Configuring MySQL For Performance

VividCortex

September 15, 2014

Abstract

With hundreds of settings, MySQL configuration can be overwhelming. In recent years, this has become even more true, as MySQL has added features such as a more sophisticated query optimizer and multiple InnoDB buffer pools. And the default configuration samples shipped with the server are designed for avoidance of harm, not optimal performance.

Fortunately, the vast majority of the benefit comes from setting just a few very important options. Although special cases may benefit from setting more obscure options, the suggestions in this ebook will serve you well most of the time.

The most exhaustive information available on these topics is the official MySQL documentation, with a lot of useful information also available in High Performance MySQL, Third Edition.
Note that this guide is not comprehensive and does not cover topics such as hardware selection. These are very important and complex subjects.

Table of Contents

Understanding MySQL Configuration

MySQL reads its configuration at startup from commandline options, as well as configuration files (typically `/etc/my.cnf`). At runtime, many variables can be changed with `SET` commands executed as SQL statements. MySQL doesn't re-read configuration files at runtime.

In addition to changing the server's configuration, you can also inspect it by querying the server. MySQL's configuration variables are exposed in the `SHOW VARIABLES` command and in the `INFORMATION_SCHEMA` , and in predefined variables you can access by name (these are typically read-write).

"

You'll need to remember to update the configuration file for any changes you make while the server is running.

The variables can be identified in different ways depending on where you're seeing them. For example, in the configuration files and commandline context, they can be abbreviated, and underscores can be interchanged freely with hyphens. We suggest avoiding messy, ambiguous syntax such as mixing underscores and abbreviations. These things can come back to bite you when you least expect it, such as during a server upgrade or an emergency firefighting session when a typo is easy to overlook.

There are a few exceptions and special cases. For example, there are some settings that have different names in the different places you'll see them. Beyond noting that this is the case for just a few settings, we won't go into details about this.

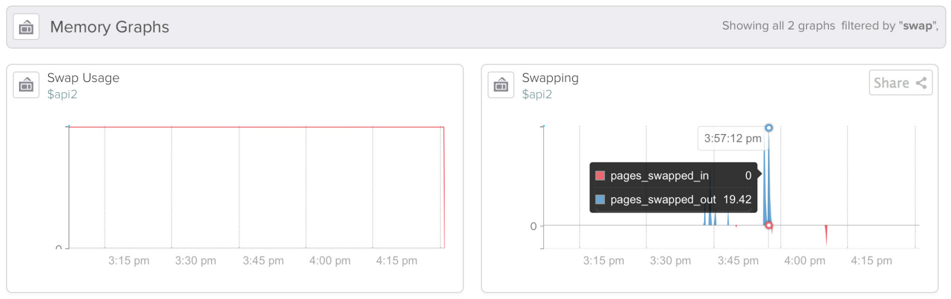
A common mistake is to change a setting with a `SET` statement and forget to update the server's configuration file, so when it restarts it reverts to the old setting. You'll need to remember to update the file for any changes you make while the server is running.

Many configuration variables have specific scopes. Some are global to the whole server; others are connection-specific but get their initial value from a server-wide default; some are per-query. Be sure you understand which types of variables you're changing, so you can reason about the effects. Also note that variables have different units; some are in bytes, some in units of some appropriate quantity, some in megabytes, and so on.

Cautions Before Beginning

You shouldn't be afraid to approach your server's configuration, but there are common mistakes that it's good to know about so you can avoid them.

The first thing to know is that some settings can harm your server's performance! There's such a thing as too much of a good thing for specific memory buffers, for example. The most common consequence of setting a buffer too large is using swap, which slows the server down a lot. If your server is actively swapping everything will grind to a halt. You can see swapping activity in VividCortex's memory graphs. Note that it's OK if some swap is used; what you want to avoid is active paging in and out. Notice the difference between the graph on the left and the graph on the right in the screenshot.



