分享

RocksDB中文网

- 开发文档
- Q群 609160979
- GitHub
- API (C++)
- API (Java)

- 开发文档
- GitHub
- API (C++)
- API (Java)

- RocksDB Wiki
- Overview
- RocksDB FAQ
- Terminology
- **Developer's Guide**
  - Basic Operations
    - Iterator
    - Prefix seek
    - SeekForPrev

        - Tailing Iterator
        - Compaction Filter
        - Read-Modify-Write Operator
        - Column Families
        - Creating and Ingesting SST files
        - Single Delete
        - Low Priority Write
        - Time to Live (TTL) Support
        - Transactions
    - Known Issues
    - Compatibility Between Releases
    - Options
        - Set-Up Options
        - Option String and Option Map
        - RocksDB Options File
    - Compression
        - Dictionary Compression
    - IO
        - Rate Limiter
        - Direct I/O
    - MANIFEST
    - Block Cache
    - MemTable
    - Huge Page TLB Support
    - Write Ahead Log
        - Write Ahead Log File Format
        - WAL Recovery Modes
    - Write Buffer Manager
    - Compaction
        - Leveled Compaction
        - Universal compaction style
        - FIFO compaction style
        - Manual Compaction
    - SST File Formats
        - Block-based Table Format
        - PlainTable Format
        - Bloom Filter
    - Logging and Monitoring
        - Logger
        - Statistics
        - Perf Context and IO Stats Context
        - EventListener

- **Tools / Utilities**

    - Administration and Data Access Tool
    - Benchmarking Tools
    - How to Backup RocksDB?
    - Checkpoints
    - How to persist in-memory RocksDB database
    - Stress Test
    - Third-party language bindings

- **Implementation Details**

  - Partitioned Index/Filters
  - WritePrepared-Transactions
  - How we keep track of live SST files
  - How we index SST
  - Merge Operator Implementation
  - Choose Level Compaction Files
  - RocksDB Repairer
  - Two Phase Commit
  - Iterator's Implementation
  - Simulation Cache
  - Persistent Read Cache

- **RocksJava**

  - RocksJava Basics
  - RocksJava Performance on Flash Storage
  - JNI Debugging
  - RocksJava API TODO

- **Lua**

  - Lua CompactionFilter

- **Performance**

  - Performance on Flash Storage
  - In Memory Workload Performance
  - Read-Modify-Write Performance
  - Delete A Range Of Keys
  - Write Stalls
  - Pipelined Write
  - Tuning Guide
  - Memory usage in RocksDB
  - Speed-Up DB Open
  - Implement Queue Service Using RocksDB

- **Misc**

  - Building on Windows
  - Open Projects
  - Talks
  - Publication
  - Features Not in LevelDB
  - How to ask a performance-related question?
  - Articles about Rocks

- Basic Operations

  - Iterator
  - Prefix seek
  - SeekForPrev
  - Tailing Iterator
  - Compaction Filter
  - Read-Modify-Write Operator
  - Column Families

- Read-Modify-Write Performance
- Delete A Range Of Keys
- Write Stalls
- Pipelined Write
- Tuning Guide
- Memory usage in RocksDB
- Speed-Up DB Open
- Implement Queue Service Using RocksDB
- Building on Windows
- Open Projects
- Talks
- Publication
- Features Not in LevelDB
- How to ask a performance-related question?
- Articles about Rocks

Here we try to explain how RocksDB uses memory. There are a couple of components in RocksDB that contribute to memory usage:

1. Block cache
2. Indexes and bloom filters
3. Memtables
4. Blocks pinned by iterators

We will describe each of them in turn.

Block cache

rocksdb_cache_size = 128M

Block cache is where RocksDB caches uncompressed data blocks. You can configure block cache's size by setting block_cache property of BlockBasedTableOptions:

```
rocksdb::BlockBasedTableOptions table_options;
table_options.block_cache = rocksdb::NewLRUCache(1 * 1024 * 1024 * 1024LL);
rocksdb::Options options;
options.table_factory.reset(new rocksdb::BlockBasedTableFactory(table_options));
```

If the data block is not found in block cache, RocksDB reads it from file using buffered IO. That means it also uses page cache -- it contains raw compressed blocks. In a way, RocksDB's cache is two-tiered: block cache and page cache. Unintuitively, decreasing block cache size will not increase IO. The memory saved will likely be used for page cache, so even more data will be cached. However, CPU usage might grow because RocksDB needs to decompress pages it reads from page cache.

