# RocksDB Setup Options and Basic Tuning

From: <https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning>

Besides writing code using [Basic Operations](https://github.com/facebook/rocksdb/wiki/Basic-Operations) on RocksDB, you may also be interested in how to tune RocksDB to achieve desired performance. In this page, we introduce how to get an initial set-up, which should work well enough for many use cases.

RocksDB has many configuration options, but most of them can be safely ignored by many users, as the majority of them are for influencing the performance of very specific workloads. For general use, most RocksDB options can be left at their defaults, however, we suggest some options below that every user might like to experiment with for general workloads.

First, you need to think about the options relating to resource limits (see also: [Basic Operations](https://github.com/facebook/rocksdb/wiki/Basic-Operations)):

## [Write Buffer Size](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#write-buffer-size)

This can be set either per Database and/or per Column Family.

### [Column Family Write Buffer Size](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#column-family-write-buffer-size)

This is the maximum write buffer size used for the Column Family.

It represents the amount of data to build up in memory (backed by an unsorted log on disk) before converting to a sorted on-disk file. The default is 64 MB.

You need to budget for 2 x your worst case memory use. If you don't have enough memory for this, you should reduce this value. Otherwise, it is not recommended to change this option. For example:

cf\_options.write\_buffer\_size = 64 << 20;

See below for sharing memory across Column Families.

### [Database Write Buffer Size](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#database-write-buffer-size)

This is the maximum size of all Write Buffers across all Column Families in the database. It represents the amount of data to build up in memtables across all column families before writing to disk.

By default this feature is disabled (by being set to 0). You should not need to change it. However, for reference, if you do need to change it to 64 GB for example:

db\_options.db\_write\_buffer\_size = 64 << 30;

## [Block Cache Size](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#block-cache-size)

You can create a Block cache of your chosen the size for caching uncompressed data.

We recommend that this should be about 1/3 of your total memory budget. The remaining free memory can be left for the OS (Operating System) page cache. Leaving a large chunk of memory for OS page cache has the benefit of avoiding tight memory budgeting (see also: [Memory Usage in RocksDB](https://github.com/facebook/rocksdb/wiki/Memory-usage-in-RocksDB)).

Setting the block cache size requires that we also set table related options, for example if you want an LRU Cache of 128 MB:

auto cache = NewLRUCache(128 << 20);

BlockBasedTableOptions table\_options;

table\_options.block\_cache = cache;

auto table\_factory = new BlockBasedTableFactory(table\_options);

cf\_options.table\_factory.reset(table\_factory);

NOTE: You should set the same Cache object on all the table\_options for all the Column Families of all DB's managed by the process. An alternative to achieve this, is to pass the same table\_factory or table\_options to all Column Families of all DB's. To learn more about the Block Cache, see: [Block Cache](https://github.com/facebook/rocksdb/wiki/Block-Cache).

## [Compression](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#compression)

You can only choose compression types which are supported on your host system. Using compression is a trade-off between CPU, I/O and storage space.

1. cf\_options.compression controls the compression type used for the first n-1 levels. We recommend to use LZ4 (kLZ4Compression), or if not available, to use Snappy (kSnappyCompression).
2. cf\_options.bottommost\_compression controls the compression type used for the nth level. We recommend to use ZStandard (kZSTD), or if not available, to use Zlib (kZlibCompression).

To learn more about compression, See [Compression](https://github.com/facebook/rocksdb/wiki/Compression).

## [Bloom Filters](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#bloom-filters)

You should only enable this if it suits your Query patterns; If you have many point lookup operations (i.e. Get()), then a Bloom Filter can help speed up those operations, conversely if most of your operations are range scans (e.g. Iterator()) then the Bloom Filter will not help.

The Bloom Filter uses a number of bits for each key, a good value is 10, which yields a filter with ~1% false positive rate. Refer to [Bloom Filter Wiki page](https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter) for false positive rates with lower number of bits per key.

If Get() is a common operation for your queries, you can configure the Bloom Filter, for example with 10 bits per key:

table\_options.filter\_policy.reset(NewBloomFilterPolicy(10, false));

We also recommend

table\_options.optimize\_filters\_for\_memory = true;

for some memory saving.

To learn more about Bloom Filters, see: [Bloom Filter](https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter).

## [Rate Limits](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#rate-limits)

It can be a good idea to limit the rate of compactions and flushes to smooth I/O operations, one reason for doing this is to avoid the read latency outliers. This can be done by means of the db\_options.rate\_limiter option. Rate limiting is a complex topic, and is covered in [Rate Limiter](https://github.com/facebook/rocksdb/wiki/Rate-Limiter).

**NOTE**: Make sure to pass the same rate\_limiter object to all the DB's in your process.

## [SST File Manager](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#sst-file-manager)

If you are using flash storage, we recommend users to mount the file system with the [discard](http://man7.org/linux/man-pages/man8/mount.8.html) flag in order to improve write amplification.

