November 2, 2022 Bosmat Tuvel 7 Minutes read

How does RocksDB Memory Management work?

RocksDB is a versatile, embeddable persistent key-value store designed for fast storage including memory (RAM), flash, SSD, and NVMe. It is popular because of its performance capabilities and versatility which are due in large part to how RocksDB memory management works and the ability to tune its many parameters and options.
Before we jump to tuning RocksDB it's important to first understand how RocksDB memory management works. The first thing we should start with are some key terms we use for RocksDB as we look at how memory management and the data read/write flow works.

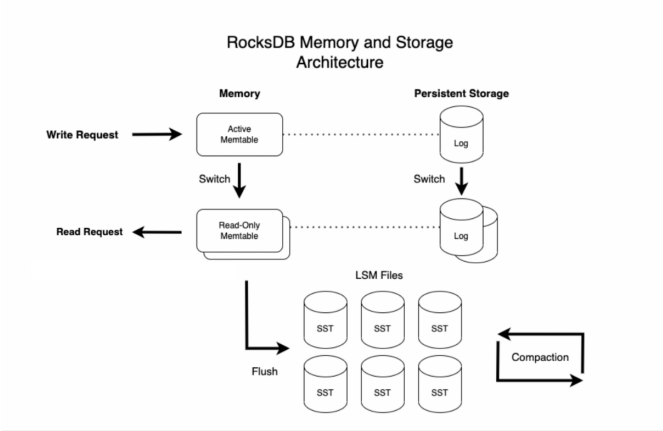
A Quick RocksDB Glossary of Terms

You may already know some of this but it's helpful to remind ourselves about the fundamentals of RocksDB that we will see as we learn about RocksDB memory management.
Memtable - an in-memory data structure containing the most recent updates to your database and typically in sorted order with a binary searchable index.
SST file - Static Sorted Table (sometimes called SSTFile) files containing persistent data stored in a sorted order with a binary searchable index.

Write-Ahead Log (WAL) - a transaction log file that contains data not yet flushed to SST files and can be used in the event of failure during a DB recovery process.
Block Cache - an in-memory data structure that contains hot blocks that have been read from SST files.
Compaction - a background job that merges SST files to optimize data organization for more efficient reads and storage.
Bloom filters - when a filter policy is set, every SST file includes a bloom filter block which will determine if a key may be located in the SST file which makes queries much more efficient and reduces the need for high amounts of I/O.
Now that we have a couple of terms down, let's jump into why it all matters.

RocksDB 101 - How does RocksDB manage data?

The diagram below illustrates a high-level view of the architecture of RocksDB. There are three key elements to the data flow which are in-memory data structure (memtable), persistent storage (transaction log files), and the LSM files (SSTable).



RocksDB has been designed for versatility and performance because it needed to support a diverse set of data and application patterns. It is purpose-built to maximize the use of extremely

high-speed memory and storage, with highly-tunable ways to manage data, index, and query behaviors.
Memory is where the lowest latency writes, and reads occur, so being able to maximize the use of memory to store active key-value pairs leads to fast read/query performance and to move data in and out of memory to file storage for persistence and to free up memory for active data.
Let's track the write pipeline as new data comes into RocksDB from your application to understand how RocksDB inserts new data to the database and where memory is a critical part of the process.

1. A write request comes from the application to RocksDB and is written to the database in-memory (memtable) which is sometimes called the active memtable (writable)
2. An optional step is to write to a transaction log called the Write-Ahead Log (WAL) to add resiliency in case we lose the data in memory
3. When the active memtable becomes full it becomes a read-only memtable
4. Read-only memtrables are flushed from memory to SST files (SSTable)
5. SST files are compacted to optimize the space needed on persistent storage.

It looks like 5 simple steps except each step actually contains many tunable parameters to choose how RocksDB manages data movement, memory usage, and storage. Memory speed is important here because we have only a limited amount of data in the active memtable for efficient high-speed writes. Bloom filters are added to make key searching more efficient and faster both in-memory and down to the SST files on flash but as data is written and read, RocksDB uses the block cache as a hot tier for the fastest possible retrieval of hot data.

