Recommended Settings for MySQL 5.6, 5.7, 8.0 Server for Online Transaction Processing (OLTP) and Benchmarking (Doc ID 1531329.1)

To Bottom

In this Document

APPLIES TO:

MySQL Server - Version 5.6 and later
Information in this document applies to any platform.

PURPOSE

Strongly recommended initial settings for MySQL Server when used for OLTP or benchmarking.

SCOPE

For DBAs having OLTP-like workloads or doing benchmarking.

DETAILS

We recommend that when using MySQL Server 5.6 you include the following settings in your my.cnf or my.ini file if you have a transaction processing or benchmark workload. They are also a good starting point for other workloads. These settings replace some of our flexible server defaults for smaller configurations with values that are better for higher load servers. These are starting points. For most settings the optimal value depends on the specific workload and you should ideally test to find out what settings are best for your situation. The suggestions are also likely to be suitable for 5.7 but 5.7-specific notes and recommendations are a work in progress.

If a support engineer advises you to change a setting, accept that advice because it will have been given after considering the data they have collected about your specific situation.

Changes to make in all cases

These improve on the defaults to improve performance in some cases, reducing your chance of encountering trouble.

innodb_stats_persistent = 1   # Also use ANALYZE TABLE for all tables periodically
innodb_read_io_threads = 16   # Check pending read requests in SHOW ENGINE INNODB STATUS to see if more might be useful, if seldom more than 64 * innodb_read_io_threads, little need for more.
innodb_write_io_threads = 4
table_open_cache_instances = 16 # 5.7.8 onwards defaults to 16

metadata_locks_hash_instances = 256 # better hash from 5.6.15,5.7.3. Irrelevant and deprecated from 5.7.4 due to change in metadata locking

Main settings to review

Also make these additions and adjust as described to find reasonably appropriate values:

innodb_buffer_pool_size: the single most important performance setting for most workloads, see the memory section later for more on this and InnoDB log files. Consider increasing innodb_buffer_pool_instances (5.6 manual) from the 8 default to buffer pool size / 2GB (so 32 for 64g pool) if concurrency is high, some old benchmark results to illustrate why..

innodb_stats_persistent_sample_pages: a value in the 100 to 1000 range will produce better statistics and is likely to produce better query optimising for non-trivial queries. The time taken by ANALYZE TABLE is proportional to this and this many dives will be done for each index, so use some care about setting it to very large values.

innodb_flush_neighbors = 0 if you have SSD storage. Do not change from server default of 1 if you are using spinning disks. Use 0 if both.

innodb_page_size: consider 4k for SSD because this better matches the internal sector size on older disks but be aware that some might use the newer 16k sector size, if so, use that. Check your drive vendor for what it uses.

innodb_io_capacity: for a few spinning disks and lower end SSD the default is OK, but 100 is probably better for a single spinning disk. For higher end and bus-attached flash consider 1000. Use smaller values for systems with low write loads, larger with high. Use the smallest value needed for flushing and purging to keep up unless you see more modified/dirty pages than you want in the InnoDB buffer pool. Do not use extreme values like 20000 or more unless you have proved that lower values are not sufficient for your workload. It regulates flushing rates and related disk i/o. You can seriously harm performance by setting this or innodb_io_capacity_max too high and wasting disk i/o operations with premature flushing.

innodb_io_capacity_max: for a few spinning disks and lower end SSD the default is OK but 200-400 is probably better for a single spinning disk. For higher end and bus-attached flash consider 2500. Use smaller values for systems with low write loads, larger with high. Use the smallest value needed for flushing and purging to keep up. Twice innodb_io_capacity will often be a good choice and this can never be lower than innodb_io_capacity.

innodb_log_file_size = 2000M is a good starting point. Avoid making it too small, that will cause excessive adaptive flushing of modified pages. More guidance here.

