RocksDB Tuning Guide

Edit New Page

Jump to bottom

Yang,Liming edited this page on Jul 13 · 97 revisions

The purpose of this guide is to provide you with enough information so you can tune RocksDB for your workload and your system configuration.

RocksDB is very flexible, which is both good and bad. You can tune it for variety of workloads and storage technologies. Inside of Facebook we use the same code base for in-memory workload, flash devices and spinning disks. However, flexibility is not always user-friendly. We introduced a huge number of tuning options that may be confusing. We hope this guide will help you squeeze the last drop of performance out of your system and fully utilize your resources.

We assume you have basic knowledge of how a Log-structured merge-tree (LSM) works. There are already plenty of great resources on LSM, no need to write one more.

Amplification factors

Tuning RocksDB is often a trade off between three amplification factors: write amplification, read amplification and space amplification.

Write amplification is the ratio of bytes written to storage versus bytes written to the database.

For example, if you are writing 10 MB/s to the database and you observe 30 MB/s disk write rate, your write amplification is 3. If write amplification is high, the workload may be bottlenecked on disk throughput. For example, if write amplification is 50 and max disk throughput is 500 MB/s, your database can sustain a 10 MB/s write rate. In this case, decreasing write amplification will directly increase the maximum write rate.

High write amplification also decreases flash lifetime. There are two ways in which you can observe your write amplification. The first is to read through the output of `DB::GetProperty("rocksdb.stats", &stats)`. The second is to divide your disk write bandwidth (you can use `iostat`) by your DB write rate.

Read amplification is the number of disk reads per query. If you need to read 5 pages to answer a query, read amplification is 5. Logical reads are those that get data from cache, either the RocksDB block cache or the OS filesystem cache. Physical reads are handled by the storage device, flash or disk. Logical reads are much cheaper than physical reads but still impose a CPU cost. You might be able to estimate the physical read rate from `iostat` output but that include reads done for queries and for compaction.

Space amplification is the ratio of the size of database files on disk to data size. If you Put 10MB in the database and it uses 100MB on disk, then the space amplification is 10. You will usually want to set a hard limit on space amplification so you don't run out of disk space or memory.

To learn more about the three amplification factors in context of different database algorithms, we strongly recommend Mark Callaghan's talk at Highload.

RocksDB statistics

When debugging performance, there are some tools that can help you:

statistics -- Set this to `rocksdb::CreateDBStatistics()`. You can get human-readable RocksDB statistics any time by calling `options.statistics.ToString()`. See Statistics for details.

stats_dump_period_sec -- We dump statistics to LOG file every `stats_dump_period_sec` seconds. This is 600 by default, which means that stats will be dumped every 10 minutes. You can get the same data in the application by calling `db->GetProperty("rocksdb.stats");`

Every **stats_dump_period_sec**, you'll find something like this in your LOG file:

```
** Compaction Stats **
Level Files  Size(MB) Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-
Amp Rd(MB/s) Wr(MB/s) Comp(sec) Comp(cnt) Avg(sec) Stall(sec) Stall(cnt) Avg(ms)
KeyIn   KeyDrop
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
------------------
L0      2/0        15   0.5     0.0     0.0      0.0      32.8     32.8       0.0   0.0
0.0     23.0    1457      4346    0.335       0.00          0    0.00             0
0
L1     22/0       125   1.0   163.7    32.8    130.9     165.5     34.6       0.0   5.1
25.6     25.9    6549      1086    6.031       0.00          0    0.00    1287667342
0
L2    227/0      1276   1.0   262.7    34.4    228.4     262.7     34.3       0.1   7.6
26.0     26.0   10344      4137    2.500       0.00          0    0.00    1023585700
0
L3   1634/0     12794   1.0   259.7    31.7    228.1     254.1     26.1       1.5   8.0
20.8     20.4   12787      3758    3.403       0.00          0    0.00    1128138363
0
L4   1819/0     15132   0.1     3.9     2.0      2.0       3.6      1.6      13.1   1.8
20.1     18.4     201       206    0.974       0.00          0    0.00      91486994
```

```
0
Sum  3704/0     29342   0.0   690.1   100.8    589.3     718.7    129.4      14.8  21.9
22.5     23.5   31338     13533    2.316       0.00          0    0.00    3530878399
0
Int     0/0         0   0.0     2.1     0.3      1.8       2.2      0.4       0.0  24.3
24.0     24.9      91        42    2.164       0.00          0    0.00      11718977
0
Flush(GB): accumulative 32.786, interval 0.091
Stalls(secs): 0.000 level0_slowdown, 0.000 level0_numfiles, 0.000 memtable_compaction,
0.000 leveln_slowdown_soft, 0.000 leveln_slowdown_hard
Stalls(count): 0 level0_slowdown, 0 level0_numfiles, 0 memtable_compaction, 0
leveln_slowdown_soft, 0 leveln_slowdown_hard

** DB Stats **
Uptime(secs): 128748.3 total, 300.1 interval
Cumulative writes: 1288457363 writes, 14173030838 keys, 357293118 batches, 3.6 writes
per batch, 3055.92 GB user ingest, stall micros: 7067721262
Cumulative WAL: 1251702527 writes, 357293117 syncs, 3.50 writes per sync, 3055.92 GB
written
Interval writes: 3621943 writes, 39841373 keys, 1013611 batches, 3.6 writes per batch,
8797.4 MB user ingest, stall micros: 112418835
Interval WAL: 3511027 writes, 1013611 syncs, 3.46 writes per sync, 8.59 MB written
```