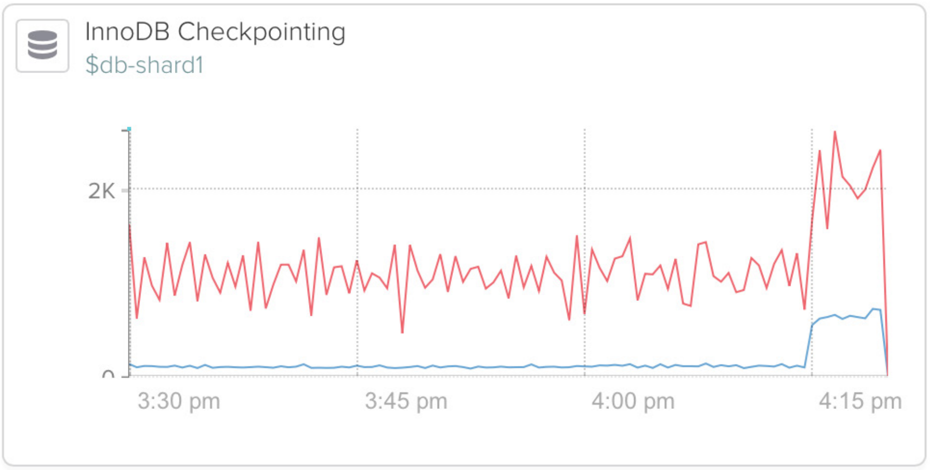
One of the most common sources of query performance problems is a bad setting for the `sort_buffer_size` variable. We'll discuss this one in detail later.

Much online advice can be found about configuration. You should take it all, including what you're reading right now, with a grain of salt. You know your server better than an online forum posting; if something seems fishy to you, check into it. Likewise, a lot of the tuning scripts have caused performance problems when rules of thumb are put to use in situations that are exceptions to the rule.

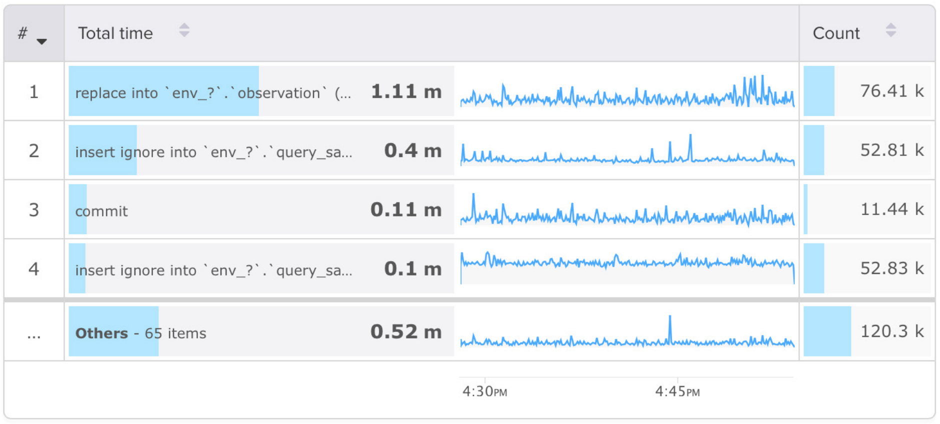
Much of this is because of their use of equations to suggest values for various settings. Many of these are cargo-cult advice that is like a broken clock: it's right twice a day, so someone decided it's a universal truth. Be careful with those.

Finally, you should always change only one setting at a time, then measure what happens. Before we had VividCortex, we used to diff before-and-after snapshots of the `SHOW STATUS` command. VividCortex makes this easy, though, because every status variable is automatically available in our extensive set of graphs.

"VividCortex measures and records every query's execution, making it trivially easy to see what's changed around a point in time.

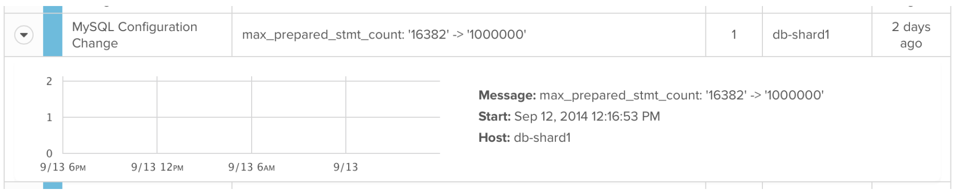


Even better than looking at status variables, though, is actual query performance. VividCortex measures and records every query's execution, making it trivially easy to see what's changed around a point in time. You can use the Compare Queries feature to see a before-and-after view of what's changed, or just eyeball specific queries of interest in the Top Queries view. We've found many changes in query behavior in this fashion.



