

# RocksDB v6.14 Document



RocksDB is a storage engine with key/value interface, where keys and values are arbitrary byte streams. It is a C++ library. It was developed at Facebook based on LevelDB and provides backwards-compatible support for LevelDB APIs.



下载手机APP  
畅享精彩阅读

# 目 录

致谢

[RocksDB Wiki](#)

[Overview](#)

[RocksDB FAQ](#)

[Terminology](#)

[Requirements](#)

[Contributors' Guide](#)

[Release Methodology](#)

[RocksDB Users and Use Cases](#)

[RocksDB Public Communication and Information Channels](#)

[Basic Operations](#)

[Iterator](#)

[Prefix seek](#)

[SeekForPrev](#)

[Tailing Iterator](#)

[Compaction Filter](#)

[Read-Modify-Write \(Merge\) Operator](#)

[Column Families](#)

[Creating and Ingesting SST files](#)

[Single Delete](#)

[Low Priority Write](#)

[Time to Live \(TTL\) Support](#)

[Transactions](#)

[Snapshot](#)

[DeleteRange](#)

[Atomic flush](#)

[Secondary instance](#)

[Approximate Size](#)

[User Timestamp \(Experimental\)](#)

[Options](#)

[Setup Options and Basic Tuning](#)

[Option String and Option Map](#)

[RocksDB Options File](#)

[MemTable](#)

[Write Ahead Log](#)

[Write Ahead Log File Format](#)

[WAL Recovery Modes](#)

[WAL Performance](#)

## [MANIFEST](#)

[Block Cache](#)

[Write Buffer Manager](#)

## [Compaction](#)

[Leveled Compaction](#)

[Universal compaction style](#)

[FIFO compaction style](#)

[Manual Compaction](#)

[Sub-Compaction](#)

[Choose Level Compaction Files](#)

[Managing Disk Space Utilization](#)

## [SST File Formats](#)

[Block-based Table Format](#)

[PlainTable Format](#)

[CuckooTable Format](#)

[Index Block Format](#)

[Bloom Filter](#)

[Data Block Hash Index](#)

## [IO](#)

[Rate Limiter](#)

[SST File Manager](#)

[Direct I/O](#)

## [Compression](#)

[Dictionary Compression](#)

[Full File Checksum](#)

[Background Error Handling](#)

[Huge Page TLB Support](#)

[Logging and Monitoring](#)

[Logger](#)

[Statistics](#)

[Perf Context and IO Stats Context](#)

[EventListener](#)

[Known Issues](#)

[Troubleshooting Guide](#)

[Tools / Utilities](#)

[Administration and Data Access Tool](#)

- Benchmarking Tools
  - How to Backup RocksDB?
- Replication Helpers
- Checkpoints
- How to persist in-memory RocksDB database
- Stress Test
- Third-party language bindings

- RocksDB Trace, Replay, Analyzer, and Workload Generation
- Block cache analysis and simulation tools

## Implementation Details

- Delete Stale Files
- Partitioned Index/Filters
- WritePrepared-Transactions
- WriteUnprepared-Transactions
- How we keep track of live SST files
- How we index SST
- Merge Operator Implementation
- RocksDB Repairer
- Two Phase Commit
- Iterator's Implementation
- Simulation Cache
- Persistent Read Cache
- DeleteRange Implementation
- unordered\_write

## 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 (Merge) Performance
  - Delete A Range Of Keys
  - Write Stalls
  - Pipelined Write

[MultiGet Performance](#)

[Tuning Guide](#)

[Memory usage in RocksDB](#)

[Speed-Up DB Open](#)

[Implement Queue Service Using RocksDB](#)

[Projects Being Developed](#)

[Misc](#)

[Building on Windows](#)

[Open Projects](#)

[Talks](#)

[Publication](#)

[Features Not in LevelDB](#)

[How to ask a performance-related question?](#)

[Articles about Rocks](#)

# 致谢

当前文档《RocksDB v6.14 Document》由进击的皇虫使用书栈网(BookStack.CN)进行构建，生成于2020-11-26。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：[Facebook](https://github.com/facebook/rocksdb) <https://github.com/facebook/rocksdb>

文档地址：<http://www.bookstack.cn/books/rocksdb-6.14-en>

书栈官网：<https://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# Welcome to RocksDB

---

RocksDB is a storage engine with key/value interface, where keys and values are arbitrary byte streams. It is a C++ library. It was developed at Facebook based on LevelDB and provides backwards-compatible support for LevelDB APIs.

RocksDB supports various storage hardware, with fast flash as the initial focus. It uses a Log Structured Database Engine for storage, is written entirely in C++, and has a Java wrapper called RocksJava. See [RocksJava Basics](#).

RocksDB can adapt to a variety of production environments, including pure memory, Flash, hard disks or remote storage. Where RocksDB cannot automatically adapt, highly flexible configuration settings are provided to allow users to tune it for them. It supports various compression algorithms and good tools for production support and debugging.

## Features

---

- Designed for application servers wanting to store up to a few terabytes of data on local or remote storage systems.
- Optimized for storing small to medium size key-values on fast storage – flash devices or in-memory
- It works well on processors with many cores

## Features Not in LevelDB

---

RocksDB introduces dozens of new major features. See [the list of features not in LevelDB](#).

## Getting Started

---

For a complete Table of Contents, see the sidebar to the right. Most readers will want to start with the [Overview](#) and the [Basic Operations](#) section of the Developer's Guide. Get your initial options set-up following [Setup Options and Basic Tuning](#). Also check [RocksDB FAQ](#). There is also a [RocksDB Tuning Guide](#) for advanced RocksDB users.

## Releases

---

RocksDB releases are done in github. For Java users, Maven Artifacts are also updated. See [RocksDB Release Methodology](#).

## Contributing to RocksDB

---

You are welcome to send pull requests to contribute to RocksDB code base! Check [RocksDB-Contribution-Guide](#) for guideline.

## Troubleshooting and asking for help

---

Follow [RocksDB Troubleshooting Guide](#) for the guidelines.

## Blog

---

- Check out our blog at [rocksdb.org/blog](http://rocksdb.org/blog)

## Project History

---

- [The History of RocksDB](#)
- [Under the Hood: Building and open-sourcing RocksDB](#).

## Links

---

- [Examples](#)
- [Official Blog](#)
- [Stack Overflow: RocksDB](#)
- [Talks](#)

## Contact

---

- [RocksDB Google Group](#)
- [Public Developer's Discussion Group](#)

# 1. Introduction

---

RocksDB started at [Facebook](#) as a storage engine for server workloads on various storage media, with the initial focus on fast storage (especially Flash storage). It is a C++ library to store keys and values, which are arbitrarily-sized byte streams. It supports both point lookups and range scans, and provides different types of ACID guarantees.

A balance is struck between customizability and self-adaptability. RocksDB features highly flexible configuration settings that may be tuned to run on a variety of production environments, including SSDs, hard disks, ramfs, or remote storage. It supports various compression algorithms and good tools for production support and debugging. On the other hand, efforts are also made to limit the number of knobs, to provide good enough out-of-box performance, and to use some adaptive algorithms wherever applicable.

RocksDB borrows significant code from the open source [leveldb](#) project as well as ideas from [Apache HBase](#). The initial code was forked from open source leveldb 1.5. It also builds upon code and ideas that were developed at Facebook before RocksDB.

## 2. Assumptions and Goals

---

### Performance:

The primary design point for RocksDB is that it should be performant for fast storage and for server workloads. It should support efficient point lookups as well as range scans. It should be configurable to support high random-read workloads, high update workloads or a combination of both. Its architecture should support easy tuning of trade-offs for different workloads and hardware.

### Production Support:

RocksDB should be designed in such a way that it has built-in support for tools and utilities that help deployment and debugging in production environments. If the storage engine cannot yet be able to automatically adapt the application and hardware, we will provide some parameters to allow users to tune performance.

### Compatibility:

Newer versions of this software should be backward compatible, so that existing applications do not need to change when upgrading to newer releases of RocksDB. Unless using newly provided features, existing applications also should be able to revert to a recent old release. See [RocksDB Compatibility Between Different Releases](#).

## 3. High Level Architecture

RocksDB is a storage engine library of key-value store interface where keys and values are arbitrary byte streams. RocksDB organizes all data in sorted order and the common operations are `Get(key)` , `NewIterator()` , `Put(key, val)` , `Delete(key)` , and `SingleDelete(key)` .

The three basic constructs of RocksDB are *memtable*, *sstfile* and *logfile*. The *memtable* is an in-memory data structure - new writes are inserted into the *memtable* and are optionally written to the *logfile*. The *logfile* is a sequentially-written file on storage. When the *memtable* fills up, it is flushed to a *sstfile* on storage and the corresponding *logfile* can be safely deleted. The data in an *sstfile* is sorted to facilitate easy lookup of keys.

The format of a default *sstfile* is described in more details [here](#).

## 4. Features

### Column Families

RocksDB supports partitioning a database instance into multiple column families. All databases are created with a column family named “default”, which is used for operations where column family is unspecified.

RocksDB guarantees users a consistent view across column families, including after crash recovery when WAL is enabled or atomic flush is enabled. It also supports atomic cross-column family operations via the `WriteBatch` API.

### Updates

A `Put` API inserts a single key-value to the database. If the key already exists in the database, the previous value will be overwritten. A `Write` API allows multiple keys-values to be atomically inserted, updated, or deleted in the database. The database guarantees that either all of the keys-values in a single `Write` call will be inserted into the database or none of them will be inserted into the database. If any of those keys already exist in the database, previous values will be overwritten. A special `Range Delete` can be used to delete all keys from a range.

### Gets, Iterators and Snapshots

Keys and values are treated as pure byte streams. There is no limit to the size of a key or a value. The `Get` API allows an application to fetch a single key-value from the database. The `MultiGet` API allows an application to retrieve a bunch of keys from the database. All the keys-values returned via a `MultiGet` call are consistent with one-another.

All data in the database is logically arranged in sorted order. An application can

specify a key comparison method that specifies a total ordering of keys. An `Iterator` API allows an application to do a `RangeScan` on the database. The `Iterator` can seek to a specified key and then the application can start scanning one key at a time from that point. The `Iterator` API can also be used to do a reverse iteration of the keys in the database. A consistent-point-in-time view of the database is created when the `Iterator` is created. Thus, all keys returned via the `Iterator` are from a consistent view of the database.

A `Snapshot` API allows an application to create a point-in-time view of a database. The `Get` and `Iterator` APIs can be used to read data from a specified snapshot. In a sense, a `Snapshot` and an `Iterator` both provide a point-in-time view of the database, but their implementations are different. Short-lived/foreground scans are best done via an iterator while long-running/background scans are better done via a snapshot. An iterator keeps a reference count on all underlying files that correspond to that point-in-time-view of the database - these files are not deleted until the `Iterator` is released. A snapshot, on the other hand, does not prevent file deletions; instead the compaction process understands the existence of snapshots and promises never to delete a key that is visible in any existing snapshot.

Snapshots are not persisted across database restarts: a reload of the RocksDB library (via a server restart) releases all pre-existing snapshots.

## Transactions

RocksDB supports multi-operational transactions. It supports both of optimistic and pessimistic mode. See [Transactions](#).

## Prefix Iterators

Most LSM-tree engines cannot support an efficient `RangeScan` API because it needs to look into multiple data files. But most applications do not do pure-random scans of key ranges in the database; instead applications typically scan within a key-prefix. RocksDB uses this to its advantage. Applications can configure a `prefix_extractor` to specify a key-prefix. RocksDB uses this to store bloom filters for every key-prefix. An iterator that specifies a prefix (via `ReadOptions`) will use these bloom bits to avoid looking into data files that do not contain keys with the specified key-prefix.

## Persistence

RocksDB has a [Write Ahead Log \(WAL\)](#). All Puts are stored in an in-memory buffer called the memtable as well as optionally inserted into WAL. On restart, it re-processes all the transactions that were recorded in the log.

WAL can be configured to be stored in a directory different from the directory where the SST files are stored. This is necessary for those cases in which you might want to store all data files in non-persistent fast storage. At the same time, you can ensure no data loss by putting all transaction logs on slower but persistent storage.

Each `Put` has a flag, set via `WriteOptions`, which specifies whether or not the `Put` should be inserted into the transaction log. The `WriteOptions` may also specify whether or not a sync call is issued to the transaction log before a `Put` is declared to be committed.

Internally, RocksDB uses a batch-commit mechanism to batch transactions into the log so that it can potentially commit multiple transactions using a single sync call.

## Data Checksuming

RocksDB uses a checksum to detect corruptions in storage. These checksums are for each SST file block (typically between `4K` to `128K` in size). A block, once written to storage, is never modified. RocksDB dynamically detects hardware support for checksum computations and avails itself of that support when available.

## Multi-Threaded Compactions

Compactions are needed to remove multiple copies of the same key that may occur if an application overwrites an existing key. Compactions also process deletions of keys. Compactions may occur in multiple threads if configured appropriately.

The entire database is stored in a set of `sstfiles`. When a `memtable` is full, its content is written out to a file in Level-0 (`L0`) of the LSM tree. RocksDB removes duplicate and overwritten keys in the memtable when it is flushed to a file in `L0`. In `compaction`, some files are periodically read in and merged to form larger files, often going into the next LSM level (such as `L1`, up to `Lmax`).

The overall write throughput of an LSM database directly depends on the speed at which compactions can occur, especially when the data is stored in fast storage like SSD or RAM. RocksDB may be configured to issue concurrent compaction requests from multiple threads. It is observed that sustained write rates may increase by as much as a factor of 10 with multi-threaded compaction when the database is on SSDs, as compared to single-threaded compactions.

## Compaction Styles

Both Level Style Compaction and Universal Style Compaction store data in a fixed number of logical levels in the database. More recent data is stored in Level-0 (`L0`) and older data in higher-numbered levels, up to `Lmax`. Files in `L0` may have overlapping keys, but files in other levels generally form a single sorted run per level.

Level Style Compaction (default) typically optimizes disk footprint vs. logical database size (space amplification) by minimizing the files involved in each compaction step: merging one file in `Ln` with all its overlapping files in `Ln+1` and replacing them with new files in `Ln+1`.

Universal Style Compaction typically optimizes total bytes written to disk vs. logical database size (write amplification) by merging potentially many files and levels at

once, requiring more temporary space. Universal typically results in lower write-amplification but higher space- and read-amplification than Level Style Compaction.

FIFO Style Compaction drops oldest file when obsolete and can be used for cache-like data.

We also enable developers to develop and experiment with custom compaction policies. For this reason, RocksDB has appropriate hooks to switch off the inbuilt compaction algorithm and has other APIs to allow applications to operate their own compaction algorithms. `Options.disable_auto_compaction`, if set, disables the native compaction algorithm. The `GetLiveFilesMetaData` API allows an external component to look at every data file in the database and decide which data files to merge and compact. Call `CompactFiles` to compact files you want. The `DeleteFile` API allows applications to delete data files that are deemed obsolete.

## Metadata storage

A `MANIFEST` file in the database records the database state. The compaction process adds new files and deletes existing files from the database, and it makes these operations persistent by recording them in the `MANIFEST` file.

## Avoiding Stalls

Background compaction threads are also used to flush *memtable* contents to a file on storage. If all background compaction threads are busy doing long-running compactations, then a sudden burst of writes can fill up the *memtable*(s) quickly, thus stalling new writes. This situation can be avoided by configuring RocksDB to keep a small set of threads explicitly reserved for the sole purpose of flushing *memtable* to storage.

## Compaction Filter

Some applications may want to process keys at compaction time. For example, a database with inherent support for time-to-live (TTL) may remove expired keys. This can be done via an application-defined Compaction Filter. If the application wants to continuously delete data older than a specific time, it can use the compaction filter to drop records that have expired. The RocksDB Compaction Filter gives control to the application to modify the value of a key or to drop a key entirely as part of the compaction process. For example, an application can continuously run a data sanitizer as part of the compaction.

## ReadOnly Mode

A database may be opened in ReadOnly mode, in which the database guarantees that the application may not modify anything in the database. This results in much higher read performance because oft-traversed code paths avoid locks completely.

## Database Debug Logs

By default, RocksDB writes detailed logs to a file named `LOG*`. These are mostly used for debugging and analyzing a running system. Users can choose different log levels. This `LOG` may be configured to roll at a specified periodicity. The logging interface is pluggable. Users can plug in a different logger.

## Data Compression

RocksDB supports lz4, zstd, snappy, zlib, and lz4\_hc compression, as well as xpress under Windows. RocksDB may be configured to support different compression algorithms for data at the bottommost level, where `90%` of data lives. A typical installation might configure ZSTD (or Zlib if not available) for the bottom-most level and LZ4 (or Snappy if it is not available) for other levels. See [Compression](#).

## Full Backups and Replication

RocksDB has provides a backup engine, [`BackupableDB`](#). You can read more about it here: [How to backup RocksDB?](#)

RocksDB itself is not a replicated, but it provides some helper functions to enable users to implement their replication system on top of RocksDB, see [Replication Helpers](#)

## Support for Multiple Embedded Databases in the same process

A common use-case for RocksDB is that applications inherently partition their data set into logical partitions or shards. This technique benefits application load balancing and fast recovery from faults. This means that a single server process should be able to operate multiple RocksDB databases simultaneously. This is done via an environment object named `Env`. Among other things, a thread pool is associated with an `Env`. If applications want to share a common thread pool (for background compactions) among multiple database instances, then it should use the same `Env` object for opening those databases.

Similarly, multiple database instances may share the same block cache.

## Block Cache – Compressed and Uncompressed Data

RocksDB uses a LRU cache for blocks to serve reads. The block cache is partitioned into two individual caches: the first caches uncompressed blocks and the second caches compressed blocks in RAM. If a compressed block cache is configured, users may wish to enable direct I/O to prevent the OS page cache from doubly-caching the same compressed data.

## Table Cache

The Table Cache is a construct that caches open file descriptors. These file descriptors are for `sstfiles`. An application can specify the maximum size of the Table Cache, or configure RocksDB to always keep all files open, to achieve better performance.

## I/O Control

RocksDB allows users to configure I/O from and to SST files in different ways. Users can enable direct I/O so that RocksDB takes full control to the I/O and caching. An alternative is to leverage some options to allow users to hint about how I/O should be executed. They can suggest RocksDB to call fadvise in files to read, call periodic range sync in files being appended, or enable direct I/O. See [IO](#)

## Stackable DB

RocksDB has a built-in wrapper mechanism to add functionality as a layer above the code database kernel. This functionality is encapsulated by the [StackableDB](#) API. For example, the time-to-live functionality is implemented by a [StackableDB](#) and is not part of the core RocksDB API. This approach keeps the code modularized and clean.

## Memtables:

### Pluggable Memtables:

The default implementation of the memtable for RocksDB is a skip list. The skip list is a sorted set, which is a necessary construct when the workload interleaves writes with range-scans. Some applications do not interleave writes and scans, however, and some applications do not do range-scans at all. For these applications, a sorted set may not provide optimal performance. For this reason, RocksDB's memtable is pluggable. Some alternative implementations are provided. Three memtables are part of the library: a skip list memtable, a vector memtable and a prefix-hash memtable. A vector memtable is appropriate for bulk-loading data into the database. Every write inserts a new element at the end of the vector; when it is time to flush the memtable to storage the elements in the vector are sorted and written out to a file in L0. A prefix-hash memtable allows efficient processing of gets, puts and scans-within-a-key-prefix. Although the pluggability of memtable is not provided as a public API, it is possible for an application to provide its own implementation of a memtable, in a private fork.

### Memtable Pipelining

RocksDB supports configuring an arbitrary number of memtables for a database. When a memtable is full, it becomes an immutable memtable and a background thread starts flushing its contents to storage. Meanwhile, new writes continue to accumulate to a newly allocated memtable. If the newly allocated memtable is filled up to its limit, it is also converted to an immutable memtable and is inserted into the flush pipeline. The background thread continues to flush all the pipelined immutable memtables to storage. This pipelining increases write throughput of RocksDB, especially when it is operating on slow storage devices.

### Garbage Collection during Memtable Flush:

When a memtable is being flushed to storage, an inline-compaction process is executed. Garbages are removed in the same way as compactions. Duplicate updates for the same key are removed from the output stream. Similarly, if an earlier put is hidden by a

later delete, then the put is not written to the output file at all. This feature reduces the size of data on storage and write amplification greatly, for some workloads.

## Merge Operator

RocksDB natively supports three types of records, a `Put` record, a `Delete` record and a `Merge` record. When a compaction process encounters a Merge record, it invokes an application-specified method called the Merge Operator. The Merge can combine multiple Put and Merge records into a single one. This powerful feature allows applications that typically do read-modify-writes to avoid the reads altogether. It allows an application to record the intent-of-the-operation as a Merge Record, and the RocksDB compaction process lazily applies that intent to the original value. This feature is described in detail in [Merge Operator](#)

## DB ID

A globally unique ID created at the time of database creation and stored in `IDENTITY` file in the DB folder by default. Optionally it can only be stored in the `MANIFEST` file. Storing in the `MANIFEST` file is recommended.

## 5. Tools

There are a number of interesting tools that are used to support a database in production. The `sst_dump` utility dumps all the keys-values in a `sst` file, as well as other information. The `ldb` tool can put, get, scan the contents of a database. `ldb` can also dump contents of the `MANIFEST`, it can also be used to change the number of configured levels of the database. See [Administration and Data Access Tool](#) for details.

## 6. Tests

There are a bunch of unit tests that test specific features of the database. A `make check` command runs all unit tests. The unit tests trigger specific features of RocksDB and are not designed to test data correctness at scale. The `db_stress` test is used to validate data correctness at scale.

## 7. Performance

RocksDB performance is benchmarked via a utility called `db_bench`. `db_bench` is part of the RocksDB source code. Performance results of a few typical workloads using Flash storage are described [here](#). You can also find RocksDB performance results for in-memory workload [here](#).

Author: Dhruba Borthakur et al.

# Building RocksDB

---

**Q: What is the absolute minimum version of gcc that we need to build RocksDB?**

A: 4.8.

**Q: What is RocksDB's latest stable release?**

A: All the releases in <https://github.com/facebook/rocksdb/releases> are stable. For RocksJava, stable releases are available in <https://oss.sonatype.org/#nexus-search;quick~rocksdb>.

## Basic Read/Write

---

**Q: Are basic operations Put(), Write(), Get() and NewIterator() thread safe?**

A: Yes.

**Q: Can I write to RocksDB using multiple processes?**

A: No. However, it can be opened using Secondary DB. If no write goes to the database, it can be opened in read-only mode from multiple processes.

**Q: Does RocksDB support multi-process read access?**

A: Yes, you can read it using secondary database. RocksDB can also support multi-process read only process without writing the database. This can be done by opening the database with DB::OpenForReadOnly() call.

**Q: Is it safe to close RocksDB while another thread is issuing read, write or manual compaction requests?**

A: No. The users of RocksDB need to make sure all functions have finished before they close RocksDB. You can speed up the waiting by calling CancelAllBackgroundWork().

**Q: What's the maximum key and value sizes supported?**

A: In general, RocksDB is not designed for large keys. The maximum recommended sizes for key and value are 8MB and 3GB respectively.