Compaction stats

Compaction stats for the compactions executed between levels N and N+1 are reported at level N+1 (compaction output). Here is the quick reference:

- Level - for leveled compaction the level of the LSM. For universal compaction all files are in L0. **Sum** has the values aggregated over all levels. **Int** is like **Sum** but limited to the data from the last reporting interval.
- Files - this has two values as (a/b). The first is the number of files in the level. The second is the number of files currently doing compaction for that level.
- Score: for levels other than L0 the score is (current level size) / (max level size). Values of 0 or 1 are okay, but any value greater than 1 means that level needs to be compacted. For L0 the score is computed from the current number of files and number of files that triggers a compaction.
- Read(GB): Total bytes read during compaction between levels N and N+1. This includes bytes read from level N and from level N+1
- Rn(GB): Bytes read from level N during compaction between levels N and N+1
- Rnp1(GB): Bytes read from level N+1 during compaction between levels N and N+1
- Write(GB): Total bytes written during compaction between levels N and N+1
- Wnew(GB): New bytes written to level N+1, calculated as (total bytes written to N+1) - (bytes read from N+1 during compaction with level N)
- Moved(GB): Bytes moved to level N+1 during compaction. In this case there is no IO other than updating the manifest to indicate that a file which used to be in level X is now in level Y

- W-Amp: (total bytes written to level N+1) / (total bytes read from level N). This is the write amplification from compaction between levels N and N+1
- Rd(MB/s): The rate at which data is read during compaction between levels N and N+1. This is (Read(GB) * 1024) / duration where duration is the time for which compactions are in progress from level N to N+1.
- Wr(MB/s): The rate at which data is written during compaction. See Rd(MB/s).
- Rn(cnt): Total files read from level N during compaction between levels N and N+1
- Rnp1(cnt): Total files read from level N+1 during compaction between levels N and N+1
- Wnp1(cnt): Total files written to level N+1 during compaction between levels N and N+1
- Wnew(cnt): (Wnp1(cnt) - Rnp1(cnt)) -- Increase in file count as result of compaction between levels N and N+1
- Comp(sec): Total time spent doing compactions between levels N and N+1
- Comp(cnt): Total number of compactions between levels N and N+1
- Avg(sec): Average time per compaction between levels N and N+1
- Stall(sec): Total time writes were stalled because level N+1 was uncompacted (compaction score was high)
- Stall(cnt): Total number of writes stalled because level N+1 was uncompacted
- Avg(ms): Average time in milliseconds a write was stalled because level N+1 was uncompacted
- KeyIn: number of records compared during compaction
- KeyDrop: number of records dropped (not written out) during compaction

General stats

After the per-level compaction stats, we also output some general stats. General stats are reported for both **cumulative** and **interval**. Cumulative stats report total values from RocksDB instance start. Interval stats report values since the last stats output.