#	Total time			Count
1	replace into \`env_?\`.\`observation\` (...	1.11 m		76.41 k
2	insert ignore into \`env_?\`.\`query_sa...	0.4 m		52.81 k
3	commit	0.11 m		11.44 k
4	insert ignore into \`env_?\`.\`query_sa...	0.1 m		52.83 k
...	**Others** - 65 items	0.52 m		120.3 k

Keeping track of your changes is a good idea, too. VividCortex will help with this: each variable change is registered as an event, with the before-and-after value in the message. You can see this in your Event dashboard.


MySQL Configuration Change
max_prepared_stmt_count: '16382' -> '1000000'
1
db-shard1
2 days ago
2
1
0
9/13 6PM
9/13 12PM
9/13 6AM
9/13

Message: max_prepared_stmt_count: '16382' -> '1000000'
Start: Sep 12, 2014 12:16:53 PM
Host: db-shard1

With all that out of the way, let's move on to looking at the settings that can help you get the best performance from your MySQL server!

Basic MySQL Setup

One of the fundamental things you need to do is ensure that overall settings are configured to smart default values. In recent MySQL versions (MySQL 5.6 and newer), this has gotten easier; some settings in older versions were truly bad for any production use at all, and these have been changed to better defaults. However, there are still some variables you'll need to change.

The Default Storage Engine

The `default_storage_engine` setting is the engine used for tables that don't specify an engine explicitly. This used to be MyISAM, but it's changed to InnoDB more recently. This is a good thing; it's the engine we've suggested as default for most use cases for years. Just check to ensure this is set as you desire. If you don't know what engine to use, the right answer is probably InnoDB.

Avoiding Outages Caused By DNS

The `skip_name_resolve` setting doesn't help performance much unless there's something wrong with DNS lookups, which happens a lot. In normal operation, connections to the server cause it to perform DNS lookups, and if DNS isn't responding fast, this setting can cause your server to become unavailable to clients. Setting this variable to ON will prevent this. It does have some consequences for server behavior and how you define users and grants; read the documentation for details on that.

The Query Cache

The query cache (variables whose names begin `query_cache`) should generally be disabled, unless you're quite familiar with MySQL and its inner workings and know that it'll help on your workload. You can find a lot of discussion online about the potential impacts of the query cache. Much of it misses the forest for the trees: the fact is that the query cache doesn't scale well on modern multi-core hardware and can cause serious lockups of the whole server. It's not worth dissecting the reasons in great detail here; if you want good detailed advice, see High Performance MySQL, 3rd Edition.

The Binary Log

The `log_bin` variable enables the MySQL binary log, which is required for replication. If you enable this, you should consider the settings for `sync_binlog` and `sync_relay_log`. These configure how these various logs of data changes are written to durable storage to protect against data loss in the event of a server crash.

For purposes of performance, these settings represent a fundamental set of tradeoffs you must make. For the safest operation, you must configure the server to write data to disk immediately when it's changed, but that is very slow in comparison with writing to memory and flushing to disk in the background. For good performance with maximum durability, you will need SSD drives, or a RAID controller with a writeback cache that's battery protected.