innodb_lru_scan_depth: Reduce if possible. Uses disk i/o and can be a CPU and disk contention source. This multiplied by innodb_buffer_pool_instances sets how much work the page cleaner thread does each second, attempting to make that many pages free. Increase or decrease this to keep the result of multiplying the two about the same whenever you change innodb_buffer_pool_instances, unless you are deliberately trying to tune the LRU scan depth. Adjust up or down so that there are almost never no free pages but do not set it much larger than needed because the scans have a significant performance cost. A smaller value than the default is probably suitable for most workloads, give 100 a try instead of the default if you just want a lower starting point for your tuning, then adjust upwards to keep some free pages most of the time. Increase innodb_page_cleaners to lower of CPU count or buffer pools if it cannot keep up, there are limits to how much writing one thread can get done; 4 is a useful change for 5.6, 5.7 default is already 4. The replication SQL thread(s) can be seriously delayed if there are not usually free pages, since they have to wait for one to be made free. Error log Messages like "*Log Messages: page_cleaner: 1000ms intended loop took 8120ms. The settings might not be optimal.*" usually indicate that you have the page cleaner told to do more work than is possible in one second, so reduce the scan depth, or that there is disk contention to fix. Page cleaner has high thread priority in 5.7, particularly important not to tell it to do too much work, helps it to keep up. Document 2014477.1 has details of related settings and measurements that can help to tune this.

innodb_checksum_algorithm=strict_crc32 if a new installation, else crc32 for backwards compatibility. 5.5 and earlier cannot read tablespaces created with crc32. Crc32 is faster and particularly desirable for those using very fast storage systems like bus-attached flash such as Fusion-IO with high write rates.

innodb_log_compressed_pages = 0 if using compression. This avoids saving two copies of changes to the InnoDB log, one compressed, one not, so reduces InnoDB log writing amounts. Particularly significant if the log files are on SSD or bus-attached flash, something that should often be avoided if practical though it can help with commit rates if you do not have a write caching disk controller, at the cost of probably quite shortened SSD lifetime.

binlog_row_image = minimal assuming all tables have primary key, unsafe if not, it would prevent applying the binary logs or replication from working. Saves binary log space. Particularly significant if the binary logs are on SSD or flash, something that should often be avoided.

table_definition_cache: Set to the typical number of actively used tables within MySQL. Use SHOW GLOBAL STATUS and verify that Opened_table_definitions is not increasing by more than a few per minute. Increase until that is true or the value becomes 30000 or more, if that happens, evaluate needs and possibly increase further. Critical: see Performance Schema memory notes. Do not set to values that are much larger than required or you will greatly increase the RAM needs of PS in 5.6, much less of an issue in 5.7. Note that in 5.6 801 can cause four times the PS RAM usage of 800 by switching to large server calculation rules and 400 can be about half that of 401 if no other setting causes large rules.

table_open_cache: set no smaller than table_definition_cache, usually twice that is a good starting value. Use SHOW GLOBAL STATUS and verify that Opened_tables is not increasing by more than a few per minute. Increase until that is true or the value becomes 30000 or more, if that happens, evaluate needs and possibly increase further. Critical: see Performance Schema memory notes. Do not set to values that are much larger than required in 5.6, much less of an issue in 5.7, or you will greatly increase the RAM needs of PS. Note that in 5.6 4001 can cause four times the PS RAM usage of 4000 by switching to large server calculation rules and 2000 can be about half that of 2001 if no other setting causes large rules.

max_connections: This is also used for autosizing Performance Schema. Do not set it to values that are far higher than really required or you will greatly increase the memory usage of PS. If you must have a large value here because you are using a connection cache, consider using a thread cache as well to reduce the number of connections to the MySQL server. Critical: see Performance Schema memory notes. Do not set to values that are much larger than required or you will greatly increase the RAM needs of PS. Note that 303 can cause four times the PS RAM usage of 302 by switching to large server calculation rules and 151 can be about half that of 302 if no other setting causes large rules.

open_files_limit: This is also used for autosizing Performance Schema. Do not set it to values that are far higher than really required in 5.6, less of an issue in 5.7.

sort_buffer_size = 32k is likely to be faster for OLTP, change to that from the server default of 256k. Use SHOW GLOBAL STATUS to check Sort_merge_passes. It the count is 0 or increasing by up to 10-20 per second you can decrease this and probably get a performance increase. If the count is increasing by less than 100 per second that is also probably good and smaller sort_buffer_size may be better. Use care with large sizes, setting this to 2M can reduce throughput for some workloads by 30% or more. If you see high values for

Sort_merge_passes, identify the queries that are performing the sorts and either improve indexing or set the session value of sort_buffer_size to a larger value just for those queries.