If you are using flash storage and the discard flag, trimming will be employed. Trimming can cause long I/O latencies temporarily if the trim size is very large. The SST File Manager can cap the file deletion speed, so that each trim's size is controlled.

The SST File Manager can be enabled, by setting the db\_options.sst\_file\_manager option. Details of the SST File Manager can be seen here: [sst\_file\_manager\_impl.h](https://github.com/facebook/rocksdb/blob/5.14.fb/util/sst_file_manager_impl.h" \l "L28).

## [Other General Options](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#other-general-options)

Below are a number of options, where we feel the values set achieve reasonable out-of-box performance for general workloads. We didn't change these options because of the concern of incompatibility or regression when users upgrade their existing RocksDB instance to a newer version. We suggest that users start their new DB projects with these settings:

cf\_options.level\_compaction\_dynamic\_level\_bytes = true;

opts.max\_background\_jobs = 6;

options.bytes\_per\_sync = 1048576;

options.compaction\_pri = kMinOverlappingRatio;

table\_options.block\_size = 16 \* 1024;

table\_options.cache\_index\_and\_filter\_blocks = true;

table\_options.pin\_l0\_filter\_and\_index\_blocks\_in\_cache = true;

table\_options.format\_version = <the latest version>;

Default format version usually lags behind the recommended version for compatibility reason. For new use cases, it's not a concern.

Don't feel sad if you have existing services running with the defaults instead of these options. Whilst we believe that these are better than the default options, none of them is likely to bring significant improvements.

## [Conclusion and Further Reading](https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#conclusion-and-further-reading)

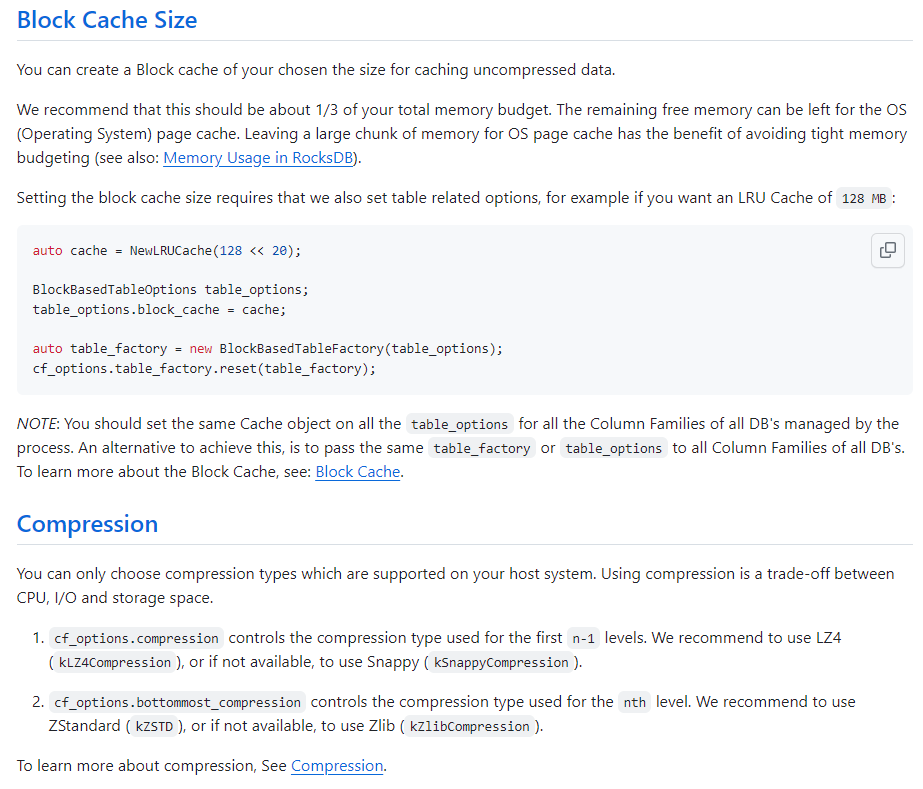
Now you are ready to test your application and see how your initial RocksDB performance looks. Hopefully it will be good enough!

If the performance of RocksDB within your application after the basic set-up described above, is good enough for you, we don't recommend that you tune it further. As it is common for a workload to change over time, if you expend unnecessary resources upfront to tune RocksDB to be highly performant for your current workload, some modest change in future to that workload may push the performance off a cliff.

On the other hand, if the performance is not good enough for you, you can further tune RocksDB by following the more detailed [Tuning Guide](https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide).

# RocksDB Block Size相关配置项目

From: https://github.com/facebook/rocksdb/wiki/Basic-Operations#block-size



## [Block size](https://github.com/facebook/rocksdb/wiki/Basic-Operations#block-size)

rocksdb groups adjacent keys together into the same block and such a block is the unit of transfer to and from persistent storage. The default block size is approximately 4096 uncompressed bytes. Applications that mostly do bulk scans over the contents of the database may wish to increase this size. Applications that do a lot of point reads of small values may wish to switch to a smaller block size if performance measurements indicate an improvement. There isn't much benefit in using blocks smaller than one kilobyte, or larger than a few megabytes. Also note that compression will be more effective with larger block sizes. To change block size parameter, use Options::block\_size.