The Table Cache

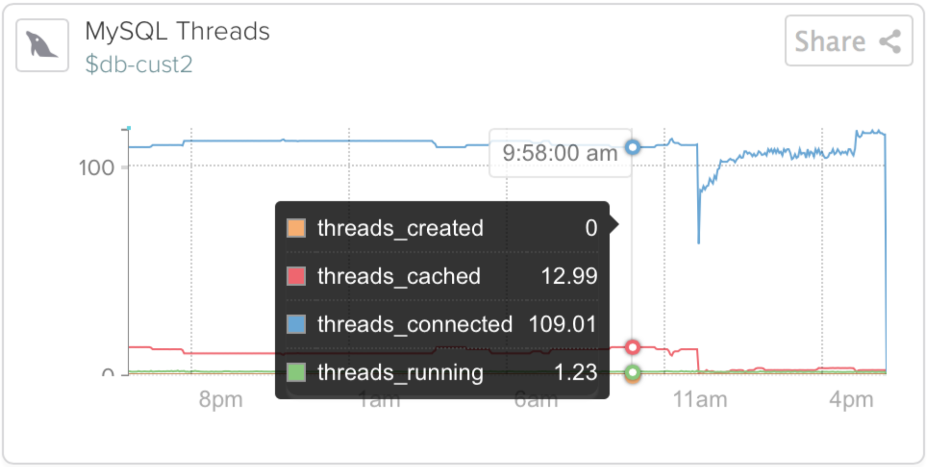
The table cache actually has several related settings and status counters, all beginning with `table_cache` , `table_definition` , and `table_open` ; the names have changed over time, and this is one of the variables to consider when planning an upgrade, to ensure you don't encounter incompatibilities.

The server's behavior has changed a lot, too. In newer versions, for example, the table cache is partitioned, and you can choose how many partitions there are. It also interacts with InnoDB's internal data dictionary cache. Read the documentation for more details on this.

This setting is mainly important if you have a lot of tables. In general, unless you are working with tens of thousands of tables, it's a good idea to keep them open in memory for fast access. If the table cache is too small, the server will open and close file descriptors a lot, as well as repeatedly reading and parsing the table definition from the `.frm` files. This can potentially be a bottleneck.

The Thread Cache

The thread cache (`thread_cache_size`) lets the server reuse operating system threads. It should generally be set large enough that threads aren't created and destroyed a lot. You can look at the graphs of the `Threads_created` variable to see whether this is the case.



🟧	threads_created	0
🟥	threads_cached	12.99
🟦	threads_connected	109.01
🟩	threads_running	1.23

The Sort Buffer

The sort buffer, which is controlled by the `sort_buffer_size` setting, is probably one of the most-discussed MySQL settings. Interestingly, in many cases it has a lot less potential to help performance than to hurt it, so configuring it is often about avoiding trouble rather than gaining a lot of performance.

The variable controls the size of a buffer that's created whenever

MySQL has to sort rows. It is per-query, meaning each query gets its own buffer, and it's allocated to its full size, not as-much-as-needed. This makes large settings potentially dangerous.

The worst abuse of this variable we've seen came from a server that was tuned with a script. The script relied on a naive formula that looked at a simplistic ratio of some server variables. Due to the server's workload the script was never satisfied and continually suggested increasing this variable, which eventually was set to 1GB. The effect was that every time the server needed to sort a few rows, 1GB of memory was allocated. As you might expect, this was not only slow, but crashed the server pretty frequently due to the out-of-memory killer being invoked.

What if the variable is too small? This matters most when there's a large number of rows to sort (the type of situation you might see with a large analytical query). In this case, the server will generate rows and write them to the buffer in memory, fill the buffer, then write the buffer to a sort file and repeat. The server then sorts these temporary files and merges them to sort the whole result-set. If this happens many times, it can be slow.

This variable can be set per-connection temporarily if such a large query is anticipated, and that's probably better than setting it large globally. Settings that help large queries can seriously hurt small, fast queries.

You can find out whether a query is potentially a candidate for a larger sort buffer by looking at the number of rows sorted or the sort merge passes required. VividCortex can use regression to estimate this on a per-query basis (MySQL makes it available as a global status counter, not per-query, unless you are using the Performance Schema in MySQL 5.6 and later). If you identify queries that might benefit from a larger sort buffer, you can then try some experiments by hand. You can look at the `Sort_merge_passes` local (not global) status counter before and after running the query, and tune and time until you determine whether the setting helps.

If you do configure a large per-connection setting, be sure that you either close the connection or reset the variable before, for example, returning the connection to a connection pool where the setting might impact other queries.