innodb_adaptive_hash_index (5.6 manual) Try both 0 and 1, 0 may show improvement if you do a lot of index scans, particularly in very heavy read-only or read-mostly workloads. Some people prefer always 0 but that misses some workloads where 1 helps. There's an improvement in concurrency from 5.7.8 to use multiple partitions and the option innodb_adaptive_hash_index_parts was added, this may change the best setting from 0 to 1 for some workloads, at the cost of slower DBT3 benchmark result with a single thread only. More work planned for 5.8.

innodb_doublewrite (5.6 manual) consider 0/off instead of the default 1/on if you can afford the data protection loss for high write load workloads. This has gradually changed from neutral to positive in 5.5 to more negative for performance in 5.6 and now 5.7.

Where there is a recommendation to check SHOW GLOBAL STATUS output you should do that after the server has been running for some time under load and has stabilised. Many values take some time to reach their steady state levels or rates.

SSD-specific settings

Ensure that trim support is enabled in your operating system, it usually is.

Set innodb_page_size=4k unless you want a larger size to try to increase compression efficiency or have an SSD with 16k sectors. Use innodb_flush_neighbors=0 .

Memory usage and InnoDB buffer pool

For the common case where InnoDB is storing most data, setting innodb_buffer_pool_size to a suitably large value is the key to good performance. Expect to use most of the RAM in the server for this, likely at least 50% on a dedicated database server.

The Performance Schema can be a far more substantial user of RAM than in previous versions, particularly in 5.6, less of an issue in 5.7. You should check the amount of RAM allocated for it using SHOW ENGINE PERFORMANCE_SCHEMA STATUS . Any increase of max_connections, open_files_limit, table_open_cache or table_definition_cache above the defaults causes PS to switch to allocating more RAM to allow faster or more extensive monitoring. For this reason in 5.6 in particular you should use great care not to set those values larger than required or should adjust PS memory allocation settings directly. You may need to make PS settings directly to lower values if you have tens of thousands of infrequently accessed tables. Or you can set this to a lower value in my.cnf and change to a higher value in the server init file. It is vital to consider the PS memory allocations in the RAM budget of the server. See On configuring the Performance Schema for more details on how to get started with tuning it. If all of max_connections, table_definition_cache and table_open_cache are the same as or lower than their 151, 400 and 2000 defaults small sizing rules will be used. If all are no more than twice the defaults medium will be used at about twice the small memory consumption (eg. 98 megabytes instead of 52 megabytes). If any is more than twice the default, large rules will be used and the memory usage can be about eight times the small consumption (eg. 400 megabytes). For this reason, avoid going just over the 302, 800 and 4000 values for these settings if PS is being used, or use direct settings for PS sizes. The size examples are with little data and all other settings default, production servers may see significantly larger allocations. From 5.7 the PS uses more dynamic allocations on demand so these settings are less likely to be troublesome and memory usage will vary more with demand than startup settings.

Very frequent and unnecessarily large memory allocations are costly and per-connection allocations can be more costly and also can greatly increase the RAM usage of the server. Please take particular care to avoid over-large settings for: read_buffer_size, read_rnd_buffer_size, join_buffer_size, sort_buffer_size, binlog_cache_size and net_buffer_length. For OLTP work the defaults or smaller values are likely to be best. Bigger is not usually better for these workloads. Use caution with larger values, increasing sort_buffer_size from the default 256k to 4M was enough to cut OLTP performance by about 30% in 5.6. If you need bigger values for some of these, do it only in the session running the query that needs something different.

The operating system is likely to cache the total size of log files configured with innodb_log_file_size. Be sure to allow for this in your memory budget.

Thread_stack is also a session setting but it is set to the minimum safe value for using stored procedures, do not reduce it if using those. A maximum reduction of 32k might work for other workloads but remember that the server will crash effectively randomly if you get it wrong. It's not worth touching unless you are both desperate and an expert. We increase this as and only when our tests show that the stack size is too small for safe operation. There is no need for you to increase it. Best not to touch this setting.

Operating systems

CPU affinity: if you are limiting the number of CPU cores, use CPU affinity to use the smallest possible number of physical CPUs to get that core count, to reduce CPU to CPU hardware consistency overhead. On Linux use commands like taskset -c 1-4 `pid of mysqld` or in windows START /AFFINITY or the Task Manager affinity control options.