Column Family versus Multiple RocksDB Database Instances

There's one more thing to consider before we dive further into RocksDB memory management and the write buffer. Your data schema and different data types will lead you to make an architectural decision: should you use column families or multiple RocksDB database instances?

RocksDB column families let you logically partition your database. Column families also have interesting performance and operational advantages over using multiple RocksDB databases. Each column family can be configured differently (including compaction settings, compaction filters, merge operators, compression settings, and bloom filters among other options. Column families also share the Write-Ahead Log (WAL) but do not share the same memtable and SST files so you can achieve better performance.
Here are a few reasons column families are advantageous:

- Each column family can have its own block cache but may also be shared between column families or multiple database instances
- Each column family has its own memtable
- Each column family has its own bloom filters
- Atomic writes across column families within the database
- Additional tuning and filter options

You may choose multiple RocksDB data instances if you need:

- A database is the unit of backups and checkpoints if you want to copy to another host easier than copying/replicating column families
- Create read-only, consistent copies for other processes

You can see that column families create a lot of flexibility for your data schema and the ability to do more granular operational tuning based on your design.

How does RocksDB Write Buffer Manager Handle Memory?

The write buffer manager controls how much memory is used by memtables. This will be true in either multiple column families or multiple database instances. The goal of this is to limit how

much memory is used in the system. There are multiple parameters you can tune for the RocksDB write buffer manager that will change the behavior for memory caching and flushing.
RocksDB is also loading indexes and bloom filters into memory and managing putting uncompressed data into block cache. These are loaded cumulatively in memory which is very fast, but also adds other risks when it comes to planning memory allocation.
It may seem counterintuitive to suggest limiting memory usage, but you may find that memory usage and your application behavior can uncover other real-time issues such as CPU and I/O congestion. Let's explore why that can occur.

How Tuning RocksDB Memory Management Can Affect Performance?

Memory management has a direct impact on the performance and durability of the system running your application. Memory management in RocksDB handles initial writes to memory at the same time data is written to the transaction log, plus read/query data being moved into block cache. Making choices about when to flush data from memtable to SST will affect CPU and memory performance during flush and compaction operations.
Many RocksDB tuning options address memory challenges but have a secondary effect on overall system performance. The biggest challenge is that any tuning you do for one application may not prove to be as effective on another application.
Performance will vary drastically depending on application read/write ratios, number of inserts, number of updates, hardware-specific performance, and the data schema you choose, and type and size of values stored in the database.
RocksDB is managing memtable, block cache, indexes and bloom filters, and interacting with the operating system for resources. There may also be congestion because of sharing CPU and memory resources with the rest of the system.
Application and overall system performance also changes over time with utilization patterns that fluctuate. Most importantly, as your application scales, issues with memory management are

amplified. So an application that begins with nominal memory usage may grow to store larger objects or just more data.

How Do You Resolve RocksDB Memory Management Issues?

Now you are probably asking how you can resolve RocksDB memory management issues in production. It's pretty clear that we can have many potential problems that may come from the RocksDB configuration and also how the application reads and writes data to RocksDB.
There are many mitigation strategies for these and other performance challenges at scale. These configuration workarounds can include adjusting memtable sizes, tuning compaction, modifying application code, changing physical configuration of storage, and offloading processes like the recent experimental introduction of remote compaction.
Results will vary depending on all of these factors and also your individual application usage pattern.

Control write buffer size

Setting your write buffer memtable parameters lets you control the amount of total memory available to RocksDB for storing active and read-only memtable files before flushing to SST. Here is an example parameter set that can define the write buffer configuration including file size and number of files:
write_buffer_size = 256MB;
max_write_buffer_number = 4;
min_write_buffer_number_to_merge = 2;
You can adjust this to match to the size of data being inserted and rate of inserts to try to find an optimal memory configuration for writes and reads/queries based on your application needs.
The trade off of memory usage to write performance is important here. The more memory you allocate, the less data is written to SST. Finding the optimal setting is workload-specific and will

vary as the system (and application) scales. There are also considerations about SST level sizing as you make changes to write buffer configuration.