**Q: What's the fastest way to load data into RocksDB?**

A: A fast way to direct insert data to the DB:

1. using single writer thread and insert in sorted order
2. batch hundreds of keys into one write batch
3. use vector memtable
4. make sure options.max\_background\_flushes is at least 4
5. before inserting the data, disable automatic compaction, set

options.level0\_file\_num\_compaction\_trigger,  
 options.level0\_slowdown\_writes\_trigger and options.level0\_stop\_writes\_trigger to very large. After inserting all the data, issue a manual compaction.

3-5 will be automatically done if you call Options::PrepareForBulkLoad() to your option

If you can pre-process the data offline before inserting. There is a faster way: you can sort the data, generate SST files with non-overlapping ranges in parallel and bulkload the SST files. See <https://github.com/facebook/rocksdb/wiki/Creating-and-Ingesting-SST-files>

**Q: What is the correct way to delete the DB? Can I simply call DestroyDB() on a live DB?**

A: Close the DB then destroy the DB is the correct way. Calling DestroyDB() on a live DB is an undefined behavior.

**Q: What is the difference between DestroyDB() and directly deleting the DB directory manually?**

A: The major difference is that DestroyDB() will take care of the case where the RocksDB database is stored in multiple directories. For instance, a single DB can be configured to store its data in multiple directories by specifying different paths to DBOptions::db\_paths, DBOptions::db\_log\_dir, and DBOptions::wal\_dir.

**Q: Any better way to dump key-value pairs generated by map-reduce job into RocksDB?**

A: A better way is to use SstFileWriter, which allows you to directly create RocksDB SST files and add them to a rocksdb database. However, if you're adding SST files to an existing RocksDB database, then its key-range must not overlap with the database. <https://github.com/facebook/rocksdb/wiki/Creating-and-Ingesting-SST-files>

**Q: Is it safe to read from or write to RocksDB inside compaction filter callback?**

A: It is safe to read but not always safe to write to RocksDB inside compaction filter callback as write might trigger deadlock when write-stop condition is triggered.

**Q: Does RocksDB hold SST files and memtables for a snapshot?**

A: No. See <https://github.com/facebook/rocksdb/wiki/RocksDB-Basics#gets-iterators-and-snapshots> for how snapshots work.

**Q: With DBWithTTL, is there a time bound for the expired keys to be removed?**

A: DBwithTTL itself does not provide an upper time bound. Expired keys will be removed when they are part of any compaction. However, there's no guarantee that when such compaction will start. For instance, if you have a certain key-range that is never updated, then compaction is less likely to apply to that key-range. For leveled compaction, you can enforce some limit using the feature of periodic compaction to do

that. The feature right now has a limitation: if the write rate is too slow that memtable flush is never triggered, the periodic compaction won't be triggered either.

**Q: If I delete a column family, and I didn't yet delete the column family handle, can I still use it to access the data?**

A: Yes. DropColumnFamily() only marks the specified column family as dropped, and it will not be dropped until its reference count goes to zero and marked as dropped.

**Q: Why does RocksDB issue reads from the disk when I only make write request?**

A: Such IO reads are from compactions. RocksDB compaction reads from one or more SST files, perform merge-sort like operation, generate new SST files, and delete the old SST files it inputs.

**Q: Is block\_size before compression , or after?**

A: block\_size is for size before compression.

**Q: After using options.prefix\_extractor, I sometimes see wrong results. What's wrong?**

A: There are limitations of options.prefix\_extractor. If prefix iterating is used, doesn't support Prev() or SeekToLast(), and many operations don't support SeekToFirst() either. A common mistake to seek the last key of a prefix by calling Seek(), followed by Prev(). This is, however, not supported. Currently there is no way to find the last key of prefix with prefix iterating. Also, you can't continue iterating keys after finishing the prefix you seek to. In the places where those operations are needed, you can try to set ReadOptions.total\_order\_seek = true to disable prefix iterating.

**Q: If issue a Put() or Write() with WriteOptions.sync=true, does it mean all previous writes are persistent too?**

A: Yes, but only for all previous writes with WriteOptions.disableWAL=false.

**Q: I disabled write-ahead-log and rely on DB::Flush() to persist the data. It works well for single family. Can I do the same if I have multiple column families?**

A: Yes. set option.atomic\_flush=true support atomic flush multi column families.

**Q: What's the best way to delete a range of keys?**

A: See <https://github.com/facebook/rocksdb/wiki/Delete-A-Range-Of-Keys> .

**Q: What are column families used for?**

A: The most common reasons of using column families: (1) use different compaction setting, comparators, compression types, merge operators, or compaction filters in different parts of data; (2) drop a column family to delete its data; (3) one column family to store metadata and another one to store the data.

**Q: What's the difference between storing data in multiple column family and in multiple rocksdb database?**

A: The main differences will be backup, atomic writes and performance of writes. The advantage of using multiple databases: database is the unit of backup or checkpoint. It's easier to copy a database to another host than a column family. Advantages of using multiple column families: (1) write batches are atomic across multiple column families on one database. You can't achieve this using multiple RocksDB databases. (2) If you issue sync writes to WAL, too many databases may hurt the performance.

**Q: Is RocksDB really “lockless” in reads?**

A: Reads might hold mutex in the following situations: (1) access the sharded block cache (2) access table cache if options.max\_open\_files != -1 (3) if a read happens just after flush or compaction finishes, it may briefly hold the global mutex to fetch the latest metadata of the LSM tree. (4) the memory allocators RocksDB relies on (e.g. jemalloc), may sometimes hold locks These locks are only held rarely, or in fine granularity.

**Q: If I update multiple keys, should I issue multiple Put(), or put them in one write batch and issue Write()?**

A: Using write batch to batch more keys usually performs better than single Put().

**Q: What's the best practice to iterate all the keys?**

A: If it's a small or read-only database, just create an iterator and iterate all the keys. Otherwise consider to recreate iterators once a while, because an iterator will hold all the resources from being released. If you need to read from consistent view, create a snapshot and iterate using it.

**Q: I have different key spaces. Should I separate them by prefixes, or use different column families?**

A: If each key space is reasonably large, it's a good idea to put them in different column families. If it can be small, then you should consider to pack multiple key spaces into one column family, to avoid the trouble of maintaining too many column families.

**Q: Is the performance of iterator Next() the same as Prev()?**

A: The performance of reversed iterating is usually much worse than forward iterating. There are various reasons for that: 1. delta encoding in data blocks is more friendly to Next(); 2. the skip list used in the memtable is single-direction, so Prev() is another binary search; 3. the internal key order is optimized for Next().

**Q: If I want to retrieve 10 keys from rocksdb, is it better to batch them and use MultiGet() versus issuing 10 individual Get() calls?**

A: The performance is similar. MultiGet() reads from the same consistent view, but it

is not faster.

**Q: If I have multiple column families and call the DB functions without a column family handle, what the result will be?**

A: It will operate only the default column family.

**Q: Can I reuse ReadOptions, WriteOptions, etc, across multiple threads?**

A: As long as they are const, you are free to reuse them.

## Feature Support

---

**Q: Can I cancel a specific compaction?**

A: No, you can't cancel one specific compaction.

**Q: Can I close the DB when a manual compaction is in progress?**

A: No, it's not safe to do that. However, you call CancelAllBackgroundWork(db, true) in another thread to abort the running compactations, so that you can close the DB sooner. Since 6.5, you can also speed it up using DB::DisableManualCompaction().

**Q: Is it safe to directly copy an open RocksDB instance?**

A: No, unless the RocksDB instance is opened in read-only mode.

**Q: Does RocksDB support replication?**

A: No, RocksDB does not directly support replication. However, it offers some APIs that can be used as building blocks to support replication. For instance, GetUpdatesSince() allows developers to iterate through all updates since a specific point in time.

<https://github.com/facebook/rocksdb/blob/4.4.fb/include/rocksdb/db.h#L676-L687>

**Q: Does RocksDB support group commit?**

A: Yes. Multiple write requests issued by multiple threads may be grouped together. One of the threads writes WAL log for those write requests in one single write request and fsync once if configured.

**Q: Is it possible to scan/iterate over keys only? If so, is that more efficient than loading keys and values?**

A: No it is usually not more efficient. RocksDB's values are normally stored inline with keys. When a user iterate over the keys, the values are already loaded in memory, so skipping the value won't save much. In BlobDB, keys and large values are stored separately so it maybe beneficial to only iterate keys, but it is not supported yet. We may add the support in the future.

**Q: Is the transaction object thread-safe?**

A: No it's not. You can't issue multiple operations to the same transaction concurrently. (Of course, you can execute multiple transactions in parallel, which is the point of the feature.)

**Q: After iterator moves away from a key/value, is the memory pointed by those key/value still kept?**

A: No, they can be freed, unless you set `ReadOptions.pin_data = true` and your setting supports this feature.

**Q: Can I programmatically read data from an SST file?**

A: We don't support it right now. But you can dump the data using `sst_dump`. Since version 6.5, you'll be able to do it using `SstFileReader`.

**Q: RocksDB repair: when can I use it? Best-practices?**

A: Check <https://github.com/facebook/rocksdb/wiki/RocksDB-Repairer>

## Configuration and Tuning

---

**Q: What's the default value of the block cache?**

A: 8MB. That's too low for most use cases, so it's likely that you need to set your own value.

**Q: Are bloom filter blocks of SST files always loaded to memory, or can they be loaded from disk?**

A: The behavior is configurable. When `BlockBaseTableOptions::cache_index_and_filter_blocks` is set to true, then bloom filters and index block will be loaded into a LRU cache only when related `Get()` requests are issued. In the other case where `cache_index_and_filter_blocks` is set to false, then RocksDB will try to keep the index block and bloom filter in memory up to `DBOptions::max_open_files` number of SST files.

**Q: Is it safe to configure different prefix extractor for different column family?**

A: Yes.

**Q: Can I change the prefix extractor?**

A: No. Once you've specified a prefix extractor, you cannot change it. However, you can disable it by specifying a null value.

**Q: How to configure rocksdb to use multiple disks?**

A: You can create a single filesystem (ext3, xfs, etc) on multiple disks. Then you can

run rocksdb on that single file system. Some tips when using disks:

- if using RAID then don't use a too small RAID stripe size (64kb is too small, 1MB would be excellent).
- consider enabling compaction readahead by specifying `ColumnFamilyOptions::compaction_readahead_size` to at least 2MB.
- if workload is write-heavy then have enough compaction threads to keep the disks busy
- consider enabling async write behind for compaction

**Q: Can I open RocksDB with a different compression type and still read old data?**

A: Yes, since rocksdb stored the compression information in each SST file and performs decompression accordingly, you can change the compression and the db will still be able to read existing files. In addition, you can also specify a different compression for the last level by specifying `ColumnFamilyOptions::bottommost_compression`.

**Q: Can I put log files and sst files in different directories? How about information logs?**

A: Yes. WAL files can be placed in a separate directory by specifying `DBOptions::wal_dir`, information logs can as well be written in a separate directory by using `DBOptions::db_log_dir`.

**Q: If I use non-default comparators or merge operators, can I still use ldb tool?**

A: You cannot use the regular ldb tool in this case. However, you can build your custom ldb tool by passing your own options using this function `rocksdb::LDBTool::Run(argc, argv, options)` and compile it.

**Q: What will happen if I open RocksDB with a different compaction style?**

A: When opening a rocksdb database with a different compaction style or compaction settings, one of the following scenarios will happen:

1. The database will refuse to open if the new configuration is incompatible with the current LSM layout.
2. If the new configuration is compatible with the current LSM layout, then rocksdb will continue and open the database. However, in order to make the new options take full effect, it might require a full compaction.

Consider to use the migration helper function `OptionChangeMigration()`, which will compact the files to satisfy the new compaction style if needed.

**Q: Does RocksDB have columns? If it doesn't have column, why there are column families?**

A: No, RocksDB doesn't have columns. See

<https://github.com/facebook/rocksdb/wiki/Column-Families> for what is column family.

**Q: How to estimate space can be reclaimed If I issue a full manual compaction?**

A: There is no easy way to predict it accurately, especially when there is a compaction filter. If the database size is steady, DB property "rocksdb.estimate-live-data-size" is the best estimation.

**Q: What's the difference between a snapshot, a checkpoint and a backup?**

A: Snapshot is a logical concept. Users can query data using program interface, but underlying compactations still rewrite existing files.

A checkpoint will create a physical mirror of all the database files using the same Env. This operation is very cheap if the file system hard-link can be used to create mirrored files.

A backup can move the physical database files to another Env (like HDFS). The backup engine also supports incremental copy between different backups.

**Q: Which compression type should I use?**

A: Start with LZ4 (Snappy if LZ4 is not available) for all levels for good performance. If you want to further reduce data size, try to use Zlib in the last level.

**Q: Is compaction needed if no key is deleted or overwritten?**

A: Even if there is no need to clear out-of-date data, compaction is needed to ensure read performance.

**Q: After a write following option.disableWAL=true, I write another record with options.sync=true, will it persist the previous write too?**

A: No. After the program crashes, writes with option.disableWAL=true will be lost, if they are not flushed to SST files.

**Q: What is options.target\_file\_size\_multiplier useful for?**

A: It's a rarely used feature. For example, you can use it to reduce the number of the SST files.

**Q: I observed burst write I/Os. How can I eliminate that?**

A: Try to use the rate limiter:

<https://github.com/facebook/rocksdb/blob/v4.9/include/rocksdb/options.h#L875-L879>

**Q: Can I change the compaction filter without reopening the DB?**

A: It's not supported. However, you can achieve it by implementing your CompactionFilterFactory which returns different compaction filters.

**Q: How many column families can a single db support?**

A: Users should be able to run at least thousands of column families without seeing any error. However, too many column families don't usually perform well. We don't recommend users to use more than a few hundreds of column families.

**Q: Can I reuse DBOptions or ColumnFamilyOptions to open multiple DBs or column families?**

A: Yes. Internally, RocksDB always makes a copy to those options, so you can freely change them and reuse these objects.

## Portability

---

**Q: Can I run RocksDB and store the data on HDFS?**

A: Yes, by using the Env returned by NewHdfsEnv(), RocksDB will store data on HDFS. However, the file lock is currently not supported in HDFS Env.

**Q: Does RocksJava support all the features?**

A: We are working toward making RocksJava feature compatible. However, you're more than welcome to submit pull request if you find something is missing

## Backup

---

**Q: Can I preserve a "snapshot" of RocksDB and later roll back the DB state to it?**

A: Yes, via the [BackupEngine](#) or [Checkpoints](#).

**Q: Does BackupableDB create a point-in-time snapshot of the database?**

A: Yes when BackupOptions::backup\_log\_files = true or flush\_before\_backup is set to true when calling CreateNewBackup().

**Q: Does the backup process affect accesses to the database in the mean while?**

A: No, you can keep reading and writing to the database at the same time.

**Q: How can I configure RocksDB to backup to HDFS?**

A: Use BackupableDB and set backup\_env to the return value of NewHdfsEnv().

## Failure Handling

---

**Q: Does RocksDB throw exceptions?**

A: No, RocksDB returns rocksdb::Status to indicate any error. However, RocksDB does not catch exceptions thrown by STL or other dependencies. For instance, so it's possible that you will see std::bad\_alloc when memory allocation fails, or similar

exceptions in other situations.

**Q: How RocksDB handles read or write I/O errors?**

A: If the I/O errors happen in the foreground operations such as Get() and Write(), then RocksDB will return rocksdb::IOError status. If the error happens in background threads and options.paranoid\_checks=true, we will switch to the read-only mode. All the writes will be rejected with the status code representing the background error.

**Q: How to distinguish type of exceptions thrown by RocksJava?**

A: Yes, RocksJava throws RocksDBException for every RocksDB related exceptions.

## Failure Recovery

---

**Q: If my process crashes, can it corrupt the database?**

A: No, but data in the un-flushed mem-tables might be lost if [Write Ahead Log \(WAL\)](#) is disabled.

**Q: If my machine crashes and rebooted, will RocksDB preserve the data?**

A: Data is synced when you issue a sync write (write with WriteOptions.sync=true), call DB::SyncWAL(), or when memtables are flushed.

**Q: How to know the number of keys stored in a RocksDB database?**

A: Use GetIntProperty(cf\_handle, "rocksdb.estimate-num-keys") to obtain an estimated number of keys stored in a column family, or use GetAggregatedIntProperty("rocksdb.estimate-num-keys", &num\_keys) to obtain an estimated number of keys stored in the whole RocksDB database.

**Q: Why GetIntProperty can only return an estimated number of keys in a RocksDB database?**

A: Obtaining an accurate number of keys in any LSM databases like RocksDB is a challenging problem as they have duplicate keys and deletion entries (i.e., tombstones) that will require a full compaction in order to get an accurate number of keys. In addition, if the RocksDB database contains merge operators, it will also make the estimated number of keys less accurate.

## Resource Management

---

**Q: How much resource does an iterator hold and when will these resource be released?**

A: Iterators hold both data blocks and memtables in memory. The resource each iterator holds are:

1. The data blocks that the iterator is currently pointing to. See

<https://github.com/facebook/rocksdb/wiki/Memory-usage-in-RocksDB#blocks-pinned-by-iterators>

2. The memtables that existed when the iterator was created, even after the memtables have been flushed.
3. All the SST files on disk that existed when the iterator was created, even if they are compacted.

These resources will be released when the iterator is deleted.

**Q: How to estimate total size of index and filter blocks in a DB?**

A: For an offline DB, “sst\_dump –show\_properties –command=none” will show you the index and filter size for a specific sst file. You can sum them up for all DB. For a running DB, you can fetch from DB property “kAggregatedTableProperties”. Or calling DB::GetPropertiesOfAllTables() and sum up the index and filter block size of individual files.

**Q: Can rocksdb tell us the total number of keys in the database? Or the total number of keys within a range?**

A: RocksDB can estimate number of keys through DB property “rocksdb.estimate-num-keys”. Note this estimation can be far off when there are merge operators, existing keys overwritten, or deleting non-existing keys.

The best way to estimate total number of keys within a range is to first estimate size of a range by calling DB::GetApproximateSizes(), and then estimate number of keys from that.

## Others

---

**Q: Who is using RocksDB?**

A: <https://github.com/facebook/rocksdb/blob/master/USERS.md>

**Q: How should I implement multiple data shards/partitions.**

A: You can start with using one RocksDB database per shard/partition.

**Q: DB operations fail because of out-of-space. How can I unblock myself?**

A: First clear up some free space. The DB will automatically start accepting operations once enough free space is available. The only exception is if 2PC is enabled and the WAL sync fails (in this case, the DB needs to be reopened). See [Background Error Handling](#) for more details.

**Iterator:** iterators are used by users to query keys in a range in sorted order. See <https://github.com/facebook/rocksdb/wiki/Basic-Operations#iteration>

**Point lookup:** In RocksDB, point lookup means reading one key using Get() or MultiGet().

**Range lookup:** Range lookup means reading a range of keys using an Iterator.

**SST File (Data file / SST table):** SST stands for Sorted Sequence Table. They are persistent files storing data. In the file keys are usually organized in sorted order so that a key or iterating position can be identified through a binary search.

**Index** The index on the data blocks in a SST file. It is persisted as an index block in the SST file. The default index format is the binary search index.

**Partitioned Index** The binary search index block partitioned to multiple smaller blocks. See <https://github.com/facebook/rocksdb/wiki/Partitioned-Index-Filters>

**LSM-tree:** See the definition in [https://en.wikipedia.org/wiki/Log-structured\\_merge-tree](https://en.wikipedia.org/wiki/Log-structured_merge-tree) RocksDB is LSM-tree-based storage engine.

**Write-Ahead-Log (WAL) or log:** A log file used to recover data that is not yet flushed to SST files, during DB recovery. See <https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-File-Format>

**memtable / write buffer:** the in-memory data structure that stores the most recent updates of the database. Usually it is organized in sorted order and includes a binary searchable index. See <https://github.com/facebook/rocksdb/wiki/Basic-Operations#memtable-and-table-factories>

**memtable switch:** During this process, the current **active memtable** (the one current writes go to) is closed and turned into an **immutable memtable**. At the same time, we will close the current WAL file and start a new one.

**immutable memtable:** A closed **memtable** that is waiting to be flushed.

**sequence number (SeqNum / Seqno):** each write to the database will be assigned an auto-incremented ID number. The number is attached with the key-value pair in WAL file, memtable, and SST files. The sequence number is used to implement snapshot read, garbage collection in compactions, MVCC in transactions, and some other purposes.

**recovery:** the process of restarting a database after it failed or was closed.

**flush:** background jobs that write out data in mem tables into SST files.

**compaction:** background jobs that merge some SST files into some other SST files. LevelDB's compaction also includes flush. In RocksDB, we further distinguished the two. See <https://github.com/facebook/rocksdb/wiki/RocksDB-Basics#multi-threaded-compactions> and <https://github.com/facebook/rocksdb/wiki/Compaction>

**(LSM) level:** a logical organization of the DB physical data for maintaining desired

LSM-tree shape and structure. See <https://github.com/facebook/rocksdb/wiki/Compaction>, particularly <https://github.com/facebook/rocksdb/wiki/Compaction#lsm-terminology-and-metaphors>

**Leveled Compaction or Level-Based Compaction Style:** the default compaction style of RocksDB. [Leveled Compaction](#)

**Universal Compaction Style:** an alternative compaction algorithm. See <https://github.com/facebook/rocksdb/wiki/Universal-Compaction>

**Comparator:** A plug-in class which can define the order of keys. See <https://github.com/facebook/rocksdb/blob/master/include/rocksdb/comparator.h>

**Column Family:** column family is a separate key space in one DB. In spite of the misleading name, it has nothing to do with the “column family” concept in other storage systems. RocksDB doesn’t even have the concept of “column”. See <https://github.com/facebook/rocksdb/wiki/Column-Families>

**snapshot:** a snapshot is a logical consistent point-in-time view, in a running DB. See <https://github.com/facebook/rocksdb/wiki/RocksDB-Basics#gets-iterators-and-snapshots>

**checkpoint:** A checkpoint is a physical mirror of the database in another directory in the file system. See <https://github.com/facebook/rocksdb/wiki/Checkpoints>

**backup:** RocksDB has a backup tool to help users backup the DB state to a different location, like HDFS. See <https://github.com/facebook/rocksdb/wiki/How-to-backup-RocksDB%3F>

**Version:** An internal concept of RocksDB. A version consists of all the live SST files in one point of the time. Once a flush or compaction finishes, a new “version” will be created because the list of live SST files has changed. An old “version” can continue being used by on-going read requests or compaction jobs. Old versions will eventually be garbage collected.

**Super Version:** An internal concept of RocksDB. A super version consists the list of SST files (a “version”) and the list of live mem tables in one point of the time. Either a compaction or flush, or a mem table switch will cause a new “super version” to be created. An old “super version” can continue being used by on-going read requests. Old super versions will eventually be garbage collected after it is not needed anymore.