- Uptime(secs): total -- number of seconds this instance has been running, interval -- number of seconds since the last stats dump.
- Cumulative/Interval writes: total -- number of Put calls; keys -- number of entries in the WriteBatches from the Put calls; batches -- number of group commits where each group commit makes persistent one or more Put calls (with concurrency there can be more than 1 Put call made persistent at one point in time); per batch -- average number of bytes in a single batch; ingest -- total bytes written into DB (not counting compactions); stall micros - number of microseconds writes have been stalled when compaction gets behind
- Cumulative/Interval WAL: writes -- number of writes logged in the WAL; syncs - number of times fsync or fdatasync has been used; writes per sync - ratio of writes to syncs; GB written - number of GB written to the WAL
- Stalls: total count and seconds of each stall type since beginning of time: level0_slowdown -- Stall because of `level0_slowdown_writes_trigger`. level0_numfiles -- Stall because of

`level0_stop_writes_trigger`. `memtable_compaction` -- Stall because all memtables were full, flush process couldn't keep up. `leveln_slowdown` -- Stall because of `soft_rate_limit` and `hard_rate_limit`

Perf Context and IO Stats Context

Perf Context and IO Stats Context can help figure out counters within one specific query.

Parallelism options

In LSM architecture, there are two background processes: flush and compaction. Both can execute concurrently via threads to take advantage of storage technology concurrency. Flush threads are in the HIGH priority pool, while compaction threads are in the LOW priority pool. To increase the number of threads in each pool call:

```
options.env->SetBackgroundThreads(num_threads, Env::Priority::HIGH);
options.env->SetBackgroundThreads(num_threads, Env::Priority::LOW);
```

To benefit from more threads you might need to set these options to change the max number of concurrent compactions and flushes:

max_background_compactions is the maximum number of concurrent background compactions. The default is 1, but to fully utilize your CPU and storage you might want to increase this to the minimum of (the number of cores in the system, the disk throughput divided by the average throughput of one compaction thread).

max_background_flushes is the maximum number of concurrent flush operations. It is usually good enough to set this to 1.

General options

filter_policy -- If you're doing point lookups on an uncompacted DB, you definitely want to turn bloom filters on. We use bloom filters to avoid unnecessary disk reads. You should set filter_policy to `rocksdb::NewBloomFilterPolicy(bits_per_key)`. Default bits_per_key is 10, which yields ~1% false positive rate. Larger bits_per_key values will reduce false positive rate, but increase memory usage and space amplification.

block_cache -- We usually recommend setting this to the result of the call `rocksdb::NewLRUCache(cache_capacity, shard_bits)`. Block cache caches uncompressed blocks. OS cache, on the other hand, caches compressed blocks (since that's the way they are stored in files). Thus, it makes sense to use both block_cache and OS cache. We need to lock accesses to block cache and sometimes we see RocksDB bottlenecked on block cache's mutex, especially when DB size is smaller than RAM. In that case, it makes sense to shard block cache by setting shard_bits to a bigger number. If shard_bits is 4, total number of shards will be 16.

allow_os_buffer -- [Deprecated] If false, we will not buffer files in OS cache. See comments above.

max_open_files -- RocksDB keeps all file descriptors in a table cache. If number of file descriptors exceeds max_open_files, some files are evicted from table cache and their file descriptors closed. This means that every read must go through the table cache to lookup the file needed. Set max_open_files to -1 to always keep all files open, which avoids expensive table cache calls.

table_cache_numshardbits -- This option controls table cache sharding. Increase it if table cache mutex is contended.

block_size -- RocksDB packs user data in blocks. When reading a key-value pair from a table file, an entire block is loaded into memory. Block size is 4KB by default. Each table file contains an index that lists offsets of all blocks. Increasing block_size means that the index contains fewer entries (since there are fewer blocks per file) and is thus smaller. Increasing block_size decreases memory usage and space amplification, but increases read amplification.

Sharing cache and thread pool

Sometimes you may wish to run multiple RocksDB instances from the same process. RocksDB provides a way for those instances to share block cache and thread pool. To share block cache, assign a single cache object to all instances:

```
first_instance_options.block_cache = second_instance_options.block_cache =
rocksdb::NewLRUCache(1GB)
```

This will make both instances share a single block cache of total size 1GB.

Thread pool is associated with Env object. When you construct Options, options.env is set to `Env::Default()`, which is best in most cases. Since all Options use the same static object `Env::Default()`, thread pool is shared by default. See Parallelism options to learn how to set number of threads in the thread pool. This way, you can set the maximum number of concurrent running compactions and flushes, even when running multiple RocksDB instances.