Configuring InnoDB Settings

InnoDB is the default, and most widely used, storage engine for MySQL. As such, it's very important to configure correctly, and the default configuration is not suitable for most production usage. (An alternative to InnoDB is TokuDB, but it isn't provided in standard MySQL from Oracle; you can build it yourself or use builds from MariaDB or Percona, among others).

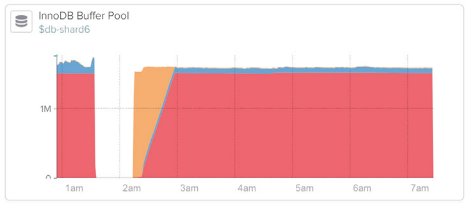
The Buffer Pool

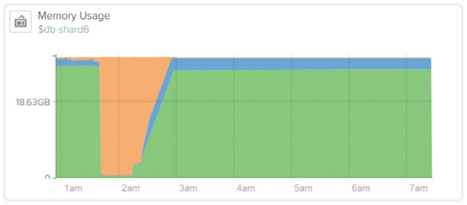
"

Give the buffer pool all the memory not needed for other purposes in the server.

The InnoDB buffer pool is usually the largest and most important memory buffer in the entire server. There are various ways to estimate how much memory you can allocate to it, but all of them need to be cross-checked by monitoring the server's memory usage in production to ensure you don't cause swapping.

The general idea, regardless of how you estimate an intial setting, is to give the buffer pool all the memory not needed for other purposes in the server. You'll need to leave some memory for the OS filesystem cache, other MySQL global and per-connection or per-query buffers, and so on. You might find that on a server with 32GB of memory, for example, you can allocate about 25GB for the InnoDB buffer pool. Consider the graphs shown below.





This shows the server during and after a restart. You can see how the buffer pool rapidly consumes a lot of memory, and the OS then slowly uses the remaining memory for its caches until the server reaches a steady-state. There is nothing wrong with using quite a bit of the server's memory for OS caches, although if this server were exclusively dedicated to MySQL we might configure it to use more of the available memory (so the blue area would shrink and the green would enlarge).

You'll need to configure the `innodb_flush_method` setting in concert with this variable. The behavior of this setting is a bit complex (see High Performance MySQL 3rd Edition for an in-depth examination), but most performance experts agree that `O_DIRECT` is the setting to use here, at least on Linux. It bypasses the operating system's filesystem cache so there's no double-buffering.

The InnoDB Log Files And Log Buffer

"

If the InnoDB transaction log is too small, you can experience frequent performance degradation, which may appear as events in the Faults Dashboard.

The second most important setting for InnoDB is the transaction log (redo log). The log needs to be larger than its typical default. The log's behavior is also quite complex, and affects a lot of InnoDB's activity during operations such as committing transactions, as well as its routine background housekeeping tasks. If it's too small, you can experience frequent performance degradations (stalls) as well as an overall increase in load due to the server not being able to batch work as efficiently. If you use VividCortex, the resulting stalls may appear as frequent events in the Faults Dashboard.

The InnoDB log can be configured in several files; it's 2 by default. You can leave this as-is. This setting is only relevant if your operating system doesn't support large files, which all modern OSes do. What you should care about most is the total allocated log, which is `innodb_log_file_size` multiplied by the number of log files.

In older versions, InnoDB only supported up to 4GB of total log space. At the same time, its recovery routines were inefficient on large logs. This has been greatly improved in recent versions, and you don't have to set the log with such caution. It may still be the case, however, that large logs cause slower recovery.

Logs are written in a round-robin fashion, and old transactions' changes have to be flushed from the buffer pool to disk before their log entries can be overwritten. As a result, one way to choose the size for the logs is by estimating how many bytes will be written to the log in an hour or so, and configure the log at least that large. To give an idea, many production servers are running with logs between 256MB and 2GB, and larger logs are certainly not unheard-of.

Be careful changing this setting in older MySQL versions. It used to require a tedious process to do it safely. In MySQL 5.6 and newer, it's automatic and online, which is much nicer.