**block cache:** in-memory data structure that cache the hot data blocks from the SST files. See <https://github.com/facebook/rocksdb/wiki/Block-Cache>

**statistics:** an in-memory data structure that contains cumulative stats of live databases. <https://github.com/facebook/rocksdb/wiki/Statistics>

**perf context:** an in-memory data structure to measure thread-local stats. It is usually used to measure per-query stats. See <https://github.com/facebook/rocksdb/wiki/Perf-Context-and-IO-Stats-Context>

**DB properties:** some running status that can be returned by the function DB::GetProperty().

**Table Properties:** metadata stored in each SST file. It includes system properties that are generated by RocksDB and user defined table properties calculated by user defined call-backs. See

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/table\\_properties.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/table_properties.h)

**write stall:** When flush or compaction is backlogged, RocksDB may actively slowdown writes to make sure flush and compaction can catch up. See

<https://github.com/facebook/rocksdb/wiki/Write-Stalls>

**bloom filter:** See <https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter>

**prefix bloom filter:** a special bloom filter that can be limitly used in iterators. Some file reads are avoided if an SST file or memtable doesn't contain the prefix of the lookup key extracted by the prefix extractor. See

<https://github.com/facebook/rocksdb/wiki/Prefix-Seek-API-Changes>

**prefix extractor:** a callback class that can extract prefix part of a key. This is most frequently used as the prefix used in prefix bloom filter. See

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/slice\\_transform.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/slice_transform.h)

**block-based bloom filter or full bloom filter:** Two different approaches of storing bloom filters in SST files. Both are features of the block-based table format. See

<https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter#new-bloom-filter-format>

**Partitioned Filters:** Partitioning a full bloom filter into multiple smaller blocks. See <https://github.com/facebook/rocksdb/wiki/Partitioned-Index-Filters>.

**compaction filter:** a user plug-in that can modify or drop existing keys during a compaction. See

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/compaction\\_filter.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/compaction_filter.h)

**merge operator:** RocksDB supports a special operator Merge(), which is a delta record, merge operand, to the existing value. Merge operator is a user defined call-back class which can merge the merge operands. See

<https://github.com/facebook/rocksdb/wiki/Merge-Operator-Implementation>

**Block-Based Table:** The default SST file format. See

<https://github.com/facebook/rocksdb/wiki/Rocksdb-BlockBasedTable-Format>

**Block:** data block of SST files. In block-based table SST files, a block is always checksummed and usually compressed for storage.

**PlainTable:** An alternative format of SST file format, optimized for ramfs. See

<https://github.com/facebook/rocksdb/wiki/PlainTable-Format>

**forward iterator / tailing iterator:** A special iterator option that optimizes for very specific use cases. See <https://github.com/facebook/rocksdb/wiki/Tailing-Iterator>

**single delete:** a special delete operation which only works when users never update an existing key: <https://github.com/facebook/rocksdb/wiki/Single-Delete>

**rate limiter:** it is used to limit the rate of bytes written to the file system by flush and compaction. See <https://github.com/facebook/rocksdb/wiki/Rate-Limiter>

**Pessimistic Transactions** Using locks to provide isolation between multiple concurrent transactions. The default write policy is WriteCommitted.

**2PC (Two-phase commit)** The pessimistic transactions could commit in two phases: first Prepare and then the actual Commit. See <https://github.com/facebook/rocksdb/wiki/Two-Phase-Commit-Implementation>

**WriteCommitted** The default write policy in pessimistic transactions, which buffers the writes in memory and write them into the DB upon commit of the transaction.

**WritePrepared** A write policy in pessimistic transactions that buffers the writes in memory and write them into the DB upon prepare if it is a 2PC transaction or commit otherwise. See <https://github.com/facebook/rocksdb/wiki/WritePrepared-Transactions>

**WriteUnprepared** A write policy in pessimistic transactions that avoid the need for larger memory buffers by writing data to the DB as they are sent by the transaction. See <https://github.com/facebook/rocksdb/wiki/WritePrepared-Transactions>

We detail the minimum requirements for compiling RocksDB and optionally RocksJava, and running RocksJava binaries supplied via Maven Central.

## Compiling

---

### All platforms

---

- **Java:** [OpenJDK 1.7+](#) (required only for RocksJava)
- Tools:
  - curl (recommended; required only for RocksJava)
- Libraries:
  - [gflags](#) 2.0+ (required for testing and benchmark code)
  - [zlib](#) 1.2.8+ (optional)
  - [bzip2](#) 1.0.6+ (optional)
  - [lz4](#) r131+ (optional)
  - [snappy](#) 1.1.3+ (optional)
  - [zstandard](#) 0.5.1+ (optional)

### Linux

---

- **Architecture:** x86 / x86\_64 / arm64 / ppc64le / s390x
- **C/C++ Compiler:** GCC 4.8+ or Clang
- Tools:
  - GNU Make or [CMake](#) 3.14.5+

### macOS

---

- **Architecture:** x86\_64
- **OS:** macOS 10.12+
- **C/C++ Compiler:** Apple XCode Clang
- Tools:
  - GNU Make or [CMake](#) 3.14.5+

### Windows

---

- **Architecture:** x86\_64
- **OS:** Windows 7+
- **C/C++ Compiler:** Microsoft Visual Studio 2015+
- **Java:** [OpenJDK 1.7+](#) (required only for RocksJava)
- Tools:
  - [CMake](#) 3.14.5+

# RocksJava Binaries

---

The minimum requirements for running the official RocksJava binaries from [Maven Central](#).

For all platforms the native component of the binaries is statically linked, and so requires very little. The Java component requires [OpenJDK 1.7+](#).

The binaries are built using Docker containers on real hardware, you can find our Docker build containers here: <https://github.com/evolvedbinary/docker-rocksjava>

## Linux

---

For Linux we provide binaries built for either GNU Lib C, or Musl based platforms (since RocksJava 6.5.2).

Architecture	glibc version (minimum)	muslc version (minimum)
x86	2.12	1.1.16
x86_64	2.12	1.1.16
aarch64	2.17	1.1.16
ppc64le	2.17	1.1.16
s390x (z10)	2.17	1.1.16

## macOS

---

- **Architecture:** x86\_64
- **OS:** macOS 10.12+

## Windows

---

The Windows binaries are built on Windows Server 2012 with Visual Studio 2015.

- **Architecture:** x86\_64
- **OS:** Windows 7+
- **libc:** Microsoft Visual C++ 2015 Redistributable (x64) - 14.0.24215

# Before Contribution

Before contributing to RocksDB, please make sure that you are able to sign CLA. Your change will not be merged unless you have proper CLA signed. See <https://code.facebook.com/cla> for more information.

## Basic Development Workflow

As most open-source projects in github, RocksDB contributors work on their fork, and send pull requests to RocksDB's facebook repo. After a reviewer approves the pull request, a RocksDB team member at Facebook will merge it.

## How to Run Unit Tests

### Build Systems

RocksDB uses gtest. The makefile used for *GNU make* has some supports to help developers run all unit tests in parallel, which will be introduced below. If you use cmake, you might need find your way to run all the unit tests (you are welcome to contribute build system to make it easier).

### Run Unit Tests In Parallel

In order to run unit tests in parallel, first install *GNU parallel* on your host, and run

```
1. make all check [-j]
```

You can specify number of parallel tests to run using environment variable `J=1` , for example:

```
1. make J=64 all check [-j]
```

If you switch between release and debug build, normal or lite build, or compiler or compiler options, call `make clean` first. So here is a safe routine to run all tests:

```
1. make clean  
2. make J=64 all check [-j]
```

## Debug Single Unit Test Failures

RocksDB uses `gtest`. You can run specific unit test by running the test binary contains it. If you use GNU make, the test binary will be just under your checkpoint. For example, test `DBBasicTest.OpenWhenOpen` is in binary `db_basic_test`, so just run

```
1. ./db_basic_test
```

will run all tests in the binary.

`gtest` provides some useful command line parameters, and you can see them by calling `--help`:

```
1. ./db_basic_test --help
```

Here are some frequently used ones:

Run subset of tests using `--gtest_filter`. If you only want to run `DBBasicTest.OpenWhenOpen`, call

```
1. ./db_basic_test --gtest_filter="*DBBasicTest.OpenWhenOpen*"
```

By default, the test DB created by tests is cleared up even if test fails. You can try to preserve it by using `--gtest_throw_on_failure`. If you want to stop the debugger when assert fails, specify `--gtest_break_on_failure`. `KEEP_DB=1` environment variable is another way to preserve the test DB from being deleted at the end of a unit-test run, irrespective of whether the test fails or not:

```
1. KEEP_DB=1 ./db_basic_test --gtest_filter=DBBasicTest.Open
```

By default, the temporary test files will be under `/tmp/rocksdbtest-<number>/` (except when running in parallel they are under `/dev/shm`). You can override the location by using environment variable `TEST_TMPDIR`. For example:

```
1. TEST_TMPDIR=/dev/shm/my_dir ./db_basic_test
```

## Java Unit Tests

Sometimes we need to run Java tests too. Run

```
1. make jclean rocksdbjava jtest
```

You can put `-j` but sometimes it causes problem. Try to remove `-j` if you see problems.

## Some other build flavors

For more complicated code changes, we ask contributors to run more build flavors before sending the code review. The makefile for *GNU make* has better pre-defined support for it, although it can be manually done in *CMake* too.

To build with *AddressSanitizer (ASAN)*, set environment variable `COMPILE_WITH_ASAN` :

```
1. COMPILE_WITH_ASAN=1 make all check -j
```

To build with *ThreadSanitizer (TSAN)*, set environment variable `COMPILE_WITH_TSAN` :

```
1. COMPILE_WITH_TSAN=1 make all check -j
```

To run all `valgrind` tests :

```
1. make valgrind_test -j
```

To run *\_UndefinedBehaviorSanitizer (UBSAN)*, set environment variable `COMPILE_WITH_UBSAN` :

```
1. COMPILE_WITH_UBSAN=1 make all check -j
```

To run `l1vm`'s analyzer, run

```
1. make analyze
```

## Code Style

RocksDB follows *Google C++ Style*: <https://google.github.io/styleguide/cppguide.html>. We limit each line to 80 characters.

Some formatting can be done by a formatter by running

```
1. build_tools/format-diff.sh
```

or simply `make format` if you use *GNU make*. If you lack of dependencies to run it, the script will print out instructions for you to install them.

## Requirements Before Sending a Pull Request

### HISTORY.md

Consider updating `HISTORY.md` to mention your change, especially if it's a bug fix, public API change or an awesome new feature.

# Pull Request Summary

We recommend a "Test Plan:" section is included in the pull request summary, which introduces what testing is done to validate the quality and performance of the change.

## Add Unit Tests

Almost all code changes need to go with changes in unit tests for validation. For new features, new unit tests or tests scenarios need to be added even if it has been validated manually. This is to make sure future contributors can rerun the tests to validate their changes don't cause problem with the feature.

## Simple Changes

Pull requests for simple changes can be sent after running all unit tests with any build flavor and see all tests pass. If any public interface is changed, or Java code involved, Java tests also need to be run.

## Complex Changes

If the change is complicated enough, ASAN, TSAN and valgrind need to be run on your local environment before sending the pull request. If you run ASAN with higher version of llvm with covers almost all the functionality of valgrind, valgrind tests can be skipped. It may be hard for developers who use Windows. Just try to use the best equivalence tools available in your environment.

## Changes with Higher Risk or Some Unknowns

For changes with higher risks, other than running all tests with multiple flavors, a crash test cycle (see [Stress Test](#)) needs to be executed and see no failure. If crash test doesn't cover the new feature, consider to add it there. To run all crash test, run

```
1. make crash_test -j  
2. make crash_test_with_atomic_flush -j
```

If you can't use *GNU make*, you can manually build db\_stress binary, and run script:

```
1. python -u tools/db_crashtest.py whitebox  
2. python -u tools/db_crashtest.py blackbox  
3. python -u tools/db_crashtest.py --simple whitebox  
4. python -u tools/db_crashtest.py --simple blackbox  
5. python -u tools/db_crashtest.py --cf_consistency blackbox  
6. python -u tools/db_crashtest.py --cf_consistency whitebox
```

# Performance Improvement Changes

---

For changes that might impact performance, we suggest normal benchmarks are run to make sure there is no regression. Depending the actual performance, you may choose to run against a database backed by disks, or memory-backed file systems. Explain in the pull request summary why the performance environment is chosen, if it is not obvious. If the change is to improve performance, bring at least one benchmark test case that favors the improvement and show the improvements.

# Release Number

Release version has three parts: MAJOR.MINOR.PATCH. An example would be 6.8.2.

See [RocksDB version macros](#) for how to keep track with version you're using.

There are not objective criteria to distinguish a major version and a minor version. All features and improvements can go to minor versions, and they are applicable to new options, option default changes or public API change if needed.

A major version is usually made when several major features or improvements become stable.

Only bug fixes go to patch releases. No option of public API change is will be made in patch releases within a minor version, unless they cannot be unavoidable to fix a critical bug.

# Compatibility Between Releases

See [RocksDB Compatibility Between Different Releases](#).

# Release Frequency

RocksDB has time-based periodic releases. We typically try to make a minor release once every month.

# Release Branching and Tagging

- The 'master' branch is used for code development. It should be somewhat stable and in usable state. It is not recommended for production use, though.
- When we release, we cut-off the branch from master. For example, release 3.0 will have a branch 3.0.fb. Once the branch is stable enough (bug fixes go to both master and the branch), we create a release tag with a name 'v3.0.0'. While we develop version 3.1 in master branch, we maintain '3.0.fb' branch and push bug fixes there. We release 3.0.1, 3.0.2, etc. as tags on a '3.0.fb' branch with names 'v3.0.X'. Once we release 3.1, we stop maintaining '3.0.fb' branch. We don't have a concept of long term supported releases.

Here is an example:

```

1. -----+-----+-----> branch: master
2.   |           |
3.   |           +-----(*)-----(*)-----> branch 3.2.fb
4.   |           3.2.0     3.2.1

```

```

5.      |
6.      +-----(*)-----(*)-----(*)---> branch: 3.1.fb
7.          3.1.0      3.1.1      3.1.2

```

All bug fixes go to release branches need to first to go master, and back port from the newest release branch to older ones. Note that some release branches may have been cut but not yet releases, so don't forget to backport bugs to there too. (Developer note: use cherry-pick for back-porting and no merges. Creating a pull request against the branch is strongly recommended for CI validation ([example](#)), but push to the branch from command line.)

For example, a bug is fixed in master that needs to be backported to 3.1. Here is the order:

```

1. -----+-----+-----[fix]-> branch: master
2.      |           |
3.      |           |           V
4.      |           +-----(*)-----(*)-----[fix]---(*)-> branch 3.2.fb
5.      |           3.2.0     3.2.1     |   3.2.2
6.      |           V
7.      +-----(*)-----(*)-----(*)-----[fix]---(*)-> branch: 3.1.fb
8.          3.1.0     3.1.1     3.1.2           3.1.3

```

Before an official release, release branch is used within Facebook servers for two weeks or more. We encourage other people to try out release branches before they are officially released too. When an official release is made, you can assume that it has been stably running in production for two weeks.

## Maven Release

(Under construction)

# RocksDB Users and Their Use Cases

---

## Facebook

At Facebook, we use RocksDB as storage engines in multiple data management services and a backend for many different stateful services, including:

1. MyRocks – <https://github.com/MySQLOnRocksDB/mysql-5.6>
2. MongoRocks – <https://github.com/mongodb-partners/mongo-rocks>
3. ZippyDB – Facebook's distributed key-value store with Paxos-style replication, built on top of RocksDB.[1] <https://www.youtube.com/watch?v=DfiN7pG0D0khtt>
4. Laser – Laser is a high query throughput, low (millisecond) latency, key-value storage service built on top of RocksDB.[1]
5. Dragon – a distributed graph query engine.  
<https://code.facebook.com/posts/1737605303120405/dragon-a-distributed-graph-query-engine/>
6. Stylus – a low-level stream processing framework written in C++.[1]
7. LogDevice – a distributed data store for logs [2]

[1] <https://research.facebook.com/publications/realtime-data-processing-at-facebook/>

[2] <https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs/>

1. Libra – Blockchain <https://libra.org>

## LinkedIn

Two different use cases at LinkedIn are using RocksDB as a storage engine:

1. LinkedIn's follow feed for storing user's activities. Check out the blog post:  
<https://engineering.linkedin.com/blog/2016/03/followfeed-linkedin-s-feed-made-faster-and-smarter>
2. Apache Samza, open source framework for stream processing Learn more about those use cases in a Tech Talk by Ankit Gupta and Naveen Somasundaram:  
[http://www.youtube.com/watch?v=plqVp\\_0nSzg](https://www.youtube.com/watch?v=plqVp_0nSzg)

## Yahoo

Yahoo is using RocksDB as a storage engine for their biggest distributed data store Sherpa. Learn more about it here: <http://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights>

## CockroachDB

---

CockroachDB is an open-source geo-replicated transactional database. They are using RocksDB as their storage engine. Check out their github:  
<https://github.com/cockroachdb/cockroach>

## DNANexus

---

DNANexus is using RocksDB to speed up processing of genomics data. You can learn more from this great blog post by Mike Lin: <http://devblog.dnanexus.com/faster-bam-sorting-with-samtools-and-rocksdb/>

## Iron.io

---

Iron.io is using RocksDB as a storage engine for their distributed queueing system. Learn more from Tech Talk by Reed Allman: <http://www.youtube.com/watch?v=HTjt6oj-RL4>

## Tango Me

---

Tango is using RocksDB as a graph storage to store all users' connection data and other social activity data.

## Turn

---

Turn is using RocksDB as a storage layer for their key/value store, serving at peak 2.4MM QPS out of different datacenters. Check out our RocksDB Protobuf merge operator at: [https://github.com/vladb38/rocksdb\\_protobuf](https://github.com/vladb38/rocksdb_protobuf)

## Santanader UK/Cloudera Profession Services

---

Check out their blog post: <http://blog.cloudera.com/blog/2015/08/inside-santanders-near-real-time-data-ingest-architecture/>

## Airbnb

---

Airbnb is using RocksDB as a storage engine for their personalized search service. You can learn more about it here: <https://www.youtube.com/watch?v=ASQ6XMtogMs>

## Alluxio

---

Alluxio uses RocksDB to serve and scale file system metadata to beyond 1 Billion files. The detailed design and implementation is described in this engineering blog:

<https://www.alluxio.io/blog/scalable-metadata-service-in-alluxio-storing-billions-of-files/>

## Pinterest

---

Pinterest's Object Retrieval System uses RocksDB for storage:

[https://www.youtube.com/watch?v=MtFEVEs\\_2Vo](https://www.youtube.com/watch?v=MtFEVEs_2Vo)

## Smyte

---

Smyte uses RocksDB as the storage layer for their core key-value storage, high-performance counters and time-windowed HyperLogLog services.

## Rakuten Marketing

---

Rakuten Marketing uses RocksDB as the disk cache layer for the real-time bidding service in their Performance DSP.

## VWO, Wingify

---

VWO's Smart Code checker and URL helper uses RocksDB to store all the URLs where VWO's Smart Code is installed.

## quasardb

---

quasardb is a high-performance, distributed, transactional key-value database that integrates well with in-memory analytics engines such as Apache Spark. quasardb uses a heavily tuned RocksDB as its persistence layer.

## Netflix

---

Netflix Netflix uses RocksDB on AWS EC2 instances with local SSD drives to cache application data.

## TiKV

---

TiKV is a GEO-replicated, high-performance, distributed, transactional key-value database. TiKV is powered by Rust and Raft. TiKV uses RocksDB as its persistence layer.

## Apache Flink

---

Apache Flink uses RocksDB to store state locally on a machine.

## Dgraph

---

Dgraph is an open-source, scalable, distributed, low latency, high throughput Graph database .They use RocksDB to store state locally on a machine.

## Uber

---

Uber uses RocksDB as a durable and scalable task queue.

## 360 Pika

---

360 Pika is a nosql compatible with redis. With the huge amount of data stored, redis may suffer for a capacity bottleneck, and pika was born for solving it. It has widely been widely used in many company

## LzLabs

---

LzLabs is using RocksDB as a storage engine in their multi-database distributed framework to store application configuration and user data.

## ProfaneDB

---

ProfaneDB is a database for Protocol Buffers, and uses RocksDB for storage. It is accessible via gRPC, and the schema is defined using directly .proto files.

## IOTA Foundation

---

IOTA Foundation is using RocksDB in the IOTA Reference Implementation (IRI) to store the local state of the Tangle. The Tangle is the first open-source distributed ledger powering the future of the Internet of Things.

## Avrio Project

---

Avrio Project is using RocksDB in Avrio to store blocks, account balances and data and other blockchain-releated data. Avrio is a multiblockchain decentralized cryptocurrency empowering monetary transactions.

## Crux

---

Crux is a document database that uses RocksDB for local EAV index storage to enable

point-in-time bitemporal Datalog queries. The “unbundled” architecture uses Kafka to provide horizontal scalability.

## Nebula Graph

---

Nebula Graph is a distributed, scalable, lightning-fast, open source graph database capable of hosting super large scale graphs with dozens of billions of vertices (nodes) and trillions of edges, with milliseconds of latency.

## Apache Hadoop Ozone

---

Ozone is a scalable, redundant, and distributed object store for Hadoop. Apart from scaling to billions of objects of varying sizes, Ozone can function effectively in containerized environments such as Kubernetes and YARN.

<https://blog.cloudera.com/apache-hadoop-ozone-object-store-architecture/>

# RocksDB Public Communication Channels

---

## RocksDB Official Website

---

<https://rocksdb.org/>

## Slack

---

[rocksdb.slack.com](https://rocksdb.slack.com) (You can apply or ask someone to invite to join)

## Google Group

---

<https://groups.google.com/forum/#!forum/rocksdb>

## Facebook Public Group (RocksDB Developers Public)

---

<https://www.facebook.com/groups/rocksdb.dev/>

## Twitter

---

<https://twitter.com/RocksDB>

## Meetup General Information

---

<https://www.meetup.com/RocksDB>

# Basic operations

The `rocksdb` library provides a persistent key value store. Keys and values are arbitrary byte arrays. The keys are ordered within the key value store according to a user-specified comparator function.

## Opening A Database

A `rocksdb` database has a name which corresponds to a file system directory. All of the contents of database are stored in this directory. The following example shows how to open a database, creating it if necessary:

```

1. #include <cassert>
2. #include "rocksdb/db.h"
3.
4. rocksdb::DB* db;
5. rocksdb::Options options;
6. options.create_if_missing = true;
7. rocksdb::Status status = rocksdb::DB::Open(options, "/tmp/testdb", &db);
8. assert(status.ok());
9. ...

```

If you want to raise an error if the database already exists, add the following line before the `rocksdb::DB::Open` call:

```
1. options.error_if_exists = true;
```

If you are porting code from `leveldb` to `rocksdb`, you can convert your `leveldb::Options` object to a `rocksdb::Options` object using `rocksdb::LevelDBOptions`, which has the same functionality as `leveldb::Options`:

```

1. #include "rocksdb/utilities/leveldb_options.h"
2.
3. rocksdb::LevelDBOptions leveldb_options;
4. leveldb_options.option1 = value1;
5. leveldb_options.option2 = value2;
6. ...
7. rocksdb::Options options = rocksdb::ConvertOptions(leveldb_options);

```

## RocksDB Options

Users can choose to always set options fields explicitly in code, as shown above. Alternatively, you can also set it through a string to string map, or an option string. See [Option String and Option Map](#).