Flushing options

All writes to RocksDB are first inserted into an in-memory data structure called memtable. Once the **active memtable** is full, we create a new one and mark the old one read-only. We call the read-only memtable **immutable**. At any point in time there is exactly one active memtable and zero or more immutable memtables. Immutable memtables are waiting to be flushed to storage. There are three options that control flushing behavior.

write_buffer_size sets the size of a single memtable. Once memtable exceeds this size, it is marked immutable and a new one is created.

max_write_buffer_number sets the maximum number of memtables, both active and immutable. If the active memtable fills up and the total number of memtables is larger than max_write_buffer_number we stall further writes. This may happen if the flush process is slower than the write rate.

min_write_buffer_number_to_merge is the minimum number of memtables to be merged before flushing to storage. For example, if this option is set to 2, immutable memtables are only flushed when there are two of them - a single immutable memtable will never be flushed. If multiple memtables are merged together, less data may be written to storage since two updates are merged to a single key. However, every Get() must traverse all immutable memtables linearly to check if the key is there. Setting this option too high may hurt read performance.

Example: options are:

```
write_buffer_size = 512MB;
max_write_buffer_number = 5;
min_write_buffer_number_to_merge = 2;
```

with a write rate of 16MB/s. In this case, a new memtable will be created every 32 seconds, and two memtables will be merged together and flushed every 64 seconds. Depending on the working set size, flush size will be between 512MB and 1GB. To prevent flushing from failing to keep up with the write rate, the memory used by memtables is capped at 5*512MB = 2.5GB. When that is reached, any further writes are blocked until the flush finishes and frees memory used by the memtables.

Level Style Compaction

In Level style compaction, database files are organized into levels. Memtables are flushed to files in level 0, which contains the newest data. Higher levels contain older data. Files in level 0 may overlap, but files in level 1 and higher are non-overlapping. As a result, Get() usually needs to check each file from level 0, but for each successive level, no more than one file may contain the key. Each level is 10 times larger than the previous one (this multiplier is configurable).

A compaction may take a few files from level N and compact them with overlapping files from level N+1. Two compactions operating at different levels or at different key ranges are independent and may be executed concurrently. Compaction speed is directly proportional to max write rate. If compaction can't keep up with the write rate, the space used by the database will continue to grow. **It is important** to configure RocksDB in such a way that compactions may be executed with high concurrency and fully utilize storage.

Compactions at levels 0 and 1 are tricky. Files at level 0 usually span the entire key space. When compacting L0->L1 (from level 0 to level 1), compaction includes all files from level 1. With all files from L1 getting compacted with L0, compaction L1->L2 cannot proceed; it must wait for the L0->L1 compaction to finish. If L0->L1 compaction is slow, it will be the only compaction running in the system most of the time, since other compactions must wait for it to finish.

L0->L1 compaction is also single-threaded. It is hard to achieve good throughput with single-threaded compaction. To see if this is causing issues, check disk utilization. If disk is not fully utilized, there might be an issue with compaction configuration. We usually recommend making L0->L1 as fast as possible by making **the size of level 0 similar to size of level 1**.

Once you determine the appropriate size of level 1, you must decide the level multiplier. Let's assume your level 1 size is 512 MB, level multiplier is 10 and size of the database is 500GB. Level 2 size will then be 5GB, level 3 51GB and level 4 512GB. Since your database size is 500GB, levels 5 and higher will be empty.

Size amplification is easy to calculate. It is `(512 MB + 512 MB + 5GB + 51GB + 512GB) / (500GB) = 1.14`. Here is how we would calculate write amplification: every byte is first written out to level 0. It is then compacted into level 1. Since level 1 size is the same as level 0, write amplification of L0->L1 compaction is 2. However, when a byte from level 1 is compacted into level 2, it is compacted with 10 bytes from level 2 (because level 2 is 10x bigger). The same is also true for L2->L3 and L3->L4 compactions.

Total write amplification is therefore approximately `1 + 2 + 10 + 10 + 10 = 33`. Point lookups must consult all files in level 0 and at most one file from each other levels. However, bloom filters help by greatly reducing read amplification. Short-lived range scans are a bit more expensive, however. Bloom filters are not useful for range scans, so the read amplification is `number_of_level0_files + number_of_non_empty_levels`.