Linux

Memory allocator: we ship built to use libc which is OK up to about 8-16 concurrent threads. From there switch to using TCMalloc using the mysqld_safe --malloc-lib option or LD_PRELOAD or experiment with the similar and possibly slightly faster jemalloc, which might do better with memory fragmentation, though we greatly reduced potential fragmentation in MySQL 5.6. TCMalloc 1.4 was shipped with many MySQL versions until 5.6.31 and 5.7.13. A the time of writing TCMalloc 2.5 is the latest version so you may want to experiment with that and jemalloc to see which works best for your workload and system.

Starting from MySQL Server 8.0.22, the new innodb_extend_and_initialize variable permits configuring how InnoDB allocates space to file-per-table and general tablespaces on Linux. By default, when an operation requires additional space in a tablespace, InnoDB allocates pages to the tablespace and physically writes NULLs to those pages. This behavior affects performance if new pages are allocated frequently.

IO scheduler: use noop or deadline. In rare cases CFQ can work better, perhaps on SAN systems, but usually it is significantly slower. *echo noop > /sys/block/{DEVICE-NAME}/queue/scheduler* .

nice: using nice -10 in mysqld_safe can make a small performance difference on dedicate servers, sometimes larger on highly contended servers. nice -20 can be used but you may find it hard to connect interactively if mysqld is overloaded and -10 is usually sufficient. If you really want -20, use -19 so you can still set the client mysql to -20 to get in and kill a rogue query.

Use *cat "/proc/`pgrep -n mysqld`/limits* to check the ulimit values for a running process. May need *ulimit -n* to set maximum open files per process and *ulimit -u* for a user. The MySQL open_files_limit setting should set this but verify and adjust directly if needed.

It is often suggested to use "numactl --interleave all" to prevent heavy swapping when a single large InnoDB buffer pool is all allocated on one CPU. Two alternatives exist, using multiple InnoDB buffer pools to try to prevent the allocations all going on one CPU is primary. In addition, check using SHOW VARIABLES whether your version has been built with support for the setting innodb_numa_interleave . If the setting is present, turn it on, setting to 1. It changes to interleaved mode (MPOL_INTERLEAVE) before allocating the buffer pool(s) then back to standard (MPOL_DEFAULT) after. The setting is present on builds compiled on a NUMA system from 5.6.27 onwards.

Set vm.swappiness=1 in /etc/sysctl.conf . It is often suggested to use 0 to swap only an out of memory situation but 1 will allow minimal swapping before that and is probably sufficient. Use whatever works for you but please use caution with 0. Higher values can have a tendency to try to swap out the InnoDB buffer pool to increase the OS disk cache size, a really bad idea for a dedicated database server that is doing its own write caching. If using a NUMA system, get NUMA settings in place before blaming swapping, on swappiness. It is known that a large single buffer pool can trigger very high swapping levels if NUMA settings aren't right, the fix is to adjust the NUMA settings, not swappiness.

Do not set the setting in this paragraph by default. You must test to see if it is worth doing, getting it wrong can harm performance. Default IO queue size is 128, higher or lower can be useful, you might try experimenting with `echo 1000 > /sys/block/[DEVICE]/queue/nr_requests` . Not likely to be useful for single spinning disk systems, more likely on RAID setups.

Do not set the setting in this paragraph by default. You must test to see if it is worth doing, getting it wrong can harm performance. The VM subsystem dirty ratios can be adjusted from the defaults of 10 and 20. To set a temporary value for testing maybe use `echo 5 > /proc/sys/vm/dirty_background_ratio` and `echo 60 > /proc/sys/vm/dirty_ratio` . After proving what works best you can add these parameters to the /etc/sysctl.conf : `vm.dirty_background_ratio = 5` `vm.dirty_ratio = 60`. Please do follow the instruction to test, it is vital not to just change this and 5 and 60 are just examples.

Tools to monitor various parts of a linux system.

Linux Filesystems

We recommend that you use ext4 mounted with (rw,noatime,nodiratime,nobarrier,data=ordered) unless ultimate speed is required, because ext4 is somewhat easier to work with. If you do not have a battery backed up write caching disk controller you can probably improve your write performance by as much as 50% by using the ext4 option data=journal and then the MySQL option skip-innodb_doublewrite. The ext4 option provides the protection against torn pages that the doublewrite buffer provides but with less overhead. The benefit with a write caching controller is likely to be minimal.