Some options can be changed dynamically while DB is running. For example:

```
1. rocksdb::Status s;
2. s = db->SetOptions({{"write_buffer_size", "131072"}});
3. assert(s.ok());
4. s = db->SetDBOptions({{"max_background_flushes", "2"}});
5. assert(s.ok());
```

RocksDB automatically keeps options used in the database in OPTIONS-xxxx files under the DB directory. Users can choose to preserve the option values after DB restart by extracting options from these option files. See [RocksDB Options File](#).

## Status

You may have noticed the `rocksdb::Status` type above. Values of this type are returned by most functions in `rocksdb` that may encounter an error. You can check if such a result is ok, and also print an associated error message:

```
1. rocksdb::Status s = ...;
2. if (!s.ok()) cerr << s.ToString() << endl;
```

## Closing A Database

When you are done with a database, there are 2 ways to gracefully close the database -

1. Simply delete the database object. This will release all the resources that were held while the database was open. However, if any error is encountered when releasing any of the resources, for example error when closing the `info_log` file, it will be lost.
2. Call `DB::Close()`, followed by deleting the database object. The `DB::Close()` returns `Status`, which can be examined to determine if there were any errors. Regardless of errors, `DB::Close()` will release all resources and is irreversible.

Example:

```
1. ... open the db as described above ...
2. ... do something with db ...
3. delete db;
```

or

```
1. ... open the db as described above ...
2. ... do something with db ...
3. Status s = db->Close();
4. ... log status ...
5. delete db;
```

## Reads

The database provides `Put`, `Delete`, `Get`, and `MultiGet` methods to modify/query the database. For example, the following code moves the value stored under key1 to key2.

```

1. std::string value;
2. rocksdb::Status s = db->Get(rocksdb::ReadOptions(), key1, &value);
3. if (s.ok()) s = db->Put(rocksdb::WriteOptions(), key2, value);
4. if (s.ok()) s = db->Delete(rocksdb::WriteOptions(), key1);
```

Right now, value size must be smaller than 4GB. RocksDB also allows [Single Delete](#) which is useful in some special cases.

Each `Get` results into at least a memcpy from the source to the value string. If the source is in the block cache, you can avoid the extra copy by using a `PinnableSlice`.

```

1. PinnableSlice pinnable_val;
2. rocksdb::Status s = db->Get(rocksdb::ReadOptions(), key1, &pinnable_val);
```

The source will be released once `pinnable_val` is destructed or `::Reset` is invoked on it. Read more [here](#).

When reading multiple keys from the database, `MultiGet` can be used. There are two variations of `MultiGet` : 1. Read multiple keys from a single column family in a more performant manner, i.e it can be faster than calling `Get` in a loop, and 2. Read keys across multiple column families consistent with each other.

For example,

```

1. std::vector<Slice> keys;
2. std::vector<PinnableSlice> values;
3. std::vector<Status> statuses;
4.
5. for ... {
6.     keys.emplace_back(key);
7. }
8. values.resize(keys.size());
9. statuses.resize(keys.size());
10.
11. db->MultiGet(ReadOptions(), cf, keys.size(), keys.data(), values.data(), statuses.data());
```

In order to avoid the overhead of memory allocations, the `keys`, `values` and `statuses` above can be of type `std::array` on stack or any other type that provides contiguous storage.

Or

```
1. std::vector<ColumnFamilyHandle*> column_families;
```

```

2.   std::vector<Slice> keys;
3.   std::vector<std::string> values;
4.
5.   for ... {
6.     keys.emplace_back(key);
7.     column_families.emplace_back(column_family);
8.   }
9.   values.resize(keys.size());
10.
11.  std::vector<Status> statuses = db->MultiGet(ReadOptions(), column_families, keys, values);

```

For a more in-depth discussion of performance benefits of using MultiGet, see [MultiGet Performance](#).

## Writes

### Atomic Updates

Note that if the process dies after the Put of key2 but before the delete of key1, the same value may be left stored under multiple keys. Such problems can be avoided by using the `WriteBatch` class to atomically apply a set of updates:

```

1. #include "rocksdb/write_batch.h"
2. ...
3. std::string value;
4. rocksdb::Status s = db->Get(rocksdb::ReadOptions(), key1, &value);
5. if (s.ok()) {
6.   rocksdb::WriteBatch batch;
7.   batch.Delete(key1);
8.   batch.Put(key2, value);
9.   s = db->Write(rocksdb::WriteOptions(), &batch);
10. }

```

The `WriteBatch` holds a sequence of edits to be made to the database, and these edits within the batch are applied in order. Note that we called `Delete` before `Put` so that if `key1` is identical to `key2`, we do not end up erroneously dropping the value entirely.

Apart from its atomicity benefits, `WriteBatch` may also be used to speed up bulk updates by placing lots of individual mutations into the same batch.

### Synchronous Writes

By default, each write to `rocksdb` is asynchronous: it returns after pushing the write from the process into the operating system. The transfer from operating system memory to the underlying persistent storage happens asynchronously. The `sync` flag can be turned on for a particular write to make the write operation not return until the data

being written has been pushed all the way to persistent storage. (On Posix systems, this is implemented by calling either `fsync(...)` or `fdatasync(...)` or `msync(..., MS_SYNC)` before the write operation returns.)

```
1. rocksdb::WriteOptions write_options;
2. write_options.sync = true;
3. db->Put(write_options, ...);
```

## Non-sync Writes

With non-sync writes, RocksDB only buffers WAL write in OS buffer or internal buffer (when `options.manual_wal_flush = true`). They are often much faster than synchronous writes. The downside of non-sync writes is that a crash of the machine may cause the last few updates to be lost. Note that a crash of just the writing process (i.e., not a reboot) will not cause any loss since even when `sync` is false, an update is pushed from the process memory into the operating system before it is considered done.

Non-sync writes can often be used safely. For example, when loading a large amount of data into the database you can handle lost updates by restarting the bulk load after a crash. A hybrid scheme is also possible where `DB::SyncWAL()` is called by a separate thread.

We also provide a way to completely disable Write Ahead Log for a particular write. If you set `write_options.disableWAL` to true, the write will not go to the log at all and may be lost in an event of process crash.

RocksDB by default uses `fdatasync()` to sync files, which might be faster than `fsync()` in certain cases. If you want to use `fsync()`, you can set `Options::use_fsync` to true. You should set this to true on filesystems like ext3 that can lose files after a reboot.

## Advanced

For more information about write performance optimizations and factors influencing performance, see [Pipelined Write](#) and [Write Stalls](#).

## Concurrency

A database may only be opened by one process at a time. The `rocksdb` implementation acquires a lock from the operating system to prevent misuse. Within a single process, the same `rocksdb::DB` object may be safely shared by multiple concurrent threads. I.e., different threads may write into or fetch iterators or call `Get` on the same database without any external synchronization (the `rocksdb` implementation will automatically do the required synchronization). However other objects (like Iterator and WriteBatch) may require external synchronization. If two threads share such an object, they must protect access to it using their own locking protocol. More details are available in

the public header files.

## Merge operators

Merge operators provide efficient support for read-modify-write operation. More on the interface and implementation can be found on:

- [Merge Operator](#)
- [Merge Operator Implementation](#)
- [Get Merge Operands](#)

## Iteration

The following example demonstrates how to print all (key, value) pairs in a database.

```

1.  rocksdb::Iterator* it = db->NewIterator(rocksdb::ReadOptions());
2.  for (it->SeekToFirst(); it->Valid(); it->Next()) {
3.      cout << it->key().ToString() << ":" << it->value().ToString() << endl;
4.  }
5.  assert(it->status().ok()); // Check for any errors found during the scan
6.  delete it;

```

The following variation shows how to process just the keys in the range `[start, limit)` :

```

1.  for (it->Seek(start);
2.      it->Valid() && it->key().ToString() < limit;
3.      it->Next()) {
4.      ...
5.  }
6.  assert(it->status().ok()); // Check for any errors found during the scan

```

You can also process entries in reverse order. (Caveat: reverse iteration may be somewhat slower than forward iteration.)

```

1.  for (it->SeekToLast(); it->Valid(); it->Prev()) {
2.      ...
3.  }
4.  assert(it->status().ok()); // Check for any errors found during the scan

```

This is an example of processing entries in range `(limit, start]` in reverse order from one specific key:

```

1.  for (it->SeekForPrev(start);
2.      it->Valid() && it->key().ToString() > limit;
3.      it->Prev()) {
4.      ...
5.  }

```

```
6. assert(it->status().ok()); // Check for any errors found during the scan
```

See [SeekForPrev](#).

For explanation of error handling, different iterating options and best practice, see [Iterator](#).

To know about implementation details, see [Iterator's Implementation](#)

## Snapshots

Snapshots provide consistent read-only views over the entire state of the key-value store. `ReadOptions::snapshot` may be non-NULL to indicate that a read should operate on a particular version of the DB state.

If `ReadOptions::snapshot` is NULL, the read will operate on an implicit snapshot of the current state.

Snapshots are created by the `DB::GetSnapshot()` method:

```
1. rocksdb::ReadOptions options;
2. options.snapshot = db->GetSnapshot();
3. ... apply some updates to db ...
4. rocksdb::Iterator* iter = db->NewIterator(options);
5. ... read using iter to view the state when the snapshot was created ...
6. delete iter;
7. db->ReleaseSnapshot(options.snapshot);
```

Note that when a snapshot is no longer needed, it should be released using the `DB::ReleaseSnapshot` interface. This allows the implementation to get rid of state that was being maintained just to support reading as of that snapshot.

## Slice

The return value of the `it->key()` and `it->value()` calls above are instances of the `rocksdb::Slice` type. `Slice` is a simple structure that contains a length and a pointer to an external byte array. Returning a `Slice` is a cheaper alternative to returning a `std::string` since we do not need to copy potentially large keys and values. In addition, `rocksdb` methods do not return null-terminated C-style strings since `rocksdb` keys and values are allowed to contain '\0' bytes.

C++ strings and null-terminated C-style strings can be easily converted to a Slice:

```
1. rocksdb::Slice s1 = "hello";
2.
3. std::string str("world");
4. rocksdb::Slice s2 = str;
```

A Slice can be easily converted back to a C++ string:

```
1. std::string str = s1.ToString();
2. assert(str == std::string("hello"));
```

Be careful when using Slices since it is up to the caller to ensure that the external byte array into which the Slice points remains live while the Slice is in use. For example, the following is buggy:

```
1. rocksdb::Slice slice;
2. if (...) {
3.     std::string str = ...;
4.     slice = str;
5. }
6. Use(slice);
```

When the `if` statement goes out of scope, `str` will be destroyed and the backing storage for `slice` will disappear.

## Transactions

RocksDB now supports multi-operation transactions. See [Transactions](#)

## Comparators

The preceding examples used the default ordering function for key, which orders bytes lexicographically. You can however supply a custom comparator when opening a database. For example, suppose each database key consists of two numbers and we should sort by the first number, breaking ties by the second number. First, define a proper subclass of `rocksdb::Comparator` that expresses these rules:

```
1. class TwoPartComparator : public rocksdb::Comparator {
2. public:
3.     // Three-way comparison function:
4.     // if a < b: negative result
5.     // if a > b: positive result
6.     // else: zero result
7.     int Compare(const rocksdb::Slice& a, const rocksdb::Slice& b) const {
8.         int a1, a2, b1, b2;
9.         ParseKey(a, &a1, &a2);
10.        ParseKey(b, &b1, &b2);
11.        if (a1 < b1) return -1;
12.        if (a1 > b1) return +1;
13.        if (a2 < b2) return -1;
14.        if (a2 > b2) return +1;
15.        return 0;
16.    }
17. }
```

```

18. // Ignore the following methods for now:
19. const char* Name() const { return "TwoPartComparator"; }
20. void FindShortestSeparator(std::string*, const rocksdb::Slice&) const { }
21. void FindShortSuccessor(std::string*) const { }
22. };

```

Now create a database using this custom comparator:

```

1. TwoPartComparator cmp;
2. rocksdb::DB* db;
3. rocksdb::Options options;
4. options.create_if_missing = true;
5. options.comparator = &cmp;
6. rocksdb::Status status = rocksdb::DB::Open(options, "/tmp/testdb", &db);
7. ...

```

## Column Families

[Column Families](#) provide a way to logically partition the database. Users can provide atomic writes of multiple keys across multiple column families and read a consistent view from them.

## Bulk Load

You can [Creating and Ingesting SST files](#) to bulk load a large amount of data directly into DB with minimum impacts on the live traffic.

## Backup and Checkpoint

[Backup](#) allows users to create periodic incremental backups in a remote file system (think about HDFS or S3) and recover from any of them.

[Checkpoints](#) provides the ability to take a snapshot of a running RocksDB database in a separate directory. Files are hardlinked, rather than copied, if possible, so it is a relatively lightweight operation.

## I/O

By default, RocksDB's I/O goes through operating system's page cache. Setting [Rate Limiter](#) can limit the speed that RocksDB issues file writes, to make room for read I/Os.

Users can also choose to bypass operating system's page cache, using [Direct I/O](#).

See [IO](#) for more details.

# Backwards compatibility

The result of the comparator's `Name` method is attached to the database when it is created, and is checked on every subsequent database open. If the name changes, the `rocksdb::DB::Open` call will fail. Therefore, change the name if and only if the new key format and comparison function are incompatible with existing databases, and it is ok to discard the contents of all existing databases.

You can however still gradually evolve your key format over time with a little bit of pre-planning. For example, you could store a version number at the end of each key (one byte should suffice for most uses). When you wish to switch to a new key format (e.g., adding an optional third part to the keys processed by `TwoPartComparator`), (a) keep the same comparator name (b) increment the version number for new keys (c) change the comparator function so it uses the version numbers found in the keys to decide how to interpret them.

## MemTable and Table factories

By default, we keep the data in memory in skiplist memtable and the data on disk in a table format described here: [RocksDB Table Format](#).

Since one of the goals of RocksDB is to have different parts of the system easily pluggable, we support different implementations of both memtable and table format. You can supply your own memtable factory by setting `Options::memtable_factory` and your own table factory by setting `Options::table_factory`. For available memtable factories, please refer to `rocksdb/memtablerep.h` and for table factories to `rocksdb/table.h`. These features are both in active development and please be wary of any API changes that might break your application going forward.

You can also read more about memtables [here](#) and [here](#).

## Performance

Start with [Setup Options and Basic Tuning](#). For more information about RocksDB performance, see the "Performance" section in the sidebar in the right side.

## 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

`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](#).

`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

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:

```
1. rocksdb::Options options;
2. options.compression = rocksdb::kNoCompression;
3. ... rocksdb::DB::Open(options, name, ...) ....
```

Also [Dictionary Compression](#) is also available.

## 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.

```

1. #include "rocksdb/cache.h"
2. rocksdb::BlockBasedTableOptions table_options;
3. table_options.block_cache = rocksdb::NewLRUCache(100 * 1048576); // 100MB uncompressed cache
4.
5. rocksdb::Options options;
6. options.table_factory.reset(rocksdb::NewBlockBasedTableFactory(table_options));
7. rocksdb::DB* db;
8. rocksdb::DB::Open(options, name, &db);
9. ... use the db ...
10. 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:

```

1. rocksdb::ReadOptions options;
2. options.fill_cache = false;
3. rocksdb::Iterator* it = db->NewIterator(options);
4. for (it->SeekToFirst(); it->Valid(); it->Next()) {
5.     ...
6. }

```

You can also disable block cache by setting `options.no_block_cache` to true.

See [Block Cache](#) for more details.

## Key Layout

Note that the unit of disk transfer and caching is a block. Adjacent keys (according to the database sort order) will usually be placed in the same block. Therefore the application can improve its performance by placing keys that are accessed together near each other and placing infrequently used keys in a separate region of the key space.

For example, suppose we are implementing a simple file system on top of `rocksdb`. The types of entries we might wish to store are:

```

1. filename -> permission-bits, length, list of file_block_ids
2. file_block_id -> data

```

We might want to prefix `filename` keys with one letter (say '/') and the `file_block_id` keys with a different letter (say '0') so that scans over just the metadata do not force us to fetch and cache bulky file contents.

## Filters

Because of the way `rocksdb` data is organized on disk, a single `Get()` call may

involve multiple reads from disk. The optional `FilterPolicy` mechanism can be used to reduce the number of disk reads substantially.

```

1.   rocksdb::Options options;
2.   rocksdb::BlockBasedTableOptions bbto;
3.   bbto.filter_policy.reset(rocksdb::NewBloomFilterPolicy(
4.       10 /* bits_per_key */,
5.       false /* use_block_based_builder*/));
6.   options.table_factory.reset(rocksdb::NewBlockBasedTableFactory(bbto));
7.   rocksdb::DB* db;
8.   rocksdb::DB::Open(options, "/tmp/testdb", &db);
9.   ... use the database ...
10.  delete db;
11.  delete options.filter_policy;

```

The preceding code associates a `Bloom Filter` based filtering policy with the database. Bloom filter based filtering relies on keeping some number of bits of data in memory per key (in this case 10 bits per key since that is the argument we passed to `NewBloomFilter`). This filter will reduce the number of unnecessary disk reads needed for `Get()` calls by a factor of approximately a 100. Increasing the bits per key will lead to a larger reduction at the cost of more memory usage. We recommend that applications whose working set does not fit in memory and that do a lot of random reads set a filter policy.

If you are using a custom comparator, you should ensure that the filter policy you are using is compatible with your comparator. For example, consider a comparator that ignores trailing spaces when comparing keys. `NewBloomFilter` must not be used with such a comparator. Instead, the application should provide a custom filter policy that also ignores trailing spaces.

For example:

```

1.  class CustomFilterPolicy : public rocksdb::FilterPolicy {
2.  private:
3.    FilterPolicy* builtin_policy_;
4.  public:
5.    CustomFilterPolicy() : builtin_policy_(NewBloomFilter(10, false)) { }
6.    ~CustomFilterPolicy() { delete builtin_policy_; }
7.
8.    const char* Name() const { return "IgnoreTrailingSpacesFilter"; }
9.
10.   void CreateFilter(const Slice* keys, int n, std::string* dst) const {
11.     // Use builtin bloom filter code after removing trailing spaces
12.     std::vector<Slice> trimmed(n);
13.     for (int i = 0; i < n; i++) {
14.       trimmed[i] = RemoveTrailingSpaces(keys[i]);
15.     }
16.     return builtin_policy_->CreateFilter(&trimmed[i], n, dst);
17.   }
18. }
```

```

19.     bool KeyMayMatch(const Slice& key, const Slice& filter) const {
20.         // Use builtin bloom filter code after removing trailing spaces
21.         return builtin_policy_->KeyMayMatch(RemoveTrailingSpaces(key), filter);
22.     }
23. };

```

Advanced applications may provide a filter policy that does not use a bloom filter but uses some other mechanisms for summarizing a set of keys. See [rocksdb/filter\\_policy.h](#) for detail.

## Checksums

`rocksdb` associates checksums with all data it stores in the file system. There are two separate controls provided over how aggressively these checksums are verified:

- `ReadOptions::verify_checksums` forces checksum verification of all data that is read from the file system on behalf of a particular read. This is on by default.
- `Options::paranoid_checks` may be set to true before opening a database to make the database implementation raise an error as soon as it detects an internal corruption. Depending on which portion of the database has been corrupted, the error may be raised when the database is opened, or later by another database operation. By default, paranoid checking is on.

Checksum verification can also be manually triggered by calling `DB::VerifyChecksum()`. This API walks through all the SST files in all levels for all column families, and for each SST file, verifies the checksum embedded in the metadata and data blocks. At present, it is only supported for the BlockBasedTable format. The files are verified serially, so the API call may take a significant amount of time to finish. This API can be useful for proactive verification of data integrity in a distributed system, for example, where a new replica can be created if the database is found to be corrupt.

If a database is corrupted (perhaps it cannot be opened when paranoid checking is turned on), the `rocksdb::RepairDB` function may be used to recover as much of the data as possible.

## Compaction

RocksDB keeps rewriting existing data files. This is to clean stale versions of keys, and to keep the data structure optimal for reads.

The information about compaction has been moved to [Compaction](#). Users don't have to know internal of compactions before operating RocksDB.

## Approximate Sizes

The `GetApproximateSizes` method can be used to get the approximate number of bytes of file system space used by one or more key ranges.

```

1.   rocksdb::Range ranges[2];
2.   ranges[0] = rocksdb::Range("a", "c");
3.   ranges[1] = rocksdb::Range("x", "z");
4.   uint64_t sizes[2];
5.   db->GetApproximateSizes(ranges, 2, sizes);

```

The preceding call will set `sizes[0]` to the approximate number of bytes of file system space used by the key range `[a..c)` and `sizes[1]` to the approximate number of bytes used by the key range `[x..z)`.

## Environment

All file operations (and other operating system calls) issued by the `rocksdb` implementation are routed through a `rocksdb::Env` object. Sophisticated clients may wish to provide their own `Env` implementation to get better control. For example, an application may introduce artificial delays in the file IO paths to limit the impact of `rocksdb` on other activities in the system.

