get it. You can either install Percona Toolkit which is now included in a number of distributions too, or you can download pt-mext as a standalone command line utility which is what I often do when doing a consulting gig:

```
$ wget percona.com/get/pt-mext
$ chmod +x pt-mext
$ ./pt-mext -r -- mysqladmin ext -i1 -c5
```

innotop

It's a top-like command line tool to monitor MySQL Status with some emphasis on InnoDB. It's a really comprehensive tool and I highly recommend to check it out.

What now

Did I answer all of your questions relating to MySQL configuration? If not, leave a comment here so we can have a discussion.

Otherwise, if you're done with MySQL configuration file, next logical step in the optimization process would be query optimization. And it's a whole new discipline to learn.

I am currently working on a MySQL performance optimization course, so if you'd like to learn when it's finished, sign up for a newsletter on speedemy.com and I will let you know as soon as it becomes available.