XFS is likely to be faster than ext4, perhaps for fsync speed, but it is more difficult to work with. Use mount options (rw,noatime,nodiratime,nobarrier,logbufs=8,logbsize=32k).

ext3 isn't too bad but ext4 is better. Avoid ext2, it has significant limits. Best to avoid these two.

NFS in homebrew setups has more reliability problems than NFS in professional SAN or other storage systems which works well but may be slower than directly attached SSD or bus-attached SSD. It's a balance of features and performance, with SAN performance possibly being boosted by large caches and drive arrays. Most common issue is locked InnoDB log files after a power outage, time or switching log files solves this. Incidence of problems has declined over the last ten years and as of 2016 is now low. If possible use NFSv4 or later protocol for its improved locking handling. If concerned about out of order application of changes, not a problem normally observed in practice, consider using TCP and hard,intr mount option.

Solaris

Use LD_PRELOAD for one of the multi-threaded oriented mallocs, either mtmalloc or umem.

Use UFS/forcedirectio

Use ZFS.

Windows

To support more connections or connection rates higher than about 32 per second you may need to set MaxUserPort higher and TcpTimedWaitDelay for TCP/IP, particularly for Windows Server 2003 and earlier. The defaults are likely to be no more than 4000 ports and TIME_WAIT of 120 seconds. See Settings that can be Modified to Improve Network Performance. Settings of 32768 ports and between 30 and 5 seconds timeout are likely to be appropriate for server usage. The symptom of incorrect settings is likely to be a sudden failure to connect after the port limit is reached, resuming at a slow rate as the timeout slowly frees ports.

Hardware

Battery-backed write-caching disk controllers are useful for all spinning disk setups and also for SSD. SSD alone is a cheaper way to get faster transaction commits than spinning disks, for lower load systems. Do not trust that the controller disables the hard drive write buffers, test with real power outages. You will probably lose data even with a battery if the controller has not disabled the hard drive write buffers.

It is best to split files across disk types in these general groups:

SSD: data, InnoDB undo logs, maybe temporary tables if not using tmpfs or other RAM-based storage for them.

Spinning disks: Binary logs, InnoDB redo logs, bulk data. Also, large SATA drives are cheap and useful for working and archival space as well as the biggest of bulk data sets.

Bus-attached SSD: the tables with the very highest change rates i the most highly loaded systems only.

You can put individual InnoDB tables on different drives, allowing use of SSD for fast storage and SATA for bulk.

Hyperthreading on is likely to be a good choice in most cases. MySQL 5.6 scales up to somewhere in the range of 32-48 cores with InnoDB and hyperthreading counts as an extra core for this purpose. For 5.5 that would be about 16 and before that about 8 cores. If you have more physical cores either without hyperthreading or more when it is enabled, experiment to determine the optimal number to use for MySQL. There is no fixed answer because it depends on workload properties.

Thread pool

Use the thread pool if you routinely run with more than about 128 concurrently active connections. Use it to keep the server at the optimal number of concurrently running operations, which is typically in the range between 32 and 48 threads on high core count servers in MySQL 5.6. If not using the thread pool, use innodb_thread_concurrency if you see that your server has trouble with a build-up of queries above about 128 or so concurrently running operations inside InnoDB. InnoDB shows positive scalability up to an optimal number of running jobs, then negative scalability but innodb_thread_concurrency = 0 has lower overhead when that regulating is not needed, so there is some trade off in throughput stability vs raw performance. The value for peak throughput depends on the application and hardware. If you see a benchmark that compares MySQL with a thread pool to MySQL without, but which does not set innodb_thread_concurrency, that is an indication that you should not trust the benchmark result: no production 5.6 server should be run with thousands of concurrently running threads and no limit to InnoDB concurrency.