```

1.   class SlowEnv : public rocksdb::Env {
2.     .. implementation of the Env interface ...
3. };
4.
5.   SlowEnv env;
6.   rocksdb::Options options;
7.   options.env = &env;
8.   Status s = rocksdb::DB::Open(options, ...);

```

## Porting

`rocksdb` may be ported to a new platform by providing platform specific implementations of the types/methods/functions exported by `rocksdb/port/port.h`. See `rocksdb/port/port_example.h` for more details.

In addition, the new platform may need a new default `rocksdb::Env` implementation. See `rocksdb/util/env_posix.h` for an example.

## Manageability

To be able to efficiently tune your application, it is always helpful if you have access to usage statistics. You can collect those statistics by setting `Options::table_properties_collectors` or `Options::statistics`. For more information, refer to `rocksdb/table_properties.h` and `rocksdb/statistics.h`. These should not add significant

overhead to your application and we recommend exporting them to other monitoring tools. See [Statistics](#). You can also profile single requests using [Perf Context](#) and [IO Stats Context](#). Users can register [EventListener](#) for callbacks for some internal events.

## Purging WAL files

---

By default, old write-ahead logs are deleted automatically when they fall out of scope and application doesn't need them anymore. There are options that enable the user to archive the logs and then delete them lazily, either in TTL fashion or based on size limit.

The options are `Options::WAL_ttl_seconds` and `Options::WAL_size_limit_MB`. Here is how they can be used:

- If both set to 0, logs will be deleted asap and will never get into the archive.
- If `WAL_ttl_seconds` is 0 and `WAL_size_limit_MB` is not 0, WAL files will be checked every 10 min and if total size is greater than `WAL_size_limit_MB`, they will be deleted starting with the earliest until size\_limit is met. All empty files will be deleted.
- If `WAL_ttl_seconds` is not 0 and `WAL_size_limit_MB` is 0, then WAL files will be checked every `WAL_ttl_seconds / 2` and those that are older than `WAL_ttl_seconds` will be deleted.
- If both are not 0, WAL files will be checked every 10 min and both checks will be performed with ttl being first.

## Other Information

---

To set up RocksDB options:

- [Set Up Options And Basic Tuning](#)
- [Some detailed Tuning Guide](#)

Details about the `rocksdb` implementation may be found in the following documents:

- [RocksDB Overview and Architecture](#)
- [Format of an immutable Table file](#)
- [Format of a log file](#)

# Consistent View

If `ReadOptions.snapshot` is given, the iterator will return data as of the snapshot. If it is `nullptr`, the iterator will read from an implicit snapshot as of the time the iterator is created. The implicit snapshot is preserved by [pinning resource](#). There is no way to convert this implicit snapshot to an explicit snapshot.

# Error Handling

`Iterator::status()` returns the error of the iterating. The errors include I/O errors, checksum mismatch, unsupported operations, internal errors, or other errors.

If there is no error, the status is `Status::OK()`. If the status is not OK, the iterator will be invalidated too. In another word, if `Iterator::Valid()` is true, `status()` is guaranteed to be `OK()` so it's safe to proceed other operations without checking `status()`:

```

1. for (it->Seek("hello"); it->Valid(); it->Next()) {
2.   // Do something with it->key() and it->value().
3. }
4. if (!it->status().ok()) {
5.   // Handle error. it->status().ToString() contains error message.
6. }
```

On the other hand, if `Iterator::Valid()` is false, there are two possibilities: (1) We reached the end of the data. In this case, `status()` is `OK()`; (2) there is an error. In this case `status()` is not `OK()`. It is always a good practice to check `status()` if the iterator is invalidated.

`Seek()` and `SeekForPrev()` discard previous status.

Note that in release 5.13.x or earlier (before <https://github.com/facebook/rocksdb/pull/3810> which was merged on May 17, 2018) the behavior of `status()` and `valid()` used to be different:

- `Valid()` could return true even if `status()` is not ok. This could sometimes be used to skip over corrupted data. This is not supported anymore. The intended way of dealing with corrupted data is `RepairDB()` (see `db.h`).
- `Seek()` and `SeekForPrev()` didn't always discard previous status. `Next()` and `Prev()` didn't always preserve non-ok status.

# Iterating upper bound and lower bound

A user can specify an upper bound of your range query by setting

`ReadOptions.iterate_upper_bound` for the read option passed to `NewIterator()`. By setting this

option, RocksDB doesn't have to find the next key after the upper bound. In some cases, some I/Os or computation can be avoided. In some specific workloads, the improvement can be significant. Note it applies to both of forward and backward iterating. The behavior is not defined when you do `SeekForPrev()` with a seek key higher than upper bound, or calling `SeekToLast()` with the last key to be higher than an iterator upper bound, although RocksDB will not crash.

Similarly, `ReadOptions.iterate_lower_bound` can be used with backward iterating to help RocksDB optimize the performance.

See the comment of the options for more information.

## Resource pinned by iterators and iterator refreshing

---

Iterators by themselves don't use much memory, but it can prevent some resource from being released. This includes:

1. memtables and SST files as of the creation time of the iterators. Even if some memtables and SST files are removed after flush or compaction, they are still preserved if an iterator pinned them.
2. data blocks for the current iterating position. These blocks will be kept in memory, either pinned in block cache, or in the heap if block cache is not set. Please note that although normally blocks are small, in some extreme cases, a single block can be quite large, if the value size is very large.

So the best use of iterator is to keep it short-lived, so that these resource is freed timely.

An iterator has some creation costs. In some use cases (especially memory-only cases), people want to avoid the creation costs of iterators by reusing iterators. When you are doing it, be aware that in case an iterator getting stale, it can block resource from being released. So make sure you destroy or refresh them if they are not used after some time, e.g. one second. When you need to treat this stale iterator, before release 5.7, you'll need to destroy the iterator and recreate it if needed. Since release 5.7, you can call an API `Iterator::Refresh()` to refresh it. By calling this function, the iterator is refreshed to represent the recent states, and the stale resource pinned previously is released.

## Prefix Iterating

---

Prefix iterator allows users to use bloom filter or hash index in iterator, in order to improve the performance. However, the feature has limitation and may return wrong results without reporting an error if misused. So we recommend you to use this feature carefully. For how to use the feature, see [Prefix Seek](#). Options `total_order_seek` and `prefix_same_as_start` are only applicable in prefix iterating.

## Read-ahead

RocksDB does automatic readahead and prefetches data on noticing more than 2 IOs for the same table file during iteration. The readahead size starts with 8KB and is exponentially increased on each additional sequential IO, up to a max of 256 KB. This helps in cutting down the number of IOs needed to complete the range scan. This automatic readahead is enabled only when `ReadOptions.readahead_size = 0` (default value). On Linux, `readahead` syscall is used in Buffered IO mode, and an `AlignedBuffer` is used in Direct IO mode to store the prefetched data. (Automatic iterator-readahead is available starting 5.12 for buffered IO and 5.15 for direct IO).

If your entire use case is dominated by iterating and you are relying on OS page cache (i.e using buffered IO), you can choose to turn on readahead manually by setting `DBOptions.advise_random_on_open = false`. This is more helpful if you run on hard drives or remote storage, but may not have much actual effects on directly attached SSD devices.

`ReadOptions.readahead_size` provides read-ahead support in RocksDB for very limited use cases. The limitation of this feature is that, if turned on, the constant cost of the iterator will be much higher. So you should only use it if you iterate a very large range of data, and can't work it around using other approaches. A typical use case will be that the storage is remote storage with very long latency, OS page cache is not available and a large amount of data will be scanned. By enabling this feature, every read of SST files will read-ahead data according to this setting. Note that one iterator can open each file per level, as well as all L0 files at the same time. You need to budget your read-ahead memory for them. And the memory used by the read-ahead buffer can't be tracked automatically.

We are looking for improving read-ahead in RocksDB.

# Why Prefix Seek?

Normally, when an iterator seek is executed, RocksDB needs to place the position at every sorted run (memtable, level-0 file, other levels, etc) to the seek position and merge these sorted runs. This can sometimes involve several I/O requests. It is also CPU heavy for decompression of several data blocks and other CPU overheads.

Prefix seek is a feature for mitigating these overheads for some use cases. The basic idea is that, if users know the iterating will be within one key prefix, the common prefix can be used to reduce costs. The most commonly used prefix iterating technique is prefix bloom filter. If many sorted runs don't contain any entry for this prefix, it can be filtered out by a bloom filter, and some I/Os and CPU for the sorted run can be ignored.

Whether prefix seek can improve performance is workload dependent. It's more likely to be effective for very short iterator queries than longer ones.

## Defining a “prefix”

“Prefix” is defined by `options.prefix_extractor`, which is a `shared_pointer` of a `SliceTransform` instance. By calling `SliceTransform.Transform()` against a key, we extract a `Slice` representing a substring of the `Slice`, usually the prefix part. In this wiki page, we use “prefix” to refer to the output of `options.prefix_extractor.Transform()` of a key. You can use fixed length prefix transformer, by calling

`NewFixedPrefixTransform(prefix_len)`, a capped length prefix transformer, by calling `NewCappedPrefixTransform(prefix_len)`, or you can implement your own prefix transformer in the way you want and pass it to `options.prefix_extractor`. Prefix extractor is related to comparator. A prefix extractor needs to be used with a comparator where keys with the same prefix are close to each others in the total order defined by the comparator.

The recommendation is to use capped prefix transform with bytewise or reverse bytewise comparators when possible. It will maximize features supported with relatively good performance.

## Configure prefix bloom filter

Although it is not the only way users can take advantage of prefix iterating, prefix bloom filter in block-based SST file is the most commonly used and effective one. Here is an example how you can set up prefix bloom filters in SST files.

```

1. Options options;
2.
3. // Set up bloom filter
4. rocksdb::BlockBasedTableOptions table_options;
```

```

5. table_options.filter_policy.reset(rocksdb::NewBloomFilterPolicy(10, false));
   table_options.whole_key_filtering = false; // If you also need Get() to use whole key filters, leave it to
6. true.
7. options.table_factory.reset(
   rocksdb::NewBlockBasedTableFactory(table_options)); // For multiple column family setting, set up
8. specific column family's ColumnFamilyOptions.table_factory instead.
9.
10. // Define a prefix. In this way, a fixed length prefix extractor. A recommended one to use.
11. options.prefix_extractor.reset(NewCappedPrefixTransform(3));
12.
13. DB* db;
14. Status s = DB::Open(options, "/tmp/rocksdb", &db);

```

How this bloom filter is used depends on `ReadOptions` setting in read queries. See the section below for details.

`options.prefix_extractor` can be changed when DB is restarted. Usually the end result would be that bloom filters in existing SST files will be ignored in reads. In some cases, old bloom filters can still be used. This will be explained in following sections.

## Configure reads while prefix bloom filters are set up

### How to ignore prefix bloom filters in read

Users can only use prefix bloom filter to read inside a specific prefix. If iterator is outside a prefix, the feature needs to be disabled for specific iterators to prevent wrong results:

```

1. ReadOptions read_options;
2. read_options.total_order_seek = true;
3. Iterator* iter = db->NewIterator(read_options);
4. Slice key = "foobar";
5. iter->Seek(key); // Seek "foobar" in total order

```

Putting `read_options.total_order_seek = true` will make sure the query returns the same result as if there is no prefix bloom filter.

## Adaptive Prefix Mode

Since Release 6.8, a new adaptive mode is introduced:

```

1. ReadOptions read_options;
2. read_options.auto_prefix_mode = true;
3. Iterator* iter = db->NewIterator(read_options);
4. // .....

```

This will always generate the same result as `read_options.total_order_seek = true`, while prefix bloom filter might be used, based on seek key and iterator upper bound. It is the recommended way of using prefix bloom filter, because it's less prone to misuse. We will turn this option as default on once it is proven to be stable.

Here is an example how this feature can be used: for a fixed length prefix extractor of 3, following queries will take advantage of bloom filters:

```

1. options.prefix_extractor.reset(NewCappedPrefixTransform(3));
2. options.comparator = BytewiseComparator(); // This is the default
3. // .....
4. ReadOptions read_options;
5. read_options.auto_prefix_mode = true;
6. std::string upper_bound;
7. Slice upper_bound_slice;
8.
9. // "foo2" and "foo9" share the same prefix "foo".
10. upper_bound = "foo9";
11. upper_bound_slice = Slice(upper_bound);
12. read_options.iterate_upper_bound = &upper_bound_slice;
13. Iterator* iter = db->NewIterator(read_options);
14. iter->Seek("foo2");
15.
16. // "foobar2" and "foobar9" share longer prefix than "foo".
17. upper_bound = "foobar9";
18. upper_bound_slice = Slice(upper_bound);
19. read_options.iterate_upper_bound = &upper_bound_slice;
20. Iterator* iter = db->NewIterator(read_options);
21. iter->Seek("foobar2");
22.
// "foo2" and "fop" doesn't share the same prefix, but "fop" is the successor key of prefix "foo" which is the
23. prefix of "foo2".
24. upper_bound = "fop";
25. upper_bound_slice = Slice(upper_bound);
26. read_options.iterate_upper_bound = &upper_bound_slice;
27. Iterator* iter = db->NewIterator(read_options);
28. iter->Seek("foo2");

```

This feature has some limitations:

- It only supports fixed and capped prefix extractor
- The comparator needs to implement `IsSameLengthImmediateSuccessor()`. The built-in byte-wise and reverse byte-wise comparators have it implemented.
- Right now only `Seek()` can automatically take advantage of the feature by comparing seek key with iterator upper bound. `SeekForPrev()` never uses prefix bloom filter.
- Some extra CPU is used to determine qualification of prefix bloom filter in each SST file to query.

## Manual prefix iterating

With this option, users determine their use case qualifies for prefix iterator and are responsible of never iterating outside iterator. **RocksDB will not return error when it is misused** and the iterating result will be undefined. Note that, when iterating outside the iterator range, the result might not be limited to missing keys. Some undefined result might include: deleted keys might show up, key ordering is not followed, or very slow queries. Also note that even data within the prefix range might not be correct if the iterator has moved out of the prefix range and come back again.

Right now this mode is the default for backward compatible reason, but users should be extra careful when using this mode.

How to use the manual mode:

```

1. options.prefix_extractor.reset(NewCappedPrefixTransform(3));
2.
3. ReadOptions read_options;
4. read_options.total_order_seek = false;
5. read_options.auto_prefix_mode = false;
6. Iterator* iter = db->NewIterator(read_options);
7.
8. iter->Seek("foobar");
9. // Iterate within prefix "foo"
10.
11. iter->SeekForPrev("foobar");
12. // iterate within prefix "foo"
```

## Prefix extractor change

If prefix extractor changes when DB restarts, some SST files may contain prefix bloom filters generated using different prefix extractor than the one in current options. When opening those files, RocksDB will compare prefix extractor name stored in the properties of the SST files. If the name is different from the prefix extractor provided in the options, the prefix bloom filter is not used for this file, with following exception: if the previous SST file uses fixed or capped prefix extractor, the filter may sometimes be used if seek key and upper bound indicates that the iterator is within a prefix specified by previous prefix extractor. The behavior within the SST file would be the same as automatic prefix mode.

## Other prefix iterating features

Besides prefix bloom filter. There are several prefix iterating features:

- Prefix bloom filter in memtable. Turn it on with `options.memtable_prefix_bloom_size_ratio`
- Prefix memtables. Hash linked list and hash skip list memtables are supported. See [MemTable](#) for details.
- Prefix hash index in block based table format. See [Data Block Hash Index](#) for details.

- PlainTable format. See [PlainTable Format](#) for details.

## General Prefix Seek API

We introduced how to use prefix bloom filter in detail above. The usage is almost the same for other prefix iterating features. The only difference is that, some options might not be supported. Here is the general workflow:

When `options.prefix_extractor` for your DB or column family is specified, RocksDB is in a "prefix seek" mode. Example of how to use it:

```

1. Options options;
2.
3. // <---- Enable some features supporting prefix extraction
4. options.prefix_extractor.reset(NewFixedPrefixTransform(3));
5.
6. DB* db;
7. Status s = DB::Open(options, "/tmp/rocksdb", &db);
8.
9. .....
10.
11. Iterator* iter = db->NewIterator(ReadOptions());
12. iter->Seek("foobar"); // Seek inside prefix "foo"
13. iter->Next(); // Find next key-value pair inside prefix "foo"
```

When `options.prefix_extractor` is not `nullptr` and with default `ReadOptions`, iterators are not guaranteed a total order of all keys, but only keys for the same prefix. When doing `Iterator.Seek(lookup_key)`, RocksDB will extract the prefix of `lookup_key`. If there is one or more keys in the database matching prefix of `lookup_key`, RocksDB will place the iterator to the key equal or larger than `lookup_key` of the same prefix, as for total ordering mode. If no key of the prefix equals or is larger than `lookup_key`, or after calling one or more `Next()`, we finish all keys for the prefix, we might return `Valid()=false`, or any key (might be non-existing) that is larger than the previous key. Setting `ReadOptions.prefix_same_as_start=true` guarantees the first of those two behaviors.

From release 4.11, we support `Prev()` in prefix mode, but only when the iterator is still within the range of all the keys for the prefix. The output of `Prev()` is not guaranteed to be correct when the iterator is out of the range.

When prefix seek mode is enabled, RocksDB will freely organize the data or build look-up data structures that can locate keys for specific prefix or rule out non-existing prefixes quickly. Here are some supported optimizations for prefix seek mode: prefix bloom for block based tables and mem tables, [hash-based mem tables](#), as well as [PlainTable](#) format. One example setting:

```

1. Options options;
2.
3. // Enable prefix bloom for mem tables
```

```

4. options.prefix_extractor.reset(NewFixedPrefixTransform(3));
5. options.memtable_prefix_bloom_size_ratio = 0.1;
6.
7. // Enable prefix hash for SST files
8. BlockBasedTableOptions table_options;
9. table_options.index_type = BlockBasedTableOptions::IndexType::kHashSearch;
10.
11. DB* db;
12. Status s = DB::Open(options, "/tmp/rocksdb", &db);
13.
14. .....
15.
16. auto iter = db->NewIterator(ReadOptions());
17. iter->Seek("foobar"); // Seek inside prefix "foo"

```

From release 3.5, we support a read option to allow RocksDB to use total order even if `options.prefix_extractor` is given. To enable the feature set `ReadOption.total_order_seek=true` to the read option passed when doing `NewIterator()`, example:

```

1. ReadOptions read_options;
2. read_options.total_order_seek = true;
3. auto iter = db->NewIterator(read_options);
4. Slice key = "foobar";
5. iter->Seek(key); // Seek "foobar" in total order

```

Performance might be worse in this mode. Please be aware that not all implementations of prefix seek support this option. For example, some implementations of `PlainTable` doesn't support it and you'll see an error in status code when you try to use it. `Hash-based mem tables` might do a very expensive online sorting if you use it. This mode is supported in prefix bloom and hash index of block based tables.

From release 6.8, we support a similar option while allowing underlying implementation to take advantage of prefix specific information when not impacting results.

```

1. ReadOptions read_options;
2. read_options.auto_prefix_mode = true;
3. auto iter = db->NewIterator(read_options);
4. Slice key = "foobar";
5. iter->Seek(key); // Seek "foobar" in total order

```

Right now, the feature is only supported for forward seeking with prefix bloom filter.

## Limitation

`SeekToLast()` is not supported well with prefix iterating. `SeekToFirst()` is only supported by some configurations. You should use total order mode, if you will execute those types of queries against your iterator.

One common bug of using prefix iterating is to use prefix mode to iterate in reverse order. But it is not yet supported. If reverse iterating is your common query pattern, you can reorder the data to turn your iterating order to be forward. You can do it through implementing a customized comparator, or encode your key in a different way.

## API change from 2.8 -> 3.0

In this section, we explain the API as of 2.8 release and the change in 3.0.

### Before the Change

As of RocksDB 2.8, there are 3 seek modes:

#### Total Order Seek

This is the traditional seek behavior you'd expect. The seek performs on a total ordered key space, positioning the iterator to a key that is greater or equal to the target key you seek.

```
1. auto iter = db->NewIterator(ReadOptions());
2. Slice key = "foo_bar";
3. iter->Seek(key);
```

Not all table formats support total order seek. For example, the newly introduced [PlainTable](#) format only supports prefix-based seek() unless it is opened in total order mode (Options.prefix\_extractor == nullptr).

#### Use ReadOptions.prefix

This is the least flexible way to do a seek. Prefix needs to be supplied when creating an iterator.

```
1. Slice prefix = "foo";
2. ReadOptions ro;
3. ro.prefix = &prefix;
4. auto iter = db->NewIterator(ro);
5. Slice key = "foo_bar"
6. iter->Seek(key);
```

`Options.prefix_extractor` is a prerequisite. The `Seek()` is constrained to the prefix provided by `ReadOptions`, which means you will need to create a new iterator to seek a different prefix. The benefit of this approach is that irrelevant files are filtered out at the time of building the new iterator. So if you want to seek multiple keys with the same prefix, it might perform better. However, we consider this is a very rare use case.

## Use ReadOptions.prefix\_seek

This mode is more flexible than `ReadOption.prefix`. No pre-filtering is done at iterator creation time. As a result, the same iterator can be reused for seek of different key/prefix.

```

1. ReadOptions ro;
2. ro.prefix_seek = true;
3. auto iter = db->NewIterator(ro);
4. Slice key = "foo_bar";
5. iter->Seek(key);
```

Same as `ReadOptions.prefix`, `Options.prefix_extractor` is a prerequisite.

## What's Changed

It becomes obvious that 3 modes of seek are confusing:

- One mode would require another option to be set (e.g. `Options.prefix_extractor`);
- It is not obvious to our users which mode of the last two is preferred under different circumstances

This change tries to address this issue and makes things straight: by default, `Seek()` is performed in prefix mode if `Options.prefix_extractor` is defined and vice versa. The motivation is simple: if `Options.prefix_extractor` is provided, it is a very clear signal that underlying data can be sharded and prefix seek is a natural fit. Usage becomes unified:

```

1. auto iter = db->NewIterator(ReadOptions());
2. Slice key = "foo_bar";
3. iter->Seek(key);
```

## Transition to the New Usage

Transition to the new style should be simple: remove the assignment to `Options.prefix` or `Options.prefix_seek`, since they are deprecated. Now, seek directly with your target key or prefix. Since `Next()` can go across the boundary to a different prefix, you will need to check the end condition:

```

1. auto iter = DB::NewIterator(ReadOptions());
2. for (iter.Seek(prefix); iter.Valid() && iter.key().starts_with(prefix); iter.Next()) {
3.     // do something
4. }
```

# SeekForPrev API

Start from 4.13, Rocksdb added `Iterator::SeekForPrev()`. This new API will seek to the last key that is less than or equal to the target key, in contrast with `Seek()`.

```
1. // Suppose we have keys "a1", "a3", "b1", "b2", "c2", "c4".
2. auto iter = db->NewIterator(ReadOptions());
3. iter->Seek("a1");           // iter->Key() == "a1";
4. iter->Seek("a3");           // iter->Key() == "a3";
5. iter->SeekForPrev("c4"); // iter->Key() == "c4";
6. iter->SeekForPrev("c3"); // iter->Key() == "c2";
```

Basically, the behavior of `SeekForPrev()` is like the code snippet below:

```
1. Seek(target);
2. if (!Valid()) {
3.   SeekToLast();
4. } else if (key() > target) {
5.   Prev();
6. }
```

In fact, this API serves more than this code snippet. One typical example is, suppose we have keys, "a1", "a3", "a5", "b1" and enable `prefix_extractor` which take the first byte as prefix. If we want to get the last key less than or equal to "a6". The code above without `SeekForPrev()` doesn't work. Since after `seek("a6")` and `Prev()`, the iterator will enter the invalid state. But now, what you need is just a call, `SeekForPrev("a6")` and get "a5".

Also, `SeekForPrev()` API serves internally to support `Prev()` in prefix mode. Now, `Next()` and `Prev()` could both be mixed in prefix mode. Thanks to `SeekForPrev()`!

Since version 2.7, RocksDB supports a special type of iterator (named *tailing iterator*) optimized for a use case in which new data is read as soon as it's added to the database. Its key features are:

- Tailing iterator doesn't create a snapshot when it's created. Therefore, it can also be used to read newly added data (whereas ordinary iterators won't see any record added after the iterator was created).
- It's optimized for doing sequential reads – it might avoid doing potentially expensive seeks on SST files and immutable memtables in many cases.

To enable it, set `ReadOptions::tailing` to `true` when creating a new iterator. Note that tailing iterator currently supports only moving in the forward direction (in other words, `Prev()` and `SeekToLast()` are not supported).

Not all new data is guaranteed to be available to a tailing iterator. `Seek()` or `SeekToFirst()` on a tailing iterator can be thought of as creating an implicit snapshot – anything written after it may, but is not guaranteed to be seen.

## Implementation details

A tailing iterator provides a merged view of two internal iterators:

- a *mutable* iterator, used to access current memtable contents only
- an *immutable* iterator, used to read data from SST files and immutable memtables

Both of these internal iterators are created by specifying `kMaxSequenceNumber`, effectively disabling filtering based on internal sequence numbers and enabling access to records inserted after the creation of these iterators. In addition, each tailing iterator keeps track of the database's state changes (such as memtable flushes and compactions) and invalidates its internal iterators when it happens. This enables it to be always up-to-date.

Since SST files and immutable memtables don't change, a tailing iterator can often get away by performing a seek operation only on the mutable iterator. For this purpose, it maintains the interval `(prev_key, current_key]` currently covered by the immutable iterator (in other words, there are no records with key `k` such that `prev_key < k < current_key` neither in SST files nor immutable memtables). Therefore, when `Seek(target)` is called and `target` is within that interval, immutable iterator is already at the correct position and it is not necessary to move it.

RocksDB provides a way to delete or modify key/value pairs based on custom logic in background. It is handy for implementing custom garbage collection, like removing expired keys based on TTL, or dropping a range of keys in the background. It can also update the value of an existing key.

To use compaction filter, applications need to implement the `CompactionFilter` interface found in `rocksdb/compaction_filter.h` and set it to `ColumnFamilyOptions`. Alternatively, applications can implement the `CompactionFilterFactory` interface, which gives the flexibility to create different compaction filter instance per (sub)compaction. The compaction filter factory also gets to know some context from the compaction (whether it is a full compaction or whether it is a manual compaction) through the given `CompactionFilter::Context` param. The factory can choose to return different compaction filter based on the context.