Let's dive into options that control level compaction. We will start with more important ones and follow with less important ones.

level0_file_num_compaction_trigger -- Once level 0 reaches this number of files, L0->L1 compaction is triggered. We can therefore estimate level 0 size in stable state as `write_buffer_size * min_write_buffer_number_to_merge * level0_file_num_compaction_trigger`.

max_bytes_for_level_base and **max_bytes_for_level_multiplier** -- max_bytes_for_level_base is total size of level 1. As mentioned, we recommend that this be around the size of level 0. Each subsequent level is max_bytes_for_level_multiplier larger than previous one. The default is 10 and we do not recommend changing that.

target_file_size_base and **target_file_size_multiplier** -- Files in level 1 will have target_file_size_base bytes. Each next level's file size will be target_file_size_multiplier bigger than previous one. However, by default target_file_size_multiplier is 1, so files in all L1..Lmax levels are equal. Increasing target_file_size_base will reduce total number of database files, which is generally a good thing. We recommend setting target_file_size_base to be `max_bytes_for_level_base / 10`, so that there are 10 files in level 1.

compression_per_level -- Use this option to set different compressions for different levels. It usually makes sense to avoid compressing levels 0 and 1 and to compress data only in higher levels. You can even set slower compression in highest level and faster compression in lower levels (by highest we mean Lmax).

num_levels -- It is safe for num_levels to be bigger than expected number of levels in the database. Some higher levels may be empty, but this will not impact performance in any way. Only change this option if you expect your number of levels will be greater than 7 (default).

Universal Compaction

Write amplification of a level style compaction may be high in some cases. For write-heavy workloads, you may be bottlenecked on disk throughput. To optimize for those workloads, RocksDB introduced a new style of compaction that we call universal compaction, intended to decrease write amplification. However, it may increase read amplification and always increases space amplification. **Universal compaction has a size limitation. Please be careful when your DB (or column family) size is over 100GB.** Check Universal Compaction for details.

With universal compaction, a compaction process may temporarily increase size amplification by a factor of two. In other words, if you store 10GB in database, the compaction process may consume additional 10GB, in addition to space amplification.

However, there are techniques to help reduce the temporary space doubling. If you use universal compaction, we strongly recommend sharding your data and keeping it in multiple RocksDB instances. Let's assume you have S shards. Then, configure Env thread pool with only N compaction threads. Only N shards out of total S shards will have additional space amplification, thus bringing it down to `N/S` instead of 1. For example, if your DB is 10GB and you configure it with 100 shards, each shard will hold 100MB of data. If you configure your thread pool with 20 concurrent compactions, you will only consume extra 2GB of data instead of 10GB. Also, compactions will execute in parallel, which will fully utilize your storage concurrency.

max_size_amplification_percent -- Size amplification as defined by amount of additional storage needed (in percentage) to store a byte of data in the database. Default is 200, which means that a 100 byte database could require up to 300 bytes of storage. 200 bytes of that 300 bytes are temporary and are used only during compaction. Increasing this limit decreases write amplification, but (obviously) increases space amplification.

compression_size_percent -- Percentage of data in the database that is compressed. Older data is compressed, newer data is not compressed. If set to -1 (default), all data is compressed. Reducing compression_size_percent will reduce CPU usage and increase space amplification.

See Universal Compaction page for more information on universal compaction.

Write stalls

See Write Stalls page for more details.

Prefix databases

RocksDB keeps all data sorted and supports ordered iteration. However, some applications don't need the keys to be fully sorted. They are only interested in ordering keys with a common prefix.

Those applications can benefit of configuring prefix_extractor for the database.

prefix_extractor -- A SliceTransform object that defines key prefixes. Key prefixes are then used to perform some interesting optimizations:

1. Define prefix bloom filters, which can reduce read amplification of prefix range queries (e.g., give me all keys that start with prefix `xxx`). Be sure to define **Options::filter_policy**.
2. Use hash-map-based memtables to avoid binary search costs in memtables.
3. Add hash index to table files to avoid binary search costs in table files. For more details on (2) and (3), see Custom memtable and table factories. Please be aware that (1) is usually sufficient in reducing I/Os. (2) and (3) can reduce CPU costs in some use cases and usually with some costs of memory. You should only try it if CPU is your bottleneck and you run out of other easier tuning to save CPU, which is not common. Make sure to check comments about prefix_extractor in `include/rocksdb/options.h`.