Background

Here are more details of why some of these changes should be made.

innodb_stats_persistent = 1

Enables persistent statistics in InnoDB, producing more stable and usually better query optimiser decisions. Very strongly recommended for all servers. With persistent statistics you should run ANALYZE TABLE periodically to update the statistics. Once a week or month is probably sufficient for tables that have fairly stable or gradually changing sizes. For tables that are small or have very rapidly changing contents more frequent will be beneficial. There are minimal possible disadvantages, mainly the need for ANALYZE TABLE sometimes.

innodb_read_io_threads = 16, innodb_write_io_threads = 4

Increases the number of threads used for some types of InnoDB operation, though not the foreground query processing work. That can help the server to keep up with heavy workloads. No significant negative effects for most workloads, though sometimes contention for disk resources between these threads and foreground threads might be an issue if disk utilisation is near 100%.

table_open_cache_instances = 16

Improves the speed of operations involving tables at higher concurrency levels, important for reducing the contention in this area to an insignificant level. No significant disadvantages. This is unlikely to be a bottleneck until 24 cores are in full use but given the lack of cost it is best to set it high enough and never worry about it.

metadata_locks_hash_instances = 256

Reduces the effects of locking during the metadata locking that is used mainly for consistency around DDL. This has been an important bottleneck. As well as the general performance benefit, the hashing algorithm used has been shown to be non-ideal for some situations and that also makes it desirable to increase this value above the default, to reduce the chance of encountering that issue. We're addressing that hash also but this will still be a useful setting with no significant negatives.

innodb_flush_neighbors = 0

When set to 1 InnoDB will look to flush nearby data pages as an optimisation for spinning disks. That optimisation is harmful for SSDs because it increase the number of writes. Set to 0 data on SSDs, 1 for spinning disks. If mixed, 0 is probably best.

innodb_log_file_size = 2000M

This is a critical setting for workloads that do lots of data modification and severe adverse performance will result if it is set too small. You must check the amount of log space used and ensure that it never reaches 75%. You must also consider the effect of your adaptive flushing settings and ensure that the percentage of the log space used does not cause excessive flushing. You can do that by using larger log files or having adaptive flushing start at a higher percentage. There is a trade off in this size because the total amount of log file space will usually be cached in operating system caches due to the nature of the read-modify-write operations performed. You must allow for this in the memory budget of the server to ensure that swapping does not occur. On SSD systems you can significantly extend the life of the drive by ensuring that this is set to a suitably high value to allow lots of dirty page caching and write combining before pages are flushed to disk.

table_definition_cache

Reduces the need to open tables to get dictionary information about the table structures. If set too low this can have a severe negative performance effect. There is little negative effect for the size range given on the table definition cache itself. See the Performance Schema portion of the memory notes above for critical memory usage considerations.

table_open_cache

Reduces the need to open tables to access data. If set too low this can have severe negative performance effects. There is little negative effect for the size range given on the table open cache itself. See the Performance Schema portion of the memory notes above for critical memory usage considerations.

sort_buffer_size = 32k

The key cost here is reduced server speed from setting this too high. Many common recommendations to use several megabytes or more have been made in a wide range of published sources and these are harmful for OLTP workloads. that normally benefit most from 32k or other small values. Do not set this to significantly larger values such as above 256k unless you see very excessive numbers of Sort_merge_passes - many hundreds or thousands per second on busy servers. Even then, it is far better to adjust the setting only in the connection of the few queries that will benefit from the larger size. In cases where it is impossible to adjust settings at the session level and when the workload is mixed it can be useful to use higher than ideal OLTP values to address the needs of the mixture of queries.

Other observations

Query cache

The query cache is effectively a single-threaded bottleneck. It can help performance at low query rates and concurrency, perhaps up to 4 cores routinely used. Above that it is likely to become a serious bottleneck. Leave this off unless you want to test it with your workload, and have measurements that will tell you if it is helping or hurting. Ensure that Qcache_free_blocks in global status is not above 10,000. 5,000 is a good action level. Above these levels the CPU time used in scans of the free list can be an issue, check with FLUSH QUERY CACHE, which defragments the free list, the change in CPU use is the cost of the free list size you had. Reducing the size is the most effective way to manage the free list size. Remember that the query cache was designed for sizes of up to a few tens of megabytes, if you're using hundreds of megabytes you should check performance with great care, it's well outside of its design limitations. Also check for waits with:

SELECT EVENT_NAME AS nm, COUNT_STAR AS cnt, sum_timer_wait, CONCAT(ROUND(sum_timer_wait / 1000000000000, 2), ' s') AS sec

FROM performance_schema.events_stages_summary_global_by_event_name WHERE COUNT_STAR > 0 ORDER BY SUM_TIMER_WAIT DESC LIMIT 20;

Also see MySQL Query Cache Fragmentation Slows Down the Server (Doc ID 1308051.1).