```
1. options.compaction_filter = new CustomCompactionFilter();
2. // or
3. options.compaction_filter_factory.reset(new CustomCompactionFilterFactory());
```

The two ways of providing compaction filter also come with different thread-safety requirement. If a single compaction filter is provided to RocksDB, it has to be thread-safe since multiple sub-compactions can run in parallel, and they all make use of the same compaction filter instance. If a compaction filter factory is provided, each sub-compaction will call the factory to create a compaction filter instance. It is thus guaranteed that each compaction filter instance will be accessed from a single thread and thread-safety of the compaction filter is not required. The compaction filter factory, though, can be accessed concurrently by sub-compactions.

Compaction filter will not be invoked during flush, despite arguably flush is a special type of compaction.

There are two sets of API that can be implemented with compaction filter. The `Filter/FilterMergeOperand` API provide a simple callback to let compaction know whether to filter out a key/value pair. The `FilterV2` API extends the basic API by allowing changing the value, or dropping a range of keys starting from the current key.

Each time a (sub)compaction sees a new key from its input and when the value is a normal value, it invokes the compaction filter. Based on the result of the compaction filter:

- If it decides to keep the key, nothing will change.
- If it requests to filter the key, the value is replaced by a deletion marker. Note that if the output level of the compaction is the bottom level, no deletion marker will need to be output.
- If it requests to change the value, the value is replaced by the changed value.
- If it requests to remove a range of keys by returning `kRemoveAndSkipUntil`, the compaction will skip over to `skip_until` (means `skip_until` will be the next possible key output by the compaction). This one is tricky because in this case the compaction does not insert deletion marker for the keys it skips. This means

older version of the keys may reappears as a result. On the other hand, it is more efficient to simply dropping the keys, if the application know there aren't older versions of the keys, or reappearance of the older versions is fine.

If there are multiple versions of the same key from the input of the compaction, compaction filter will only be invoked once for the newest version. If the newest version is a deletion marker, compaction filter will not be invoked. However, it is possible the compaction filter is invoked on a deleted key, if the deletion marker isn't included in the input of the compaction.

When merge is being used, compaction filter is invoked per merge operand. The result of compaction filter is applied to the merge operand before merge operator is invoked.

Before release 6.0, if there is a snapshot taken later than the key/value pair, RocksDB always try to prevent the key/value pair from being filtered by compaction filter so that users can preserve the same view from a snapshot, unless the compaction filter returns `IgnoreSnapshots() = true`. However, this feature is deleted since 6.0, after realized that the feature has a bug which can't be easily fixed. Since release 6.0, with compaction filter enabled, RocksDB always invoke filtering for any key, even if it knows it will make a snapshot not repeatable.

This page describes the Atomic Read-Modify-Write operation in RocksDB, known as the "Merge" operation. It is an interface overview, aimed at the client or RocksDB user who has the questions: when and why should I use Merge; and how do I use Merge?

## Why

RocksDB is a high-performance embedded persistent key-value store. It traditionally provides three simple operations Get, Put and Delete to allow an elegant Lookup-table-like interface. <https://github.com/facebook/rocksdb/blob/master/include/rocksdb/db.h>

Often times, it's a common pattern to update an existing value in some ways. To do this in rocksdb, the client would have to read (Get) the existing value, modify it and then write (Put) it back to the db. Let's look at a concrete example.

Imagine we are maintaining a set of uint64 counters. Each counter has a distinct name. We would like to support four high level operations: Set, Add, Get and Remove.

First, we define the interface and get the semantics right. Error handling is brushed aside for clarity.

```

1. class Counters {
2. public:
3.   // (re)set the value of a named counter
4.   virtual void Set(const string& key, uint64_t value);
5.
6.   // remove the named counter
7.   virtual void Remove(const string& key);
8.
9.   // retrieve the current value of the named counter, return false if not found
10.  virtual bool Get(const string& key, uint64_t *value);
11.
12.  // increase the named counter by value.
13.  // if the counter does not exist, treat it as if the counter was initialized to zero
14.  virtual void Add(const string& key, uint64_t value);
15. };

```

Second, we implement it with the existing rocksdb support. Pseudo-code follows:

```

1. class RocksCounters : public Counters {
2. public:
3.   static uint64_t kDefaultCount = 0;
4.   RocksCounters(std::shared_ptr<DB> db);
5.
6.   // mapped to a RocksDB Put
7.   virtual void Set(const string& key, uint64_t value) {
8.     string serialized = Serialize(value);
9.     db_->Put(put_option_, key, serialized));
10. }
11.

```

```

12.     // mapped to a RocksDB Delete
13.     virtual void Remove(const string& key) {
14.         db_->Delete(delete_option_, key);
15.     }
16.
17.     // mapped to a RocksDB Get
18.     virtual bool Get(const string& key, uint64_t *value) {
19.         string str;
20.         auto s = db_->Get(get_option_, key, &str);
21.         if (s.ok()) {
22.             *value = Deserialize(str);
23.             return true;
24.         } else {
25.             return false;
26.         }
27.     }
28.
29.     // implemented as get -> modify -> set
30.     virtual void Add(const string& key, uint64_t value) {
31.         uint64_t base;
32.         if (!Get(key, &base)) {
33.             base = kDefaultValue;
34.         }
35.         Set(key, base + value);
36.     }
37. };

```

Note that, other than the Add operation, all other three operations can be mapped directly to a single operation in rocksdb. Coding-wise, it's not that bad. However, a conceptually single operation Add is nevertheless mapped to two rocksdb operations. This has performance implication too - random Get is relatively slow in rocksdb.

Now, suppose we are going to host Counters as a service. Given the number of cores of servers nowadays, our service is almost certainly multithreaded. If the threads are not partitioned by the key space, it's possible that multiple Add requests of the same counter, be picked up by different threads and executed concurrently. Well, if we also have strict consistency requirement (missing an update is not acceptable), we would have to wrap Add with external synchronization, a lock of some sort. The overhead adds up.

What if RocksDb directly supports the Add functionality? We might come up with something like this then:

```

1.     virtual void Add(const string& key, uint64_t value) {
2.         string serialized = Serialize(value);
3.         db->Add(add_option_, key, serialized);
4.     }

```

This seems reasonable for Counters. But not everything you store in RocksDB is a counter. Say we need to track the locations where a user has been to. We could store a

(serialized) list of locations as the value of a user key. It would be a common operation to add a new location to the existing list. We might want an Append operation in this case: `db->Append(user_key, serialize(new_location))`. This suggests that the semantics of the read-modify-write operation are really client value-type determined. To keep the library generic, we better abstract out this operation, and allow the client to specify the semantics. That brings us to our proposal: Merge.

## What

---

We have developed a generic Merge operation as a new first-class operation in RocksDB to capture the read-modify-write semantics.

This Merge operation:

- Encapsulates the semantics for read-modify-write into a simple abstract interface.
- Allows user to avoid incurring extra cost from repeated `Get()` calls.
- Performs back-end optimizations in deciding when/how to combine the operands without changing the underlying semantics.
- Can, in some cases, amortize the cost over all incremental updates to provide asymptotic increases in efficiency.

## How to Use It

---

In the following sections, the client-specific code changes are explained. We also provide a brief outline of how to use Merge. It is assumed that the reader already knows how to use classic RocksDB (or LevelDB), including:

- The DB class (including construction, `DB::Put()`, `DB::Get()`, and `DB::Delete()`)
- The Options class (and how to specify database options upon creation)
- Knowledge that all keys/values written to the database are simple strings of bytes.

## Overview of the Interface

---

We have defined a new interface/abstract-base-class: `MergeOperator`. It exposes some functions telling RocksDB how to combine incremental update operations (called “merge operands”) with base-values (Put/Delete). These functions can also be used to tell RocksDB how to combine merge operands with each other to form new merge operands (called “Partial” or “Associative” merging).

For simplicity, we will temporarily ignore this concept of Partial vs. non-Partial merging. So we have provided a separate interface called `AssociativeMergeOperator` which encapsulates and hides all of the details around partial merging. And, for most simple applications (such as in our 64-Bit Counters example above), this will suffice.

So the reader should assume that all merging is handled via an interface called `AssociativeMergeOperator`. Here is the public interface:

```

1.   // The Associative Merge Operator interface.
2.   // Client needs to provide an object implementing this interface.
3.   // Essentially, this class specifies the SEMANTICS of a merge, which only
4.   // client knows. It could be numeric addition, list append, string
5.   // concatenation, ... , anything.
6.   // The library, on the other hand, is concerned with the exercise of this
7.   // interface, at the right time (during get, iteration, compaction...)
8.   class AssociativeMergeOperator : public MergeOperator {
9.     public:
10.    virtual ~AssociativeMergeOperator() {}
11.
12.    // Gives the client a way to express the read -> modify -> write semantics
13.    // key:          (IN) The key that's associated with this merge operation.
14.    // existing_value:(IN) null indicates the key does not exist before this op
15.    // value:         (IN) the value to update/merge the existing_value with
16.    // new_value:     (OUT) Client is responsible for filling the merge result here
17.    // logger:        (IN) Client could use this to log errors during merge.
18.    //
19.    // Return true on success. Return false failure / error / corruption.
20.    virtual bool Merge(const Slice& key,
21.                      const Slice* existing_value,
22.                      const Slice& value,
23.                      std::string* new_value,
24.                      Logger* logger) const = 0;
25.
26.    // The name of the MergeOperator. Used to check for MergeOperator
27.    // mismatches (i.e., a DB created with one MergeOperator is
28.    // accessed using a different MergeOperator)
29.    virtual const char* Name() const = 0;
30.
31.    private:
32.    ...
33. };

```

### Some Notes:

- `AssociativeMergeOperator` is a sub-class of a class called `MergeOperator`. We will see later that the more generic `MergeOperator` class can be more powerful in certain cases. The `AssociativeMergeOperator` we use here is, on the other hand, a simpler interface.
- `existing_value` could be `nullptr`. This is useful in case the Merge operation is the first operation of a key. `nullptr` indicates that the 'existing' value does not exist. This basically defers to the client to interpret the semantics of a merge operation without a pre-value. Client could do whatever reasonable. For example, `Counters::Add` assumes a zero value, if none exists.
- We pass in the key so that client could multiplex the merge operator based on it, if the key space is partitioned and different subspaces refer to different types

of data which have different merge operation semantics. For example, the client might choose to store the current balance (a number) of a user account under the key "BAL:" and the history of the account activities (a list) under the key "HIS:uid", in the same DB. (Whether or not this is a good practice is debatable). For current balance, numeric addition is a perfect merge operator; for activity history, we would need a list append though. Thus, by passing the key back to the Merge callback, we allow the client to differentiate between the two types.

Example:

```

1. void Merge(...) {
2.     if (key start with "BAL:") {
3.         NumericAddition(...);
4.     } else if (key start with "HIS:") {
5.         ListAppend(...);
6.     }
7. }
```

## Other Changes to the client-visible interface

To use Merge in an application, the client must first define a class which inherits from the `AssociativeMergeOperator` interface (or the `MergeOperator` interface as we will see later). This object class should implement the functions of the interface, which will (eventually) be called by RocksDB at the appropriate time, whenever it needs to apply merging. In this way, the merge-semantics are completely client-specified.

After defining this class, the user should have a way to specify to RocksDB to use this merge operator for its merges. We have introduced additional fields/methods to the `DB` class and the `Options` class for this purpose:

```

1. // In addition to Get(), Put(), and Delete(), the DB class now also has an additional method: Merge().
2. class DB {
3.     ...
4.     // Merge the database entry for "key" with "value". Returns OK on success,
5.     // and a non-OK status on error. The semantics of this operation is
6.     // determined by the user provided merge_operator when opening DB.
7.     // Returns Status::NotSupported if DB does not have a merge_operator.
8.     virtual Status Merge(
9.         const WriteOptions& options,
10.        const Slice& key,
11.        const Slice& value) = 0;
12.    ...
13. };
14.
15. Struct Options {
16.     ...
17.     // REQUIRES: The client must provide a merge operator if Merge operation
18.     // needs to be accessed. Calling Merge on a DB without a merge operator
19.     // would result in Status::NotSupported. The client must ensure that the
```

```

20.     // merge operator supplied here has the same name and *exactly* the same
21.     // semantics as the merge operator provided to previous open calls on
22.     // the same DB. The only exception is reserved for upgrade, where a DB
23.     // previously without a merge operator is introduced to Merge operation
24.     // for the first time. It's necessary to specify a merge operator when
25.     // opening the DB in this case.
26.     // Default: nullptr
27.     const std::shared_ptr<MergeOperator> merge_operator;
28.     ...
29. };

```

**Note:** The Options::merge\_operator field is defined as a shared-pointer to a MergeOperator. As specified above, the AssociativeMergeOperator inherits from MergeOperator, so it is okay to specify an AssociativeMergeOperator here. This is the approach used in the following example.

## Client code change:

Given the above interface change, the client can implement a version of Counters that directly utilizes the built-in Merge operation.

### Counters v2:

```

1.     // A 'model' merge operator with uint64 addition semantics
2.     class UInt64AddOperator : public AssociativeMergeOperator {
3.     public:
4.         virtual bool Merge(
5.             const Slice& key,
6.             const Slice* existing_value,
7.             const Slice& value,
8.             std::string* new_value,
9.             Logger* logger) const override {
10.
11.             // assuming 0 if no existing value
12.             uint64_t existing = 0;
13.             if (existing_value) {
14.                 if (!Deserialize(*existing_value, &existing)) {
15.                     // if existing_value is corrupted, treat it as 0
16.                     Log(logger, "existing value corruption");
17.                     existing = 0;
18.                 }
19.             }
20.
21.             uint64_t oper;
22.             if (!Deserialize(value, &oper)) {
23.                 // if operand is corrupted, treat it as 0
24.                 Log(logger, "operand value corruption");
25.                 oper = 0;
26.             }
27.
28.             auto new = existing + oper;

```

```

29.         *new_value = Serialize(new);
30.         return true;           // always return true for this, since we treat all errors as "zero".
31.     }
32.
33.     virtual const char* Name() const override {
34.         return "UInt64AddOperator";
35.     }
36. };
37.
38. // Implement 'add' directly with the new Merge operation
39. class MergeBasedCounters : public RocksCounters {
40. public:
41.     MergeBasedCounters(std::shared_ptr<DB> db);
42.
43.     // mapped to a leveldb Merge operation
44.     virtual void Add(const string& key, uint64_t value) override {
45.         string serialized = Serialize(value);
46.         db_->Merge(merge_option_, key, serialized);
47.     }
48. };
49.
50. // How to use it
51. DB* dbp;
52. Options options;
53. options.merge_operator.reset(new UInt64AddOperator);
54. DB::Open(options, "/tmp/db", &dbp);
55. std::shared_ptr<DB> db(dbp);
56. MergeBasedCounters counters(db);
57. counters.Add("a", 1);
58. ...
59. uint64_t v;
60. counters.Get("a", &v);

```

The user interface change is relatively small. And the RocksDB back-end takes care of the rest.

## Associativity vs. Non-Associativity

Up until now, we have used the relatively simple example of maintaining a database of counters. And it turns out that the aforementioned `AssociativeMergeOperator` interface is generally pretty good for handling many use-cases such as this. For instance, if you wanted to maintain a set of strings, with an “append” operation, then what we’ve seen so far could be easily adapted to handle that as well.

So, why are these cases considered “simple”? Well, implicitly, we have assumed something about the data: associativity. This means we have assumed that:

- The values that are `Put()` into the RocksDB database have the same format as the merge operands called with `Merge()`; and
- It is okay to combine multiple merge operands into a single merge operand using

the same user-specified merge operator.

For example, look at the Counters case. The RocksDB database internally stores each value as a serialized 8-byte integer. So, when the client calls `Counters::Set` (corresponding to a `DB::Put()`), the argument is exactly in that format. And similarly, when the client calls `Counters::Add` (corresponding to a `DB::Merge()`), the merge operand is also a serialized 8-byte integer. This means that, in the client's `UInt64AddOperator`, the `*existing_value` may have corresponded to the original `Put()`, or it may have corresponded to a merge operand; it doesn't really matter! In all cases, as long as the `*existing_value` and `value` are given, the `UInt64AddOperator` behaves in the same way: it adds them together and computes the `*new_value`. And in turn, this `*new_value` may be fed into the merge operator later, upon subsequent merge calls.

By contrast, it turns out that RocksDB merge can be used in more powerful ways than this. For example, suppose we wanted our database to store a set of json strings (such as PHP arrays or objects). Then, within the database, we would want them to be stored and retrieved as fully formatted json strings, but we might want the "update" operation to correspond to updating a property of the json object. So we might be able to write code like:

```

1. ...
2. // Put/store the json string into to the database
3. db_->Put(put_option_, "json_obj_key",
4.           "{ employees: [ {first_name: john, last_name: doe}, {first_name: adam, last_name: smith}] }");
5.
6. ...
7.
8. // Use a pre-defined "merge operator" to incrementally update the value of the json string
9. db_->Merge(merge_option_, "json_obj_key", "employees[1].first_name = lucy");
10. db_->Merge(merge_option_, "json_obj_key", "employees[0].last_name = dow");

```

In the above pseudo-code, we see that the data would be stored in RocksDB as a json string (corresponding to the original `Put()`), but when the client wants to update the value, a "javascript-like" assignment-statement string is passed as the merge-operand. The database would store all of these strings as-is, and would expect the user's merge operator to be able to handle it.

Now, the `AssociativeMergeOperator` model cannot handle this, simply because it assumes the associativity constraints as mentioned above. That is, in this case, we have to distinguish between the base-values (json strings) and the merge-operands (the assignment statements); and we also don't have an (intuitive) way of combining the merge-operands into a single merge-operand. So this use-case does not fit into our "associative" merge model. That is where the generic `MergeOperator` interface becomes useful.

## The Generic MergeOperator interface

The MergeOperator interface is designed to support generality and also to exploit some of the key ways in which RocksDB operates in order to provide an efficient solution for “incremental updates”. As noted above in the json example, it is possible for the base-value types (Put() into the database) to be formatted completely differently than the merge operands that are used to update them. Also, we will see that it is sometimes beneficial to exploit the fact that some merge operands can be combined to form a single merge operand, while some others may not. It all depends on the client’s specific semantics. The MergeOperator interface provides a relatively simple way of providing these semantics as a client.

```

1.   // The Merge Operator
2.   //
3.   // Essentially, a MergeOperator specifies the SEMANTICS of a merge, which only
4.   // client knows. It could be numeric addition, list append, string
5.   // concatenation, edit data structure, ... , anything.
6.   // The library, on the other hand, is concerned with the exercise of this
7.   // interface, at the right time (during get, iteration, compaction...)
8.   class MergeOperator {
9.     public:
10.    virtual ~MergeOperator() {}
11.
12.    // Gives the client a way to express the read -> modify -> write semantics
13.    // key:          (IN) The key that's associated with this merge operation.
14.    // existing:     (IN) null indicates that the key does not exist before this op
15.    // operand_list:(IN) the sequence of merge operations to apply, front() first.
16.    // new_value:   (OUT) Client is responsible for filling the merge result here
17.    // logger:      (IN) Client could use this to log errors during merge.
18.    //
19.    // Return true on success. Return false failure / error / corruption.
20.    virtual bool FullMerge(const Slice& key,
21.                           const Slice* existing_value,
22.                           const std::deque<std::string>& operand_list,
23.                           std::string* new_value,
24.                           Logger* logger) const = 0;
25.
26.    struct MergeOperationInput { ... };
27.    struct MergeOperationOutput { ... };
28.    virtual bool FullMergeV2(const MergeOperationInput& merge_in,
29.                            MergeOperationOutput* merge_out) const;
30.
31.    // This function performs merge(left_op, right_op)
32.    // when both the operands are themselves merge operation types.
33.    // Save the result in *new_value and return true. If it is impossible
34.    // or infeasible to combine the two operations, return false instead.
35.    virtual bool PartialMerge(const Slice& key,
36.                             const Slice& left_operand,
37.                             const Slice& right_operand,
38.                             std::string* new_value,
39.                             Logger* logger) const = 0;
40.
41.    // The name of the MergeOperator. Used to check for MergeOperator
42.    // mismatches (i.e., a DB created with one MergeOperator is

```

```

43.     // accessed using a different MergeOperator)
44.     virtual const char* Name() const = 0;
45.
46.     // Determines whether the MergeOperator can be called with just a single
47.     // merge operand.
48.     // Override and return true for allowing a single operand. FullMergeV2 and
49.     // PartialMerge/PartialMergeMulti should be implemented accordingly to handle
50.     // a single operand.
51.     virtual bool AllowSingleOperand() const { return false; }
52. };

```

**Some Notes:**

- MergeOperator has two methods, `FullMerge()` and `PartialMerge()`. The first method is used when a Put/Delete is the `*existing_value` (or `nullptr`). The latter method is used to combine two-merge operands (if possible).
- AssociativeMergeOperator simply inherits from MergeOperator and provides private default implementations of these methods, while exposing a wrapper function for simplicity.
- In MergeOperator, the “`FullMerge()`” function takes in an `*existing_value` and a sequence (`std::deque`) of merge operands, rather than a single operand. We explain below.

## How do these methods work?

On a high level, it should be noted that any call to `DB::Put()` or `DB::Merge()` does not necessarily force the value to be computed or the merge to occur immediately. RocksDB will more-or-less lazily decide when to actually apply the operations (e.g.: the next time the user calls `Get()`, or when the system decides to do its clean-up process called “Compaction”). This means that, when the MergeOperator is actually invoked, it may have several “stacked” operands that need to be applied. Hence, the `MergeOperator::FullMerge()` function is given an `*existing_value` and a list of operands that have been stacked. The MergeOperator should then apply the operands one-by-one (or in whatever optimized way the client decides so that the final `*new_value` is computed as if the operands were applied one-by-one).

## Partial Merge vs. Stacking

Sometimes, it may be useful to start combining the merge-operands as soon as the system encounters them, instead of stacking them. The `MergeOperator::PartialMerge()` function is supplied for this case. If the client-specified operator can logically handle “combining” two merge-operands into a single operand, the semantics for doing so should be provided in this method, which should then return true. If it is not logically possible, then it should simply leave `*new_value` unchanged and return false.

Conceptually, when the library decides to begin its stacking and applying process, it first tries to apply the client-specified `PartialMerge()` on each pair of operands it

encounters. Whenever this returns false, it will instead resort to stacking, until it finds a Put/Delete base-value, in which case it will call the `FullMerge()` function, passing the operands as a list parameter. Generally speaking, this final `FullMerge()` call should return true. It should really only return false if there is some form of corruption or bad-data.

## How `AssociativeMergeOperator` fits in

As alluded to above, `AssociativeMergeOperator` inherits from `MergeOperator` and allows the client to specify a single merge function. It overrides `PartialMerge()` and `FullMerge()` to use this `AssociativeMergeOperator::Merge()`. It is then used for combining operands, and also when a base-value is encountered. That is why it only works under the “associativity” assumptions described above (and it also explains the name).

## When to allow a single merge operand

Typically a merge operator is invoked only if there are at least two merge operands to act on. Override `AllowSingleOperand()` to return true if you need the merge operator to be invoked even with a single operand. An example use case for this is if you are using merge operator to change the value based on a TTL so that it could be dropped during later compactions (may be using a compaction filter).

## JSON Example

Using our generic `MergeOperator` interface, we now have the ability to implement the json example.