Control block cache size

RocksDB uses block cache for storing uncompressed data in memory for reads. Compressed data is handled by OS cache by default or you can configure a secondary block cache for compressed data as well. Optimizing block cache can speed up reads but also has risks depending on many application factors.
For example, block cache might not be as effective for highly randomized reads, so your application usage pattern will define how well block cache can affect overall performance. Adjusting block cache also has a CPU impact trade off, so take care as you think of memory as a standalone resource.

Monitor both memory and persistent storage performance

You need to continuously monitor for latency to commit writes to the filesystem. For example, in the options.h configuration file there is an *allow_stall* option that can be set to enable stalling of all writers when memory usage exceeds the *buffer_size* limit. This forces a wait for flush SST to complete and memory usage to drop down.
You can configure your applications to avoid it by setting an option for *no_slowdown = true* in the WriteOptions. The default setting is false which means the write behavior is to wait/sleep for the write completion. By setting the no_slowdown to true it will cancel the write and send back an *Status::Incomplete()* result.
If you find yourself configuring settings to manage write stalls it's often an indicator that other configuration parameters for memory usage and performance need attention. Solving the problem in memory by forcing write stalls may be hiding another issue.

Monitor cache hit rates

Identifying how your application and RocksDB are actually optimizing performance can be done by monitoring overall cache hit rates. Monitoring cache statistics will let you narrow down where clean cache hits are happening, where miss ratios are high, and give you hints for tuning block cache and memtable configuration.
Here's an example of using *perf* to check the RocksDB statistics:

```
TEST_TMPDIR=/dev/shm/ perf record -g ./db_bench -statistics -use_existing_db=true -benchmarks=readrandom -threads=32 -cache_size=1048576000 -num=1000000 -reads=1000000 && perf report -g --children
```

Here is the output snippet for the cache-misses event showing the percentage of missed ratio for cache queries:

```
Children  Self   Command   Shared Object  Symbol
+  19.54% 19.50% db_bench  db_bench    [.] rocksdb::StatisticsImpl::recordTick
+   3.44%  0.57% db_bench  db_bench    [.] rocksdb::StatisticsImpl::measureTime
```

This helps you to understand cache behaviors based on read patterns of the application at any point in time and can guide how you tune your RocksDB instance.

Why RocksDB Configuration Workarounds Degrade Memory Performance at Scale

You can see both the flexibility, and the complexity in finding an optimal memory management configuration for RocksDB. Every option has trade offs that will be unique to your application and data pattern. Issues become amplified as the system and your data scales.
Here are a few important challenges that occur and will increase at scale:

1. Memory is shared by multiple processes and also depends on persistent storage and processor access for many operations - congestion increases as usage increases
2. Moving more data in and out of the block cache increases overhead and can bring down system performance.
3. Increasing SST Files also means more bloom filters are loaded in memory and additional CPU overhead - balancing speed and efficiency for queries will impact the system memory and other processes.
4. Background processes (e.g. flushing to SST, compaction) also use system resources which may compete with active memory used by memtable.
5. Data architecture and operational processes (e.g. shared WAL, backups, snapshots) will affect overall memory usage and performance.
6. I/O hangs are more prevalent as database size increases and more data movement in and out of memory creates system congestion.
7. SSD lifespan will be impacted as data size and I/O increases because of how writes and rewrites impact how SSDs operate.

Scaling any system will change the behavior and performance. RocksDB is designed for fast memory and storage but is not immune to configuration workarounds leading to other bottlenecks.

Conclusion

The good news is that there are many tunable options available in RocksDB for memory management. The bad news is that there are many tunable options in RocksDB that you can't know the impact of until you make a change in a live system.
RocksDB has many configuration options for effective, efficient memory management. The issue with out-of-the-box RocksDB memory management is that it will require tuning and adjustment based on your production database usage.

The measurement and tuning must be done regularly because as the system scales you will find performance changes. This may also happen just because of changes to your application code and how you choose to manage key-value data and the schema.

As with many things in systems architecture, when asked “how can we optimize?”, the answer is “it depends”. With RocksDB configuration options you have a lot of flexibility, but also a multi-dimensional challenge when finding the optimal configuration.

 # RocksDB **# Memory-Management** **# Key-Value Storage Engine**

 # application performance **# Resource utilization effeciency**