Bloom filters

Bloom filters are probabilistic data structures that are used to test whether an element is part of a set. Bloom filters in RocksDB are controlled by an option *filter_policy*. When a user calls Get(key), there is a list of files that may contain the key. This is usually all files on Level 0 and one file from each Level bigger than 0. However, before we read each file, we first consult the bloom filters. Bloom filters will filter out reads for most files that do not contain the key. In most cases, Get() will do only one file read. Bloom filters are always kept in memory for open files, unless `BlockBasedTableOptions::cache_index_and_filter_blocks` is set to true. Number of open files is controlled by `max_open_files` option.

There are two types of bloom filters: block-based and full filter.

Block-based filter (deprecated)

Set up block based filter by calling: `options.filter_policy.reset(rocksdb::NewBloomFilterPolicy(10, true))`. Block-based bloom filter is built separately for each block. On a read we first consult an index, which returns the block of the key we're looking for. Now that we have a block, we consult the bloom filter for that block.

Full filter

Set up full filter by calling: `options.filter_policy.reset(rocksdb::NewBloomFilterPolicy(10, false))`. Full filters are built per-file. There is only one bloom filter for a file. This means we can first consult the bloom filter without going to the index. In situations when key is not in the bloom filter, we saved one index lookup compared to block-based filter.

Full filters could further be partitioned: Partitioned Filters

Custom memtable and table format

Advanced users may configure custom memtable and table format.

memtable_factory -- Defines the memtable. Here's the list of memtables we support:

1. SkipList -- this is the default memtable.
2. HashSkipList -- it only makes sense with prefix_extractor. It keeps keys in buckets based on prefix of the key. Each bucket is a skip list.
3. HashLinkedList -- it only makes sense with prefix_extractor. It keeps keys in buckets based on prefix of the key. Each bucket is a linked list.

table_factory -- Defines the table format. Here's the list of tables we support:

1. Block based -- This is the default table. It is suited for storing data on disk and flash storage. It is addressed and loaded in block sized chunks (see block_size option). Thus the name block based.
2. Plain Table -- Only makes sense with prefix_extractor. It is suited for storing data on memory (on tmpfs filesystem). It is byte-addressible.

Memory usage

To learn more about how RocksDB uses memory, check out this wiki page: https://github.com/facebook/rocksdb/wiki/Memory-usage-in-RocksDB

Difference of spinning disk

Memory / Persistent Storage ratio is usually much lower for databases on spinning disks. If the ratio of data to RAM is too large then you can reduce the memory required to keep performance critical data in

RAM. Suggestions:

- Use relatively **larger block sizes** to reduce index block size. You should use at least 64KB block size. You can consider 256KB or even 512KB. The downside of using large blocks is that RAM is wasted in the block cache.
- Turn on **BlockBasedTableOptions.cache_index_and_filter_blocks=true** as it's very likely you can't fit all index and bloom filters in memory. Even if you can, it's better to set it for safety.
- **enable options.optimize_filters_for_hits** to reduce some bloom filter block size.
- Be careful about whether you have enough memory to keep all bloom filters. If you can't then bloom filters might hurt performance.
- Try to **encode keys as compact as possible**. Shorter keys can reduce index block size.

Spinning disks usually provide much lower random read throughput than flash.

- Set **options.skip_stats_update_on_db_open=true** to speed up DB open time.
- This is a controversial suggestion: use **level-based compaction**, as it is more friendly to reduce reads from disks.
- If you use level-based compaction, use **options.level_compaction_dynamic_level_bytes=true**.
- Set **options.max_file_opening_threads** to a value larger than 1 if the server has multiple disks.

Throughput gap between random read vs. sequential read is much higher in spinning disks. Suggestions:

- Enable RocksDB-level read ahead for compaction inputs: **options.compaction_readahead_size** with **options.new_table_reader_for_compaction_inputs=true**
- Use relatively **large file sizes**. We suggest at least 256MB
- Use relatively larger block sizes

Spinning disks are much larger than flash:

- To avoid too many file descriptors, use larger files. We suggest at least file size of 256MB.
- If you use universal compaction style, don't make single DB size too large, because the full compaction will take a long time and impact performance. You can use more DBs but single DB size is smaller than 500GB.