```

1.   // A 'model' pseudo-code merge operator with json update semantics
2.   // We pretend we have some in-memory data-structure (called JsonDataStructure) for
3.   // parsing and serializing json strings.
4.   class JsonMergeOperator : public MergeOperator {           // not associative
5.   public:
6.     virtual bool FullMerge(const Slice& key,
7.                           const Slice* existing_value,
8.                           const std::deque<std::string>& operand_list,
9.                           std::string* new_value,
10.                          Logger* logger) const override {
11.     JsonDataStructure obj;
12.     if (existing_value) {
13.       obj.ParseFrom(existing_value->ToString());
14.     }
15.
16.     if (obj.IsValid()) {
17.       Log(logger, "Invalid json string after parsing: %s", existing_value->ToString().c_str());
18.       return false;
19.     }
20.   }
```

```

21.     for (const auto& value : operand_list) {
22.         auto split_vector = Split(value, " = ");           // "xyz[0] = 5" might return ["xyz[0]", 5] as an
23.         std::vector<etc>
24.         obj.SelectFromHierarchy(split_vector[0]) = split_vector[1];
25.         if (obj.IsValid()) {
26.             Log(logger, "Invalid json after parsing operand: %s", value.c_str());
27.             return false;
28.         }
29.     }
30.     obj.SerializeTo(new_value);
31.     return true;
32. }
33.
34.
35. // Partial-merge two operands if and only if the two operands
36. // both update the same value. If so, take the "later" operand.
37. virtual bool PartialMerge(const Slice& key,
38.                           const Slice& left_operand,
39.                           const Slice& right_operand,
40.                           std::string* new_value,
41.                           Logger* logger) const override {
42.     auto split_vector1 = Split(left_operand, " = ");           // "xyz[0] = 5" might return ["xyz[0]", 5] as an
43.     std::vector<etc>
44.     auto split_vector2 = Split(right_operand, " = ");
45.
46.     // If the two operations update the same value, just take the later one.
47.     if (split_vector1[0] == split_vector2[0]) {
48.         new_value->assign(right_operand.data(), right_operand.size());
49.         return true;
50.     } else {
51.         return false;
52.     }
53.
54.     virtual const char* Name() const override {
55.         return "JsonMergeOperator";
56.     }
57. };
58.
59. ...
60.
61. // How to use it
62. DB* db;
63. Options options;
64. options.merge_operator.reset(new JsonMergeOperator);
65. DB::Open(options, "/tmp/db", &db);
66. std::shared_ptr<DB> db_(db);
67. ...
68. // Put/store the json string into to the database
69. db_->Put(put_option_, "json_obj_key",
70.           "{ employees: [ {first_name: john, last_name: doe}, {first_name: adam, last_name: smith}] }");
71.
72. ...

```

```

73.
74.    // Use the "merge operator" to incrementally update the value of the json string
75.    db_->Merge(merge_option_, "json_obj_key", "employees[1].first_name = lucy");
76.    db_->Merge(merge_option_, "json_obj_key", "employees[0].last_name = dow");

```

## Error Handling

If the `MergeOperator::PartialMerge()` returns false, this is a signal to RocksDB that the merging should be deferred (stacked) until we find a Put/Delete value to `FullMerge()` with. However, if `FullMerge()` returns false, then this is treated as “corruption” or error. This means that RocksDB will usually reply to the client with a `Status::Corruption` message or something similar. Hence, the `MergeOperator::FullMerge()` method should only return false if there is absolutely no robust way of handling the error within the client logic itself. (See the `JsonMergeOperator` example)

For `AssociativeMergeOperator`, the `Merge()` method follows the same “error” rules as `MergeOperator::FullMerge()` in terms of error-handling. Return false only if there is no logical way of dealing with the values. In the `Counters` example above, our `Merge()` always returns true, since we can interpret any bad value as 0.

## Get Merge Operands

This is an API to allow for fetching all merge operands associated with a Key. The main motivation for this API is to support use cases where doing a full online merge is not necessary as it is performance sensitive. This API is available from version 6.4.0.

Example use-cases:

1. Storing a KV pair where V is a collection of sorted integers and new values may get appended to the collection and subsequently users want to search for a value in the collection.

Example KV:

Key: ‘Some-Key’ Value: [2], [3,4,5], [21,100], [1,6,8,9]

To store such a KV pair users would typically call the Merge API as:

- a. `db->Merge(WriteOptions(), ‘Some-Key’, ‘2’);`
- b. `db->Merge(WriteOptions(), ‘Some-Key’, ‘3,4,5’);`
- c. `db->Merge(WriteOptions(), ‘Some-Key’, ‘21,100’);`
- d. `db->Merge(WriteOptions(), ‘Some-Key’, ‘1,6,8,9’);`

and implement a Merge Operator that would simply convert the Value to [2,3,4,5,21,100,1,6,8,9] upon a Get API call and then search in the resultant value. In such a case doing the merge online is unnecessary and simply returning all the operands [2], [3,4,5], [21, 100] and [1,6,8,9] and then search through the sub-lists proves to be faster while saving CPU and achieving the same outcome.

2. Update subset of columns and read subset of columns - Imagine a SQL Table, a row

may be encoded as a KV pair. If there are many columns and users only updated one of them, we can use merge operator to reduce write amplification. While users only read one or two columns in the read query, this feature can avoid a full merging of the whole row, and save some CPU.

3. Updating very few attributes in a value which is a JSON-like document - Updating one attribute can be done efficiently using merge operator, while reading back one attribute can be done more efficiently if we don't need to do a full merge.

```

1. API:
2. // Returns all the merge operands corresponding to the key. If the
3. // number of merge operands in DB is greater than
4. // merge_operands_options.expected_max_number_of_operands
5. // no merge operands are returned and status is Incomplete. Merge operands
6. // returned are in the order of insertion.
7. // merge_operands- Points to an array of at-least
8. //           merge_operands_options.expected_max_number_of_operands and the
9. //           caller is responsible for allocating it. If the status
10. //          returned is Incomplete then number_of_operands will contain
11. //          the total number of merge operands found in DB for key.
12. virtual Status GetMergeOperands(
13.     const ReadOptions& options, ColumnFamilyHandle* column_family,
14.     const Slice& key, PinnableSlice* merge_operands,
15.     GetMergeOperandsOptions* get_merge_operands_options,
16.     int* number_of_operands) = 0;
17.
18. Example:
19. int size = 100;
20. int number_of_operands = 0;
21. std::vector values(size);
22. GetMergeOperandsOptions merge_operands_info;
23. merge_operands_info.expected_max_number_of_operands = size;
24. db_->GetMergeOperands(ReadOptions(), db_->DefaultColumnFamily(), "k1", values.data(), merge_operands_info,
25. &number_of_operands);

```

The above API returns all the merge operands corresponding to the key. If the number of merge operands in DB is greater than `merge_operands_options.expected_max_number_of_operands`, no merge operands are returned and status is Incomplete. Merge operands returned are in the order of insertion.

DB Bench has a benchmark that uses Example 1 to demonstrate the performance difference of doing an online merge and then operating on the collection vs simply returning the sub-lists and operating on the sub-lists. To run the benchmark the command is :

```
./db_bench -benchmarks=getmergeoperands --merge_operator=sortlist
```

The `merge_operator` used above is used to sort the data across all the sublists for the online merge case which happens automatically when Get API is called.

## Review and Best Practices

Altogether, we have described the Merge Operator, and how to use it. Here are a couple tips on when/how to use the MergeOperator and AssociativeMergeOperator depending on use-cases.

## When to use merge

---

If the following are true:

- You have data that needs to be incrementally updated.
- You would usually need to read the data before knowing what the new value would be.

Then use one of the two Merge operators as specified in this wiki.

## Associative Data

---

If the following are true:

- Your merge operands are formatted the same as your Put values, AND
- It is okay to combine multiple operands into one (as long as they are in the same order)

Then use **AssociativeMergeOperator**.

## Generic Merge

---

If either of the two associativity constraints do not hold, then use **MergeOperator**.

If there are some times where it is okay to combine multiple operands into one (but not always):

- Use **MergeOperator**
- Have the PartialMerge() function return true in cases where the operands can be combined.

## Tips

---

**Multiplexing:** While a RocksDB DB object can only be passed 1 merge-operator at the time of construction, your user-defined merge operator class can behave differently depending on the data passed to it. The key, as well as the values themselves, will be passed to the merge operator; so one can encode different “operations” in the operands themselves, and get the MergeOperator to perform different functions accordingly.

**Is my use-case Associative?:** If you are unsure of whether the “associativity” constraints apply to your use-case, you can ALWAYS use the generic MergeOperator. The AssociativeMergeOperator is a direct subclass of MergeOperator, so any use-case that

can be solved with the `AssociativeMergeOperator` can be solved with the more generic `MergeOperator`. The `AssociativeMergeOperator` is mostly provided for convenience.

## Useful Links

---

- [Merge+Compaction Implementation Details](#): For RocksDB engineers who want to know how `MergeOperator` affects their code.

# Introduction

In RocksDB 3.0, we added support for Column Families.

Each key-value pair in RocksDB is associated with exactly one Column Family. If there is no Column Family specified, key-value pair is associated with Column Family "default".

Column Families provide a way to logically partition the database. Some interesting properties:

- Atomic writes across Column Families are supported. This means you can atomically execute `Write({cf1, key1, value1}, {cf2, key2, value2})`.
- Consistent view of the database across Column Families.
- Ability to configure different Column Families independently.
- On-the-fly adding new Column Families and dropping them. Both operations are reasonably fast.

## API

### Backward compatibility

Although we needed to make drastic API changes to support Column Families, we still support the old API. You don't need to make any changes to upgrade your application to RocksDB 3.0. All key-value pairs inserted through the old API are inserted into the Column Family "default". The same is true for downgrade after an upgrade. If you never use more than one Column Family, we don't change any disk format, which means you can safely roll back to RocksDB 2.8. This is very important for our customers inside Facebook.

### Example usage

[https://github.com/facebook/rocksdb/blob/master/examples/column\\_families\\_example.cc](https://github.com/facebook/rocksdb/blob/master/examples/column_families_example.cc)

### Reference

#### 1. Options, ColumnFamilyOptions, DBOptions

Defined in `include/rocksdb/options.h`, `Options` structures define how RocksDB behaves and performs. Before, every option was defined in a single `Options` struct. Going forward, options specific to a single Column Family will be defined in `ColumnFamilyOptions` and options specific to the whole RocksDB instance will be defined in `DBOptions`. `Options` struct is inheriting both `ColumnFamilyOptions` and `DBOptions`, which means you can still use it to define all the options for a DB instance with a single

(default) column family.

1. `ColumnFamilyHandle`

Column Families are handled and referenced with a `ColumnFamilyHandle`. Think of it as a open file descriptor. You need to delete all `ColumnFamilyHandle`s before you delete your DB pointer. One interesting thing: Even if `ColumnFamilyHandle` is pointing to a dropped Column Family, you can continue using it. The data is actually deleted only after you delete all outstanding `ColumnFamilyHandle`s.

```
DB::Open(const DBOptions& db_options, const std::string& name, const std::vector<ColumnFamilyDescriptor>&
1. column_families, std::vector<ColumnFamilyHandle*>* handles, DB** dbptr);
```

When opening a DB in a read-write mode, you need to specify all Column Families that currently exist in a DB. If that's not the case, `DB::Open` call will return `Status::InvalidArgument()`. You specify Column Families with a vector of `ColumnFamilyDescriptor`s. `ColumnFamilyDescriptor` is just a struct with a Column Family name and `ColumnFamilyOptions`. Open call will return a `Status` and also a vector of pointers to `ColumnFamilyHandle`s, which you can then use to reference Column Families. Make sure to delete all `ColumnFamilyHandle`s before you delete the DB pointer.

```
DB::OpenForReadOnly(const DBOptions& db_options, const std::string& name, const
                     std::vector<ColumnFamilyDescriptor>& column_families, std::vector<ColumnFamilyHandle*>* handles, DB** dbptr,
1. bool error_if_log_file_exist = false)
```

The behavior is similar to `DB::Open`, except that it opens DB in read-only mode. One big difference is that when opening the DB as read-only, you don't need to specify all Column Families – you can only open a subset of Column Families.

```
DB::ListColumnFamilies(const DBOptions& db_options, const std::string& name, std::vector<std::string>*
1. column_families)
```

`ListColumnFamilies` is a static function that returns the list of all column families currently present in the DB.

```
DB::CreateColumnFamily(const ColumnFamilyOptions& options, const std::string& column_family_name,
1. ColumnFamilyHandle** handle)
```

Creates a Column Family specified with option and a name and returns `ColumnFamilyHandle` through an argument.

```
1. DropColumnFamily(ColumnFamilyHandle* column_family)
```

Drop the column family specified by `ColumnFamilyHandle`. Note that the actual data is not deleted until the client calls `delete column_family;`. You can still continue using the column family if you have outstanding `ColumnFamilyHandle` pointer.

```
DB::NewIterators(const ReadOptions& options, const std::vector<ColumnFamilyHandle*>& column_families,
1. std::vector<Iterator*>* iterators)
```

This is the new call, which enables you to create iterators on multiple Column Families that have consistent view of the database.

## WriteBatch

To execute multiple writes atomically, you need to build a `WriteBatch`. All API calls now also take `ColumnFamilyHandle*` to specify the Column Family you want to write to.

## All other API calls

All other API calls have a new argument `ColumnFamilyHandle*`, through which you can specify the Column Family.

## Implementation

The main idea behind Column Families is that they share the write-ahead log and don't share memtables and table files. By sharing write-ahead logs we get awesome benefit of atomic writes. By separating memtables and table files, we are able to configure column families independently and delete them quickly.

Every time a single Column Family is flushed, we create a new WAL (write-ahead log). All new writes to all Column Families go to the new WAL. However, we still can't delete the old WAL since it contains live data from other Column Families. We can delete the old WAL only when all Column Families have been flushed and all data contained in that WAL persisted in table files. This created some interesting implementation details and will create interesting tuning requirements. Make sure to tune your RocksDB such that all column families are regularly flushed. Also, take a look at `Options::max_total_wal_size`, which can be configured such that stale column families are automatically flushed.

RocksDB provide the user with APIs that can be used to create SST files that can be ingested later. This can be useful if you have a use case that needs to load the data quickly, but the process of creating the data can be done offline.

## Creating SST file

`rocksdb::SstFileWriter` can be used to create SST file. After creating a `SstFileWriter` object you can open a file, insert rows into it and finish.

This is an example of how to create SST file in `/home/usr/file1.sst`

```

1. Options options;
2.
3. SstFileWriter sst_file_writer(EnvOptions(), options);
4. // Path to where we will write the SST file
5. std::string file_path = "/home/usr/file1.sst";
6.
7. // Open the file for writing
8. Status s = sst_file_writer.Open(file_path);
9. if (!s.ok()) {
10.     printf("Error while opening file %s, Error: %s\n", file_path.c_str(),
11.            s.ToString().c_str());
12.     return 1;
13. }
14.
15. // Insert rows into the SST file, note that inserted keys must be
16. // strictly increasing (based on options.comparator)
17. for (...) {
18.     s = sst_file_writer.Put(key, value);
19.     if (!s.ok()) {
20.         printf("Error while adding Key: %s, Error: %s\n", key.c_str(),
21.                s.ToString().c_str());
22.         return 1;
23.     }
24. }
25.
26. // Close the file
27. s = sst_file_writer.Finish();
28. if (!s.ok()) {
29.     printf("Error while finishing file %s, Error: %s\n", file_path.c_str(),
30.            s.ToString().c_str());
31.     return 1;
32. }
33. return 0;

```

Now we have our SST file located at `/home/usr/file1.sst`.

Please note that:

- Options passed to `SstFileWriter` will be used to figure out the table type,

compression options, etc that will be used to create the SST file.

- The Comparator that is passed to the SstFileWriter must be exactly the same as the Comparator used in the DB that this file will be ingested into.
- Rows must be inserted in a strictly increasing order.

You can learn more about the SstFileWriter by checking

[include/rocksdb/sst\\_file\\_writer.h](#)

## Ingesting SST files

Ingesting an SST files is simple, all you need to do is to call

DB::IngestExternalFile() and pass the file paths as a vector of

`std::string`

```

1. IngestExternalFileOptions info;
2. // Ingest the 2 passed SST files into the DB
3. Status s = db_->IngestExternalFile({"~/home/usr/file1.sst", "~/home/usr/file2.sst"}, info);
4. if (!s.ok()) {
5.     printf("Error while adding file %s and %s, Error %s\n",
6.            file_path1.c_str(), file_path2.c_str(), s.ToString().c_str());
7.     return 1;
8. }
```

You can learn more by checking DB::IngestExternalFile() and DB::IngestExternalFiles() in [include/rocksdb/db.h](#). DB::IngestExternalFiles() ingests a collection of external SST files for **multiple** column families following the 'all-or-nothing' property. If the function returns Status::OK, then all files are ingested successfully for **all** column families of interest. If the function returns non-OK status, then none of the files are ingested into none of the column families.

## What happens when you ingest a file

When you call DB::IngestExternalFile() We will

- Copy or link the file into the DB directory
- block (not skip) writes to the DB because we have to keep a consistent db state so we have to make sure we can safely assign the right sequence number to all the keys in the file we are going to ingest
- If file key range overlap with memtable key range, flush memtable
- Assign the file to the best level possible in the LSM-tree
- Assign the file a global sequence number
- Resume writes to the DB

We pick the lowest level in the LSM-Tree that satisfies these conditions

- The file can fit in the level
- The file key range don't overlap with any keys in upper layers
- The file don't overlap with the outputs of running compactions going to this

level

### Global sequence number

Files created using `SstFileWriter` have a special field in their metablock called `global sequence number`, when this field is used, all the keys inside this file start acting as if they have such sequence number. When we ingest a file, we assign a sequence number to all the keys in this file. Before RocksDB 5.16, RocksDB always updates this global sequence number field in the metablock of the SST file using a random write. From RocksDB 5.16, RocksDB enables user to choose whether to update this field via `IngestExternalFileOptions::write_global_seqno`. If this field is false during ingestion, then RocksDB uses the information in MANIFEST to deduce the global sequence number when accessing the file. This can be useful if the underlying file system does not support random write or if users wish to minimize sync operations. If backward compatibility is the concern, set this option to true so that external SST files ingested by RocksDB 5.16 or newer can be opened by RocksDB 5.15 or older.

## Ingestion Behind

---

Starting from 5.5, `IngestExternalFile()` will load a list of external SST files with ingestion behind supported, which means duplicate keys will be skipped if `ingest_behind==true`. In this mode we will always ingest in the bottom mode level. Duplicate keys in the file being ingested to be skipped rather than overwriting existing data under that key.

### Use case

Back-fill of some historical data in the database without over-writing existing newer version of data. This option could only be used if the DB has been running with `allow_ingest_behind=true` since the dawn of time. All files will be ingested at the bottommost level with `seqno=0`.

It deletes the most recent version of a key, and whether older versions of the key will come back to life is undefined.

## Basic Usage

`SingleDelete` is a new database operation. In contrast to the conventional `Delete()` operation, the deletion entry is removed along with the value when the two are lined up in a compaction. Therefore, similar to `Delete()` method, `SingleDelete()` removes the database entry for a key, but has the prerequisites that the key exists and was not overwritten. Returns `OK` on success, and a non-`OK` status on error. It is not an error if key did not exist in the database. If a key is overwritten (by calling `Put()` multiple times), then the result of calling `SingleDelete()` on this key is undefined.

`SingleDelete()` only behaves correctly if there has been only one `Put()` for this key since the previous call to `SingleDelete()` for this key. This feature is currently an experimental performance optimization for a very specific workload. The following code shows how to use `SingleDelete` :

```

1. std::string value;
2. rocksdb::Status s;
3. db->Put(rocksdb::WriteOptions(), "foo", "bar1");
4. db->SingleDelete(rocksdb::WriteOptions(), "foo");
5. s = db->Get(rocksdb::ReadOptions(), "foo", &value); // s.IsNotFound()==true
6. db->Put(rocksdb::WriteOptions(), "foo", "bar2");
7. db->Put(rocksdb::WriteOptions(), "foo", "bar3");
8. db->SingleDelete(rocksdb::ReadOptions(), "foo", &value); // Undefined result

```

`SingleDelete` API is also available in `WriteBatch`. Actually, `DB::SingleDelete()` is implemented by creating a `WriteBatch` with only one operation, `SingleDelete`, in this batch. The following code snippet shows the basic usage of `WriteBatch::SingleDelete()` :

```

1. rocksdb::WriteBatch batch;
2. batch.Put(key1, value);
3. batch.SingleDelete(key1);
4. s = db->Write(rocksdb::WriteOptions(), &batch);

```

## Notes

- Callers have to ensure that `SingleDelete` only applies to a key having not been deleted using `Delete()` or written using `Merge()`. Mixing `SingleDelete()` operations with `Delete()` and `Merge()` can result in undefined behavior (other keys are not affected by this)
- `SingleDelete` is NOT compatible with cuckoo hash tables, which means you should not call `SingleDelete` if you set `options.memtable_factory` with `NewHashCuckooRepFactory`
- Consecutive single deletions are currently not allowed
- Consider setting `write_options.sync = true` ([Asynchronous Writes](#))

Users sometimes need to do large amount of background write. One example is that they want to load a large amount of data. Another one is that they want to do some data migration.

The best way to handle these cases is to [Creating and Ingesting SST files](#). However, there may be use cases where users are not able to load in batch and have to directly write the data to the DB continuously. When doing that, they may hit this problem: if they issue background writes as fast as possible, it will trigger DB's throttling mechanism (see [Write Stalls](#)), it will slowdown not only the background writes, but the online queries by users too.

Low Priority Write can help users manage use cases like this. Since version 5.6, users can set `writeOptions.low_pri=true` for background writes. RocksDB will do more aggressive throttling to low priority writes to make sure high priority writes won't hit stalls.

While DB is running, RocksDB will keep evaluating whether we have any compaction pressure by looking at outstanding L0 files and bytes pending compaction. If RocksDB thinks there is a compaction pressure, it will ingest artificial sleeping to low priority write so that the total rate of low priority write is small. By doing that, the total write rate will drop much earlier than the overall write throttling condition, so the qualify of service of high priority writes is more likely to be guaranteed.

In [Two Phase Commit](#), the slowdown of low priority writes is done in the prepare phase, rather than the commit phase.

# Rocksdb can be opened with Time to Live(TTL) support

## USE-CASES

This API should be used to open the db when key-values inserted are meant to be removed from the db in a non-strict 'ttl' amount of time therefore, this guarantees that key-values inserted will remain in the db for at least ttl amount of time and the db will make efforts to remove the key-values as soon as possible after ttl seconds of their insertion.

## BEHAVIOUR

- TTL is accepted in seconds
- (int32\_t)Timestamp(creation) is suffixed to values in Put internally
- Expired TTL values are deleted in compaction only:(Timestamp+ttl<time\_now)
- Get/Iterator may return expired entries(compaction not run on them yet)
- Different TTL may be used during different Opens
- Example: Open1 at t=0 with ttl=4 and insert k1,k2, close at t=2. Open2 at t=3 with ttl=5. Now k1,k2 should be deleted at t>=5
- read\_only=true opens in the usual read-only mode. Compactions will not be triggered(neither manual nor automatic), so no expired entries removed

## CONSTRAINTS

Not specifying/passing or non-positive TTL behaves like TTL = infinity

## !!!WARNING!!!

- Calling DB::Open directly to re-open a db created by this API will get corrupt values(timestamp suffixed) and no ttl effect will be there during the second Open, so use this API consistently to open the db
- Be careful when passing ttl with a small positive value because the whole database may be deleted in a small amount of time

## API

Defined in header <rocksdb/utilities/db\_ttl.h>

```
1. static Status DBWithTTL::Open(const Options& options, const std::string& name, StackableDB** dbptr,
2.                               int32_t ttl = 0, bool read_only = false);
```

RocksDB supports Transactions when using a TransactionDB or OptimisticTransactionDB. Transactions have a simple BEGIN/COMMIT/ROLLBACK api and allow applications to modify their data concurrently while letting RocksDB handle the conflict checking. RocksDB supports both pessimistic and optimistic concurrency control.

Note that RocksDB provides Atomicity by default when writing multiple keys via WriteBatch. Transactions provide a way to guarantee that a batch of writes will only be written if there are no conflicts. Similar to a WriteBatch, no other threads can see the changes in a transaction until it has been written (committed).

## TransactionDB

When using a TransactionDB, all keys that are written are locked internally by RocksDB to perform conflict detection. If a key cannot be locked, the operation will return an error. When the transaction is committed, it is guaranteed to succeed as long as the database is able to be written to.

A TransactionDB can be better for workloads with heavy concurrency compared to an OptimisticTransactionDB. However, there is a small cost to using a TransactionDB due to the locking overhead. A TransactionDB will do conflict checking for all write operations, including writes performed outside of a Transaction.

Locking timeouts and limits can be tuned in the TransactionDBOptions.