## [Write buffer](https://github.com/facebook/rocksdb/wiki/Basic-Operations#write-buffer)

Options::write\_buffer\_size specifies the amount of data to build up in memory before converting to a sorted on-disk file. Larger values increase performance, especially during bulk loads. Up to max\_write\_buffer\_number write buffers may be held in memory at the same time, so you may wish to adjust this parameter to control memory usage. Also, a larger write buffer will result in a longer recovery time the next time the database is opened.

Related option is Options::max\_write\_buffer\_number, which is maximum number of write buffers that are built up in memory. The default is 2, so that when 1 write buffer is being flushed to storage, new writes can continue to the other write buffer. The flush operation is executed in a [Thread Pool](https://github.com/facebook/rocksdb/wiki/Thread-Pool).

Options::min\_write\_buffer\_number\_to\_merge is the minimum number of write buffers that will be merged together before writing to storage. If set to 1, then all write buffers are flushed to L0 as individual files and this increases read amplification because a get request has to check all of these files. Also, an in-memory merge may result in writing lesser data to storage if there are duplicate records in each of these individual write buffers. Default: 1

## [Compression](https://github.com/facebook/rocksdb/wiki/Basic-Operations#compression)

Each block is individually compressed before being written to persistent storage. Compression is on by default since the default compression method is very fast, and is automatically disabled for uncompressible data. In rare cases, applications may want to disable compression entirely, but should only do so if benchmarks show a performance improvement:

rocksdb::Options options;

options.compression = rocksdb::kNoCompression;

... rocksdb::DB::Open(options, name, ...) ....

Also [Dictionary Compression](https://github.com/facebook/rocksdb/wiki/Dictionary-Compression) is also available.

## [Cache](https://github.com/facebook/rocksdb/wiki/Basic-Operations#cache)

The contents of the database are stored in a set of files in the filesystem and each file stores a sequence of compressed blocks. If options.block\_cache is non-NULL, it is used to cache frequently used uncompressed block contents. We use operating systems file cache to cache our raw data, which is compressed. So file cache acts as a cache for compressed data.

#include "rocksdb/cache.h"

rocksdb::BlockBasedTableOptions table\_options;

table\_options.block\_cache = rocksdb::NewLRUCache(100 \* 1048576); // 100MB uncompressed cache

rocksdb::Options options;

options.table\_factory.reset(rocksdb::NewBlockBasedTableFactory(table\_options));

rocksdb::DB\* db;

rocksdb::DB::Open(options, name, &db);

... use the db ...

delete db

When performing a bulk read, the application may wish to disable caching so that the data processed by the bulk read does not end up displacing most of the cached contents. A per-iterator option can be used to achieve this:

rocksdb::ReadOptions options;

options.fill\_cache = false;

rocksdb::Iterator\* it = db->NewIterator(options);

for (it->SeekToFirst(); it->Valid(); it->Next()) {

...

}

You can also disable block cache by setting options.no\_block\_cache to true.

See [Block Cache](https://github.com/facebook/rocksdb/wiki/Block-Cache) for more details.

## RocksDB默认配置值可以参考代码：<https://github.com/facebook/rocksdb/blob/main/include/rocksdb/table.h#L129>

## RocksDB Block Cache

From: https://github.com/facebook/rocksdb/wiki/Block-Cache

Block cache is where RocksDB caches data in memory for reads. User can pass in a Cache object to a RocksDB instance with a desired capacity (size). A Cache object can be shared by multiple RocksDB instances in the same process, allowing users to control the overall cache capacity. The block cache stores uncompressed blocks. Optionally user can set a second block cache storing compressed blocks. Reads will fetch data blocks first from uncompressed block cache, then compressed block cache. The compressed block cache can be a replacement of OS page cache, if [Direct-IO](https://github.com/facebook/rocksdb/wiki/Direct-IO) is used.

There are two cache implementations in RocksDB, namely LRUCache and ClockCache. Both types of the cache are sharded to mitigate lock contention. Capacity is divided evenly to each shard and shards don't share capacity. By default each cache will be sharded into at most 64 shards, which provides a reasonable balance of scalability with parallel reads and locality of block cache metadata with lighter read loads. The minimum capacity for cache shards is 512KB, to limit the risk of random

# Tuning RocksDB on Spinning Disks

From: <https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-Disks>

Spinning disks are different for RocksDB, for some main reasons:

##### [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:](https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-Disks#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.](https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-Disks#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:](https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-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:](https://github.com/facebook/rocksdb/wiki/Tuning-RocksDB-on-Spinning-Disks#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.