There's another variable you'll want to consider at the same time: the `innodb_log_buffer_size` . This is the size of the in-memory buffer to which writes are directed before being written to the log. This is an optimization that can make a great deal of difference on write-heavy workloads. It can reduce the amount of load on the hardware as well as helping avoid contention inside InnoDB. Values up to 100MB or so are not uncommon to see.

"

The InnoDB transaction log flush policy is very important for durability and impacts performance greatly.

Finally, you'll want to think about your performance versus durability requirements. The setting that can make a huge difference for durability also impacts performance greatly: `innodb_flush_log_at_trx_commit` . When set to 1, its default, every transaction's log writes are flushed to disk with an `fsync()` or similar call immediately, which can be very slow. When set to 2, such log writes can be flushed in batches every 1 second or so, which can be much less burdensome and faster, at the cost of exposing you to a potential 1 second or so of lost transactions in the event of a crash.

General InnoDB Settings

We'll conclude this section with a couple of variables you should consider to avoid hassles or potential problems, but which aren't absolutely vital.

The first is the `innodb_stats_on_metadata` setting. This controls whether InnoDB does the equivalent of an `ANALYZE TABLE` when you issue queries to get metadata information about tables, e.g. `SHOW TABLE STATUS` . Setting this variable to OFF can avoid serious impact to your server in some scenarios with lots of data, especially since it avoids an expensive path in a lot of queries that are used by GUI tools many people like to use.

The other setting is `innodb_file_per_table`. This defaults to ON in recent versions of MySQL. Having this set ON avoids giant, monolithic single files with all your data, that are difficult to manage operationally.

Advanced Considerations

Any guide such as this is automatically incomplete, and can't even begin to cover the breadth of scenarios in which MySQL is deployed. A few things we won't cover, but we'll just mention quickly, are probably worth looking into more deeply. You may want to hire a professional services firm (consultant or support provider) to help you with these topics.

Consider Your Hardware

Settings depend a lot on your hardware, and the choice of hardware depends in turn on what you intend to use your server for. Some things to consider include the amount of memory, networking, CPUs (number, speed, and configuration), and of course your storage: spindle-based, SSD, PCIe, SAN, RAID, and so on. Entire books could be written about these topics. And we didn't even mention the cloud yet!

On modern multicore, high-memory hardware, some internal bottlenecks in MySQL and InnoDB can become more prominent, and specific configuration variables may help or hurt in these circumstances. For example, InnoDB's partitioned buffer pool was largely a result of multi-CPU servers; you can change the number of buffer pool "instances" if necessary. There are many settings you might wish to consider in such specific cases.

Consider Your Workload

What types of data does your server store? What types of queries do you run against it? A high-throughput, high-speed single-table primary key lookup workload is different from a data warehouse that may handle long-running queries that touch millions of rows. Consider your requirements for reads versus writes, how much throughput you need, how much concurrency you'll drive at various mutexes and other contention points, and whether your goal is minimum response time, maximum throughput, or maximum work capacity.

This is an area where workload analysis is very helpful. If you don't use VividCortex, you should certainly use another performance management tool that is capable of high-resolution query analysis, because workload is all about queries and their performance. There's no substitute to be found in global status counters, which don't measure and expose information about vital metrics such as query latency.

Conclusions

Although MySQL has nearly 400 configuration settings to consider, most of the benefit comes from getting the fundamentals right. These include simple choices among alternatives such as storage engines, the various thread and connection settings, and crucially, InnoDB's buffer pool, transaction log, and related items. You should always keep careful records of what you change, and examine performance changes to all queries whenever you reconfigure something with a specific query's performance in mind. Happy configuring!

About VividCortex:

*VividCortex is SaaS database performance management for MySQL. VividCortex gives you rapid insight into MySQL that accelerates IT delivery, improves application performance and availability, and reduces infrastructure cost. Unlike traditional monitoring, we measure and analyze the system's work and resource consumption. This leads directly to better performance for IT as a whole, at reduced cost and effort. To find out how leading companies are benefiting from VividCortex and learn how it could help you too, **contact us** for a personalized demo.*