Sync_binlog and innodb_flush_log_at_trx_commit

The 1 setting for these causes one fsync each at every transaction commit in 5.5 and earlier. From 5.6 concurrent commit support helps greatly to reduce that but you should still use care with these settings. Sync_binlog=1 can be expected to cause perhaps a 20% throughput drop with concurrent commit and 30 connections trying to commit, an effect that reduces as actively working connections count increases through to a peak throughput at about 100 working connections. To check the effect, just set sync_binlog to 0 and observe, then set innodb_flush_log_at_trx_commit = 0 and observer. Try innodb_flush_log_at_trx-commit = 2 also, it has less overhead than 1 and more than 0. Finally try both at 0. The speed increase from the 0 settings effect will be greatest on spinning disks with low concurrency and lowest at higher concurrency on fast SSD or with write caching disk controllers.

Note that it is mandatory to use innodb_flush_log_at_trx_commit=1 to get full durability guarantees. Write caching disk controllers with batter backup are the typical way that full durability combined with low performance penalty is achieved.

Bugs that affect upgrades and usage for 5.6 compared to 5.5 and earlier

http://bugs.mysql.com/bug.php?id=69174

Innodb_max_dirty_pages_pct is effectively broken at present, only working when the server has been idle for a second or more. There are a range of implications:

1. In past versions the limit on innodb_log_file_size sometimes made it necessary to use this setting to avoid hitting 75% of log space use and having a production disruption incident due to hitting async flushing at 75%. The much more gentle flushing batches from innodb_max_dirty_pages_pct were normally acceptable and it wasn't uncommon for systems with large buffer pools and high needs to have innodb_max_dirty_pages_pct set to values in the 2-5% range just for this reason. In 5.6 you have two possibilities that should work better:

1a. You can use larger values for innodb_log_file_size. That will let you use more of your buffer pool for write combining and reduce total io operations, instead of being forced to do lots of avoidable ones just to avoid reaching 75% of the log file use. Be sure you allow for the RAM your OS will use for buffering the log files, assume as much RAM use as the total log file space you set. This should greatly increase the value of larger buffer pools for high write load workloads.

1b. You can set innodb_adaptive_flushing_lwm to avoid reaching 75% of log space use. The highest permitted value is 70%, so adaptive flushing will start to increase flushing rate before the server gets to 75% of the log file use. 70% is a good setting for systems with low write rates or very fast disk systems that can easily handle a burst of writes. For others you should adjust to whatever lower value it takes to produce a nice and smooth transition from innodb_io_capacity based level flushing to adaptive flushing. 10% is the default but that is probably too low for most production systems, just what we need for a default that has to handle a wide range of possible cases.

2. You can't effectively use the normal practice of gradually reducing innodb_max_dirty_pages_pct before a shutdown, to reduce outage duration. The best workaround at present is to set innodb_io_capacity to high values so it will cause more flushing.

3. You can't use innodb_max_dirty_pages_pct to manage crash recovery time, something it could do with less disruptive writing than the alternative of letting the server hit async flushing at 75% of log file space use, after deliberately setting innodb_log_file_size too low. The workarounds are to use higher than desirable innodb_io_capacity and smaller than desirable innodb_log_file_size. Both cause unnecessary flushing compared to using innodb_max_dirty_pages_pct for this task. Before using a too small innodb_log_file_size, experiment with innodb_io_capacity and innodb_adaptive_flushing_lwm. Also ensure that innodb_io_capacity_max is set to around twice innodb_io_capacity, rarely up to four or more times. This may eliminate the issue with less redundant io than very constrained log file sizes because adaptive flushing will increase the writing rate as the percentage of log space used increases, so you should be able to reach almost any target recovery time limit, though still at the cost of more io than using innodb_max_dirty_pages_pct to do it only when a hard cap is reached.

4. You can't use innodb_max_dirty_pages_pct to effectively regulate the maximum percentage of dirty pages in the buffer pool, constraining them to a target value. This is likely to be of particular significance during data loading and with well cached workloads where you want to control the split between pages used for caching modified data and pages used for caching data used purely for reads.