Example configurations

In this section we will present some RocksDB configurations that we actually run in production.

Prefix database on flash storage

This service uses RocksDB to perform prefix range scans and point lookups. It is running on Flash storage.

```
options.prefix_extractor.reset(new CustomPrefixExtractor());
```

Since the service doesn't need total order iterations (see Prefix databases), we define prefix extractor.

```
rocksdb::BlockBasedTableOptions table_options;
table_options.index_type = rocksdb::BlockBasedTableOptions::kHashSearch;
table_options.block_size = 4 * 1024;
options.table_factory.reset(NewBlockBasedTableFactory(table_options));
```

We use a hash index in table files to speed up prefix lookup, but it increases storage space and memory usage.

```
options.compression = rocksdb::kLZ4Compression;
```

LZ4 compression reduces CPU usage, but increases storage space.

```
options.max_open_files = -1;
```

This setting disables looking up files in table cache, thus speeding up all queries. This is always a good thing to set if your server has a big limit on open files.

```
options.options.compaction_style = kCompactionStyleLevel;
options.level0_file_num_compaction_trigger = 10;
options.level0_slowdown_writes_trigger = 20;
options.level0_stop_writes_trigger = 40;
options.write_buffer_size = 64 * 1024 * 1024;
options.target_file_size_base = 64 * 1024 * 1024;
options.max_bytes_for_level_base = 512 * 1024 * 1024;
```

We use level style compaction. Memtable size is 64MB and is flushed periodically to Level 0. Compaction L0->L1 is triggered when there are 10 level 0 files (total 640MB). When L0 is 640MB, compaction is triggered into L1, the max size of which is 512MB. Total DB size???

```
options.max_background_compactions = 1
options.max_background_flushes = 1
```

There can be only 1 concurrent compaction and 1 flush executing at any given time. However, there are multiple shards in the system, so multiple compactions occur on different shards. Otherwise, storage wouldn't be saturated with only 2 threads writing to storage.

```
options.memtable_prefix_bloom_bits = 1024 * 1024 * 8;
```

With memtable bloom filter, some accesses to the memtable can be avoided.

```
options.block_cache = rocksdb::NewLRUCache(512 * 1024 * 1024, 8);
```

Block cache is configured to be 512MB. (is it shared across the shards?)

Total ordered database, flash storage

This database performs both Get() and total order iteration. Shards????

```
options.env->SetBackgroundThreads(4);
```

We first set a total of 4 threads in the thread pool.

```
options.options.compaction_style = kCompactionStyleLevel;
options.write_buffer_size = 67108864; // 64MB
options.max_write_buffer_number = 3;
options.target_file_size_base = 67108864; // 64MB
options.max_background_compactions = 4;
options.level0_file_num_compaction_trigger = 8;
options.level0_slowdown_writes_trigger = 17;
options.level0_stop_writes_trigger = 24;
options.num_levels = 4;
options.max_bytes_for_level_base = 536870912; // 512MB
options.max_bytes_for_level_multiplier = 8;
```

We use level style compaction with high concurrency. Memtable size is 64MB and the total number of level 0 files is 8. This means compaction is triggered when L0 size grows to 512MB. L1 size is 512MB and every level is 8 times larger than the previous one. L2 is 4GB and L3 is 32GB.

Database on Spinning Disks

Coming soon...

In-memory database with full functionalities

In this example, database is mounted in tmpfs file system.

Use mmap read:

options.allow_mmap_reads = true;

Disable block cache, enable bloom fitlers and reduce the delta encoding restart interval:

```
BlockBasedTableOptions table_options;
table_options.filter_policy.reset(NewBloomFilterPolicy(10, true));
table_options.no_block_cache = true;
table_options.block_restart_interval = 4;
options.table_factory.reset(NewBlockBasedTableFactory(table_options));
```

If you want to prioritize speed. You can disable compression:

```
options.compression = rocksdb::CompressionType::kNoCompression;
```

Otherwise, enable a lightweight compression, LZ4 or Snappy.

Set up compression more aggressively and allocate more threads for flush and compaction:

```
options.level0_file_num_compaction_trigger = 1;
options.max_background_flushes = 8;
options.max_background_compactions = 8;
options.max_subcompactions = 4;
```

Keep all the files open:

```
options.max_open_files = -1;
```

When reading data, consider to turn ReadOptions.verify_checksums = false.