To learn how much memory is block cache using, you can call a function GetUsage() on block cache object:

```
table_options.block_cache->GetUsage();
```

In MongoRocks, you can get the size of block cache by calling

```
> db.serverStatus()["rocksdb"]["block-cache-usage"]
```

Indexes and filter blocks

Indexes and filter blocks can be big memory users and by default they don't count in memory you allocate for block cache. This can sometimes cause confusion for users: you allocate 10GB for block cache, but RocksDB is using 15GB of memory. The difference is usually explained by index and bloom filter blocks.

Here's how you can roughly calculate and manage sizes of index and filter blocks:

1. For each data block we store three information in the index: a key, a offset and size. Therefore, there are two ways you can reduce the size of the index. If you increase block size, the number of blocks will decrease, so the index size will also reduce linearly. By default our block size is 4KB, although we usually run with 16-32KB in production. The second way to reduce the index size is the reduce key size, although that might not be an option for some use-cases.

   rocksdb_metadata_block_size = 4096

2. Calculating the size of filter blocks is easy. If you configure bloom filters with 10 bits per key (default, which gives 1% of false positives), the bloom filter size is `number_of_keys * 10 bits`. There's one trick you can play here, though. If you're certain that Get() will mostly find a key you're looking for, you can set `options.optimize_filters_for_hits = true`. With this option turned on, we will not build bloom filters on the last level, which contains 90% of the database. Thus, the memory usage for bloom filters will be 10X less. You will pay one IO for each Get() that doesn't find data in the database, though.

   options.optimize_filters_for_hits

There are two options that configure how much index and filter blocks we fit in memory:

rocksdb_cache_index_and_filter_blocks = true

1. If you set `cache_index_and_filter_blocks` to true, index and filter blocks will be stored in block cache, together with all other data blocks. This also means they can be paged out. If your access pattern is very local (i.e. you have some very cold key ranges), this setting might make sense. However, in most cases it will hurt your performance, since you need to have index and filter to access a certain file. Always consider to set `pin_l0_filter_and_index_blocks_in_cache` too to minimize the performance impact.

   pin_l0_filter_and_index_blocks_in_cache = true

2. If `cache_index_and_filter_blocks` is false (which is default), the number of index/filter blocks is controlled by option `max_open_files`. If you are certain that your ulimit will always be bigger than number of files in the database, we recommend setting `max_open_files` to -1, which means infinity. This option will preload all filter and index blocks and will not need to maintain LRU of files. Setting `max_open_files` to -1 will get you the best possible performance.

To learn how much memory is being used by index and filter blocks, you can use RocksDB's GetProperty() API:

```
std::string out;
db->GetProperty("rocksdb.estimate-table-readers-mem", &out);
```

In MongoRocks, just call this API from the mongo shell:

```
> db.serverStatus()["rocksdb"]["estimate-table-readers-mem"]
```

In partitioned index/filters the partitions are alwasy stored in block cache. The top-level index can be configured to be stored in heap or block cache via `cache_index_and_filter_blocks`.

Memtable

You can think of memtables as in-memory write buffers. Each new key-value pair is first written to the memtable. Memtable size is controlled by the option `write_buffer_size`. It's usually not a big memory consumer. However, memtable size is inversely proportional to write amplification -- the more memory you give to the memtable, the less the write amplification is. If you increase your memtable size, be sure to also increase your L1 size! L1 size is controlled by the option `max_bytes_for_level_base`.

To get the current memtable size, you can use:

```
std::string out;
db->GetProperty("rocksdb.cur-size-all-mem-tables", &out);
```

In MongoRocks, the equivalent call is

```
> db.serverStatus()["rocksdb"]["cur-size-all-mem-tables"]
```

Since version 5.6, you can cost the memory budget of memtables as a part of block cache. Check Write Buffer Manager for the information.

Blocks pinned by iterators

Blocks pinned by iterators usually don't contribute much to the overall memory usage. However, in some cases, when you have 100k read transactions happening simultaneously, it might put a strain on memory. Memory usage for pinned blocks is easy to calculate. Each iterator pins exactly one data block for each L0 file plus one data block for each L1+ level. So the total memory usage from pinned blocks is approximately `num_iterators * block_size * ((num_levels-1) + num_l0_files)`. To get the statistics about this memory usage, call GetPinnedUsage() on block cache object:

```
    table_options.block_cache->GetPinnedUsage();
```