The workaround for this is to regard innodb_adaptive_flushing_lwm as equivalent to the use of innodb_max_dirty_pages_pct for normal production and set it to something like 60% with a suitable value of innodb_io_capacity for the times when the workload hasn't reached that amount of log file usage. Start low like 100 and gradually increase so that at medium load times it just about keeps up. Have innodb_io_capacity_max set to a relatively high value so that as soon as the low water mark is passed, lots of extra IO will be done to cap the dirty pages/log space use.

You may then be able to reduce the size of your InnoDB log files if you find that you don't reach 60% of log space use when you have reached a suitable percentage of dirty pages for the page read/write balance for your server. If you can you should do this because you can reallocate the OS RAM used for caching the bigger log files to the InnoDB buffer pool or other uses.

For benchmarking and opitimization of 8.0, it is important to note that "Benchmarking" is basically an "artificial" max speed that doesn't take into account many of the factors that can hit and slow down a Database. However, proper Optimization and architecture usually needs to override those artificial settings and while those changes may cause slower benchmarks they will lead to a more solid performance of your database over time.

That said, The manual has great information in the following places:
https://dev.mysql.com/doc/refman/8.0/en/optimize-benchmarking.html Is a great link that will give you some helpful tips and tools to use for benchmarking.

https://dev.mysql.com/doc/refman/8.0/en/optimization.html will get into the actual tuning advice which is highly recommended reading for all MySQL Database professionals.

REFERENCES

https://dev.mysql.com/doc/refman/5.6/en/replication-options-binary-log.html#sysvar_binlog_row_image
https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_table_definition_cache
https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_table_open_cache
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_adaptive_hash_index
https://dev.mysql.com/doc/refman/5.6/en/server-status-variables.html#statvar_Innodb_page_size
http://dimitrik.free.fr/blog/archives/2012/10/mysql-performance-innodb-buffer-pool-instances-in-56.html
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_stats_persistent
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_buffer_pool_instances
https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_page_cleaners
http://msdn.microsoft.com/en-us/library/ee377084.aspx
https://dev.mysql.com/doc/refman/5.6/en/server-options.html#option_mysqld_init-file
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_log_compressed_pages
http://dimitrik.free.fr/blog/archives/2013/02/mysql-performance-mysql-56-ga-vs-mysql-55-tuning-details.html
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_doublewrite
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_read_io_threads
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_flush_neighbors
https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_adaptive_hash_index_parts
https://dev.mysql.com/doc/refman/5.7/en/innodb-adaptive-hash.html

https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_numa_interleave
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_checksum_algorithm
https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_max_connections
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_lru_scan_depth
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size
https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_adaptive_hash_index
NOTE:2014477.1 - MySQL 5.7 Log Messages: page_cleaner: 1000ms intended loop took 8120ms. The settings might not be optimal. (flushed=0 and evicted=25273, during the time.)
NOTE:1308051.1 - MySQL Query Cache Fragmentation Slows Down the Server
https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_doublewrite
https://dev.mysql.com/doc/refman/5.6/en/mysqld-safe.html#option_mysqld_safe_malloc-lib
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_io_capacity
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_stats_persistent_sample_pages
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_io_capacity_max
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_buffer_pool_instances
https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_instances
https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_table_open_cache_instances
https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_metadata_locks_hash_instances
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_page_size
https://dev.mysql.com/doc/refman/5.6/en/server-options.html#option_mysqld_open-files-limit
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_write_io_threads
http://mysqlserverteam.com/removing-scalability-bottlenecks-in-the-metadata-locking-and-thr_lock-subsystems-in-mysql-5-7/
https://bugs.mysql.com/bug.php?id=68487
http://www.brendangregg.com/linuxperf.html
https://dev.mysql.com/doc/refman/5.6/en/innodb-adaptive-hash.html
https://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html#sysvar_sort_buffer_size
NOTE:1604225.1 - Autosized Performance Schema Options in MySQL Server in MySQL 5.6
http://marcalff.blogspot.co.uk/2013/04/on-configuring-performance-schema.html
https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_doublewrite
http://mysqlmusings.blogspot.co.uk/2012/06/binary-log-group-commit-in-mysql-56.html

https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html#sysvar_innodb_log_file_size