In-memory prefix database

In this example, database is mounted in tmpfs file system. We use customized formats to speed up, while some functionalities are not supported. We support only Get() and prefix range scans. Write-ahead logs are stored on hard drive to avoid consuming memory not used for querying. Prev() is not supported.

Since this database is in-memory, we don't care about write amplification. We do, however, care a lot about read amplification and space amplification. This is an interesting example because we tune the compaction to an extreme so that usually only one SST table exists in the system. We therefore decrease read and space amplification, while write amplification is extremely high.

Since universal compaction is used, we will effectively double our space usage during compaction. This is very dangerous with in-memory database. We therefore shard the data into 400 RocksDB instances. We allow only two concurrent compactions, so only two shards may double space use at any one time.

In this case, prefix hash can be used to allow the system to use hash indexing instead of a binary one, as well as bloom filter for iterations when possible:

```
options.prefix_extractor.reset(new CustomPrefixExtractor());
```

Use the memory addressing table format built for low-latency access, which requires mmap read mode to be on:

```
options.table_factory = std::shared_ptr<rocksdb::TableFactory>
(rocksdb::NewPlainTableFactory(0, 8, 0.85));
options.allow_mmap_reads = true;
options.allow_mmap_writes = false;
```

Use hash link list memtable to change binary search to hash lookup in mem table:

```
options.memtable_factory.reset(rocksdb::NewHashLinkListRepFactory(200000));
```

Enable bloom filter for hash table to reduce memory accesses (usually means CPU cache misses) when reading from mem table to one, for the case where key is not found in mem tables:

```
options.memtable_prefix_bloom_bits = 10000000;
options.memtable_prefix_bloom_probes = 6;
```

Tune compaction so that, a full compaction is kicked off as soon as we have two files. We hack the parameter of universal compaction:

```
options.compaction_style = kUniversalCompaction;
options.compaction_options_universal.size_ratio = 10;
options.compaction_options_universal.min_merge_width = 2;
options.compaction_options_universal.max_size_amplification_percent = 1;
options.level0_file_num_compaction_trigger = 1;
options.level0_slowdown_writes_trigger = 8;
options.level0_stop_writes_trigger = 16;
```

Tune bloom filter to minimize memory accesses:

```
options.bloom_locality = 1;
```

Reader objects for all tables are always cached, avoiding table cache access when reading:

```
options.max_open_files = -1;
```

Use one mem table at one time. Its size is determined by the full compaction interval we want to pay. We tune compaction such that after every flush, a full compaction will be triggered, which costs CPU. The larger the mem table size, the longer the compaction interval will be, and at the same time, we see less memory efficiency, worse query performance and longer recovery time when restarting the DB.

```
options.write_buffer_size = 32 << 20;
options.max_write_buffer_number = 2;
options.min_write_buffer_number_to_merge = 1;
```

Multiple DBs sharing the same compaction pool of 2:

```
options.max_background_compactions = 1;
options.max_background_flushes = 1;
options.env->SetBackgroundThreads(1, rocksdb::Env::Priority::HIGH);
options.env->SetBackgroundThreads(2, rocksdb::Env::Priority::LOW);
```

Settings for WAL logs:

```
options.bytes_per_sync = 2 << 20;
```

Suggestion for in memory block table

hash_index: In the new version, hash index is enabled for block based table. It would use 5% more storage space but speed up the random read by 50% compared to normal binary search index.

```
table_options.index_type = rocksdb::BlockBasedTableOptions::kHashSearch;
```

block_size: By default, this value is set to be 4k. If compression is enabled, a smaller block size would lead to higher random read speed because decompression overhead is reduced. But the block size cannot be too small to make compression useless. It is recommended to set it to be 1k.

verify_checksum: As we are storing data in tmpfs and care read performance a lot, checksum could be disabled.

Final thoughts

Unfortunately, configuring RocksDB optimally is not trivial. Even we as RocksDB developers don't fully understand the effect of each configuration change. If you want to fully optimize RocksDB for your workload, we recommend experiments and benchmarking, while keeping an eye on the three amplification factors. Also, please don't hesitate to ask us for help on the RocksDB Developer's Discussion Group.

Clone this wiki locally

Clone this wiki locally

https://github.com/facebook/rocksdb.wiki.git