```
1. TransactionDB* txn_db;
2. Status s = TransactionDB::Open(options, path, &txn_db);
3.
4. Transaction* txn = txn_db->BeginTransaction(write_options, txn_options);
5. s = txn->Put("key", "value");
6. s = txn->Delete("key2");
7. s = txn->Merge("key3", "value");
8. s = txn->Commit();
9. delete txn;
```

The default write policy is WriteCommitted. The alternatives are WritePrepared and WriteUnprepared. Read more about them [here](#).

## OptimisticTransactionDB

Optimistic Transactions provide light-weight optimistic concurrency control for workloads that do not expect high contention/interference between multiple transactions.

Optimistic Transactions do not take any locks when preparing writes. Instead, they rely on doing conflict-detection at commit time to validate that no other writers have modified the keys being written by the current transaction. If there is a conflict with another write (or it cannot be determined), the commit will return an error and no keys will be written.

Optimistic concurrency control is useful for many workloads that need to protect against occasional write conflicts. However, this may not be a good solution for workloads where write-conflicts occur frequently due to many transactions constantly attempting to update the same keys. For these workloads, using a TransactionDB may be a better fit. An OptimisticTransactionDB may be more performant than a TransactionDB for workloads that have many non-transactional writes and few transactions.

```

1. DB* db;
2. OptimisticTransactionDB* txn_db;
3.
4. Status s = OptimisticTransactionDB::Open(options, path, &txn_db);
5. db = txn_db->GetBaseDB();
6.
7. OptimisticTransaction* txn = txn_db->BeginTransaction(write_options, txn_options);
8. txn->Put("key", "value");
9. txn->Delete("key2");
10. txn->Merge("key3", "value");
11. s = txn->Commit();
12. delete txn;

```

## Reading from a Transaction

Transactions also support easily reading the state of keys that are currently batched in a given transaction but not yet committed:

```

1. db->Put(write_options, "a", "old");
2. db->Put(write_options, "b", "old");
3. txn->Put("a", "new");
4.
5. vector<string> values;
6. vector<Status> results = txn->MultiGet(read_options, {"a", "b"}, &values);
7. // The value returned for key "a" will be "new" since it was written by this transaction.
8. // The value returned for key "b" will be "old" since it is unchanged in this transaction.

```

You can also iterate through keys that exist in both the db and the current transaction by using Transaction::GetIterator().

## Guarding against Read-Write Conflicts:

Call GetForUpdate() to read a key and make the read value a precondition for transaction commit.

```

1. // Start a transaction
2. txn = txn_db->BeginTransaction(write_options);
3.
4. // Read key1 in this transaction
5. Status s = txn->GetForUpdate(read_options, "key1", &value);
6.

```

```

7. // Write to key1 OUTSIDE of the transaction
8. s = db->Put(write_options, "key1", "value0");

```

If this transaction was created by a TransactionDB, the Put would either timeout or block until the transaction commits or aborts. If this transaction were created by an OptimisticTransactionDB(), then the Put would succeed, but the transaction would not succeed if txn->Commit() were called.

```

1. // Repeat the previous example but just do a Get() instead of a GetForUpdate()
2. txn = txn_db->BeginTransaction(write_options);
3.
4. // Read key1 in this transaction
5. Status s = txn->Get(read_options, "key1", &value);
6.
7. // Write to key1 OUTSIDE of the transaction
8. s = db->Put(write_options, "key1", "value0");
9.
10. // No conflict since transactions only do conflict checking for keys read using GetForUpdate().
11. s = txn->Commit();

```

Currently, GetForUpdate() is the only way to establish Read-Write conflicts, so it can be used in combination with iterators, for example.

## Setting a Snapshot

By default, Transaction conflict checking validates that no one else has written a key *after* the time the key was first written in this transaction. This isolation guarantee is sufficient for many use-cases. However, you may want to guarantee that no one else has written a key since the start of the transaction. This can be accomplished by calling SetSnapshot() after creating the transaction.

Default behavior:

```

1. // Create a txn using either a TransactionDB or OptimisticTransactionDB
2. txn = txn_db->BeginTransaction(write_options);
3.
4. // Write to key1 OUTSIDE of the transaction
5. db->Put(write_options, "key1", "value0");
6.
7. // Write to key1 IN transaction
8. s = txn->Put("key1", "value1");
9. s = txn->Commit();
10. // There is no conflict since the write to key1 outside of the transaction happened before it was written in
    this transaction.

```

Using SetSnapshot():

```

1. txn = txn_db->BeginTransaction(write_options);
2. txn->SetSnapshot();

```

```

3.
4. // Write to key1 OUTSIDE of the transaction
5. db->Put(write_options, "key1", "value0");
6.
7. // Write to key1 IN transaction
8. s = txn->Put("key1", "value1");
9. s = txn->Commit();
   // Transaction will NOT commit since key1 was written outside of this transaction after SetSnapshot() was
10. called (even though this write
11. // occurred before this key was written in this transaction).

```

Note that in the SetSnapshot() case of the previous example, if this were a TransactionDB, the Put() would have failed. If this were an OptimisticTransactionDB, the Commit() would fail.

## Repeatable Read

Similar to normal RocksDB DB reads, you can achieve repeatable reads when reading through a transaction by setting a Snapshot in the ReadOptions.

```

1. read_options.snapshot = db->GetSnapshot();
2. s = txn->GetForUpdate(read_options, "key1", &value);
3. ...
4. s = txn->GetForUpdate(read_options, "key1", &value);
5. db->ReleaseSnapshot(read_options.snapshot);

```

Note that Setting a snapshot in the ReadOptions only affects the version of the data that is read. This does not have any affect on whether the transaction will be able to be committed.

If you have called SetSnapshot(), you can read using the same snapshot that was set in the transaction:

```

1. read_options.snapshot = txn->GetSnapshot();
2. Status s = txn->GetForUpdate(read_options, "key1", &value);

```

## Tuning / Memory Usage

Internally, Transactions need to keep track of which keys have been written recently. The existing in-memory write buffers are re-used for this purpose. Transactions will still obey the existing **max\_write\_buffer\_number** option when deciding how many write buffers to keep in memory. In addition, using transactions will not affect flushes or compactions.

It is possible that switching to using a [Optimistic]TransactionDB will use more memory than was used previously. If you have set a very large value for **max\_write\_buffer\_number**, a typical RocksDB instance will could never come close to this maximum memory limit. However, an [Optimistic]TransactionDB will try to use as

many write buffers as allowed. But this can be tuned by either reducing `max_write_buffer_number` or by setting `max_write_buffer_size_to_maintain`. See [memtable](#) for more details about `max_write_buffer_size_to_maintain`.

**OptimisticTransactionDBs:** At commit time, optimistic transactions will use the in-memory write buffers for conflict detection. For this to be successful, the data buffered must be older than the changes in the transaction. If not, Commit will fail. To decrease the likelihood of Commit failures due to insufficient buffer history, increase `max_write_buffer_size_to_maintain`.

**TransactionDBs:** If `SetSnapshot()` is used, Put/Delete/Merge/GetForUpdate operations will first check the in-memory buffers to do conflict detection. If there isn't sufficient data history in the in-memory buffers, the SST files will then be checked. Increasing `max_write_buffer_size_to_maintain` will reduce the chance that SST files must be read during conflict detection.

## Save Points

In addition to `Rollback()`, Transactions can also be partially rolled back if `SavePoints` are used.

```

1. s = txn->Put("A", "a");
2. txn->SetSavePoint();
3. s = txn->Put("B", "b");
4. txn->RollbackToSavePoint()
5. s = txn->Commit()
6. // Since RollbackToSavePoint() was called, this transaction will only write key A and not write key B.

```

## Under the hood

A high-level, succinct overview of how transactions work under the hood.

## Read Snapshot

Each update in RocksDB is done by inserting an entry tagged with a monotonically increasing sequence number. Assigning a seq to `read_options.snapshot` will be used by (transactional or non-transactional) db to read only values with seq smaller than that, i.e., read snapshots → `DBImpl::GetImpl`

That aside, transactions can call `TransactionBaseImpl::SetSnapshot` which will invoke `DBImpl::GetSnapshot`. It achieves two goals:

1. Return the current seq: transactions will use the seq (instead of the seq of its written value) to check for write-write conflicts → `TransactionImpl::TryLock` → `TransactionImpl::ValidateSnapshot` → `TransactionUtil::CheckKeyForConflicts`
2. Makes sure that values with smaller seq will not be erased by compaction jobs, etc. (`snapshots_.GetAll`). Such marked snapshots must be released by the callee

```
(DBImpl::ReleaseSnapshot)
```

## Read-Write conflict detection

Read-Write conflicts can be prevented by escalating them to write-write conflict: doing reads via GetForUpdate (instead of Get).

## Write-Write conflict detection: pessimistic approach

Write-write conflicts are detected at the write time using a lock table.

Non-transactional updates (put, merge, delete) are internally run under a transaction. So every update is through transactions → TransactionDBImpl::Put

Every update acquires a lock beforehand → TransactionImpl::TryLock

TransactionLockMgr::TryLock has only 16 locks per column family → size\_t num\_stripes = 16

Commit simply writes the write batch to WAL as well as to Memtable by calling DBImpl::Write → TransactionImpl::Commit

To support distributed transactions, the clients can call Prepare after performing the writes. It writes the value to WAL but not to the MemTable, which allows recovery in the case of machine crash → TransactionImpl::Prepare If Prepare is invoked, Commit writes a commit marker to the WAL and writes values to MemTable. This is done by calling MarkWalTerminationPoint() before adding value to the write batch.

## Write-Write conflict detection: optimistic approach

Write-write conflicts are detected using seq of the latest values at the commit time.

Each update adds the key to an in-memory vector → TransactionDBImpl::Put and OptimisticTransactionImpl::TryLock

Commit links OptimisticTransactionImpl::CheckTransactionForConflicts as a callback to the write batch → OptimisticTransactionImpl::Commit which will be invoked in DBImpl::WriteImpl via write->CheckCallback

The conflict detection logic is implemented at TransactionUtil::CheckKeysForConflicts

- only checks for conflicts against keys present in memory and fails otherwise.
- conflict detection is done by checking the latest seq of each key (DBImpl::GetLatestSequenceForKey) against the seq used for writing it.

A snapshot captures a point-in-time view of the DB at the time it's created. Snapshots do not persist across DB restarts.

## API Usage

---

- Create a snapshot with the `GetSnapshot()` API.
- Read from a snapshot by setting `ReadOptions::snapshot`.
- When finished, release resources associated with the snapshot by calling `ReleaseSnapshot()`.

## Implementation

---

### Flush/compaction

### Representation

A snapshot is represented by a small object of `SnapshotImpl` class. It holds only a few primitive fields, like the seqnum at which the snapshot was taken.

Snapshots are stored in a linked list owned by `DBImpl`. One benefit is we can allocate the list node before acquiring the DB mutex. Then while holding the mutex, we only need to update list pointers. Additionally, `ReleaseSnapshot()` can be called on the snapshots in an arbitrary order. With linked list, we can remove a node from the middle without shifting.

### Scalability

The main downside of linked list is it cannot be binary searched despite its ordering. During flush/compaction, we have to scan the snapshot list when we need to find out the earliest snapshot to which a key is visible. When there are many snapshots, this scan can significantly slow down flush/compaction to the point of causing write stalls. We've noticed problems when snapshot count is in the hundreds of thousands.

DeleteRange is an operation designed to replace the following pattern where a user wants to delete a range of keys in the range `[start, end]` :

```
1. ...
2. Slice start, end;
3. // set start and end
4. auto it = db->NewIterator(ReadOptions());
5.
6. for (it->Seek(start); cmp->Compare(it->key(), end) < 0; it->Next()) {
7.   db->Delete(WriteOptions(), it->key());
8. }
9. ...
```

This pattern requires performing a range scan, which prevents it from being an atomic operation, and makes it unsuitable for any performance-sensitive write path. To mitigate this, RocksDB provides a native operation to perform this task:

```
1. ...
2. Slice start, end;
3. // set start and end
4. db->DeleteRange(WriteOptions(), start, end);
5. ...
```

Under the hood, this creates a range tombstone represented as a single kv, which significantly speeds up write performance. Read performance with range tombstones is competitive to the scan-and-delete pattern. (For a more detailed performance analysis, see [the DeleteRange blog post](#).

For implementation details, see [this page](#).

RocksDB supports atomic flush of multiple column families if the DB option `atomic_flush` is set to `true`. The execution result of flushing `multiple` column families is written to the MANIFEST with ‘all-or-nothing’ guarantee (logically). With atomic flush, either all or no memtables of the column families of interest are persisted to SST files and added to the database.

This can be desirable if data in multiple column families must be consistent with each other. For example, imagine there is one metadata column family `meta_cf`, and a data column family `data_cf`. Every time we write a new record to `data_cf`, we also write its metadata to `meta_cf`. `meta_cf` and `data_cf` must be flushed atomically. Database becomes inconsistent if one of them is persisted but the other is not. Atomic flush provides a good guarantee. Suppose at a certain time, kv1 exists in the memtables of `meta_cf` and kv2 exists in the memtables of `data_cf`. After atomically flushing these two column families, both kv1 and kv2 are persistent if the flush succeeds. Otherwise neither of them exist in the database.

Since atomic flush also goes through the `write_thread`, it is guaranteed that no flush can occur in the middle of write batch.

It’s easy to enable/disable atomic flush as a DB option. To open the DB with atomic flush enabled,

```

1. Options options;
2. ... // Set other options
3. options.atomic_flush = true;
4. DBOptions db_opts(options);
5. DB* db = nullptr;
6. Status s = DB::Open(db_opts, dbname, column_families, &handles, &db);

```

Currently the user is responsible for specifying the column families to flush atomically in the case of manual flush.

```

1. w_opts.disable_wal = true;
2. db->Put(w_opts, cf_handle1, key1, value1);
3. db->Put(w_opts, cf_handle2, key2, value2);
4. FlushOptions flush_opts;
5. Status s = db->Flush(flush_opts, {cf_handle1, cf_handle2});

```

In the case automatic flushes triggered internally by RocksDB, we currently flush all column families in the database for simplicity.

RocksDB supports shared access to a database directory in a primary-secondary mode. The primary instance is a regular RocksDB instance capable of read, write, flush and compaction. (Or just compaction since flush can be viewed as a special type of compaction of L0 files). The secondary instance is similar to read-only instance, since it supports read but not write, flush or compaction. However, the secondary instance is able to dynamically tail the MANIFEST and write-ahead-logs (WALs) of the primary and apply the related changes if applicable. User has to call

```
DB::TryCatchUpWithPrimary() explicitly at chosen times.
```

## Example usage

```

1. const std::string kDbPath = "/tmp/rocksdbtest";
2. ...
3. // Assume we have already opened a regular RocksDB instance db_primary
4. // whose database directory is kDbPath.
5. assert(db_primary);
6. Options options;
7. options.max_open_files = -1;
8. // Secondary instance needs its own directory to store info logs (LOG)
9. const std::string kSecondaryPath = "/tmp/rocksdb_secondary/";
10. DB* db_secondary = nullptr;
11. Status s = DB::OpenAsSecondary(options, kDbPath, kSecondaryPath, &db_secondary);
12. assert(!s.ok() || db_secondary);
13. // Let secondary **try** to catch up with primary
14. s = db_secondary->TryCatchUpWithPrimary();
15. assert(s.ok());
16. // Read operations
17. std::string value;
18. s = db_secondary->Get(ReadOptions(), "foo", &value);
19. ...

```

More detailed example can be found in examples/multi\_processes\_example.cc.

## Current Limitations and Caveats

- The secondary instance must be opened with `max_open_files = -1`, indicating the secondary has to keep all file descriptors open in order to prevent them from becoming inaccessible after the primary unlinks them, which does not work on some non-POSIX file systems. We have a plan to relax this limitation in the future.
- RocksDB relies heavily on compaction to improve read performance. If a secondary instance tails and applies the log files of the primary right before a compaction, then it is possible to observe worse read performance on the secondary instance than on the primary until the secondary tails the logs again and advances to the state after compaction.

The approximate size APIs allow a user to get a reasonably accurate guess of disk space and memory utilization of a key range.

## API Usage

The main APIs are `GetApproximateSizes()` and `GetApproximateMemTableStats()`. The former takes a `struct SizeApproximationOptions` as an argument. It has the following fields -

- `include_memtables` - Indicates whether to count the memory usage of a given key range in memtables towards the overall size of the key range.
- `include_files` - Indicates whether to count the size of SST files occupied by the key range towards the overall size. At least one of this or `include_memtables` must be set to `true`.
- `files_size_error_margin` - This option indicates the acceptable ratio of over/under estimation of file size to the actual file size. For example, a value of `0.1` means the approximate size will be within 10% of the actual size of the key range. The main purpose of this option is to make the calculation more efficient. Setting this to `-1.0` will force RocksDB to seek into SST files to accurately calculate the size, which will be more CPU intensive.

Example,

```

1. std::array<Range, NUM_RANGES> ranges;
2. std::array<uint64_t, NUM_RANGES> sizes;
3. SizeApproximationOptions options;
4.
5. options.include_memtables = true;
6. options.files_size_error_margin = 0.1;
7.
8. ranges[0].start = start_key1;
9. ranges[0].limit = end_key1;
10. ranges[1].start = start_key2;
11. ranges[1].limit = end_key2;
12.
13. Status s = GetApproximateSizes(options, column_family, ranges.data(), NUM_RANGES, sizes.data());
14. // sizes[0] and sizes[1] contain the size in bytes for the respective ranges

```

The size estimated within SST files is size on disk, which is compressed. The size estimated in memtable is memory usage. It's up to the user to determine whether it makes sense to add these values together.

The API counterpart for memtable usage is `GetApproximateMemTableStats()`, which returns the number of entries and total size of a given key range. Example,

```

1. Range range;
2. uint64_t count;
3. uint64_t size;
4.

```

```

5.     range.start = start_key;
6.     range.limit = end_key;
7.
8.     Status s = GetApproximateMemTableStats(column_family, range, &count, &size);

```

The `GetApproximateMemTableStats` is only supported for memtables created by `SkipListFactory`.

Note that the approximate size from SST files are size of compressed blocks. It might be significantly smaller than the actual key/value size.

## Implementation

### SST File Estimation

To estimate the size of a level, RocksDB first finds and adds up size of full files with in the range for non-L0 levels. Total size of partial files is also estimated. If they are not going to change the result based on `files_size_error_margin`, they are skipped. Otherwise, RocksDB dives into the partial files one by one to estimate size included in the file.

For each file, it searches the index and figure out sum of all block sizes with in the range. RocksDB only queries index blocks where offsets of each block are kept. RocksDB finds the block for the start and end key of the range, and take a difference of the offsets of the two blocks. In the case where the file is only partial in one side, 0 or file size is used for the open side.

After RocksDB finishes each file, it adds the size to the total size, and re-evaluate whether `files_size_error_margin` is guaranteed. The query ends as soon as the condition is met.

### SkipList Memtable Size Estimation

Size in memtable is estimated as estimated number of entries, multiplied by average entry size. Number of entries within the range is estimated by looking at branches taken in binary search reaching the entry. For both of start and end key of the range, we estimate number of entries smaller than the key in the skip list, and take difference between the two. Number of keys smaller than a key is estimated by considering `Next()` called in the binary search reaching the key. For every `Next()` call, estimated count increases. The higher level the `Next()` call it, the more estimated count increased. This is a very rough estimation, but is effective for distinguishing a large range and a small range.

This feature is in early, experimental stage. Both API and internal implementation are subject to change.

We plan to allow users to assign timestamps to their data. Users can choose the format/length/encoding of the timestamp, they just need to tell RocksDB how to parse. In order to use the feature, user has to pass a custom `Comparator` as an option to the database during open. Once opened, the timestamp format is fixed unless a full compaction is run. Within one DB, the length of timestamp has to be fixed.

In the current implementation, user-specified timestamp is stored between original user key and RocksDB internal sequence number, as follows.

```
1. |user key|timestamp|seqno|type|
2. |<-----internal key----->|
```

## API

At present, `Get()`, `Put()`, and `Write()` are supported. Details of the API can be found in `db.h`. This list is not complete, and we plan to add more API functions in the future.

## Open DB

```
1. class MyComparator : public Comparator {
2. public:
3.   // Compare lhs and rhs, taking timestamp, if exists, into consideration.
4.   int Compare(const Slice& lhs, const Slice& rhs) const override {...}
5.   // Compare two timestamps ts1 and ts2.
6.   int CompareTimestamp(const Slice& ts1, const Slice& ts2) const override {...}
7.   // Compare a and b after stripping timestamp from them.
8.   int CompareWithoutTimestamp(const Slice& a, const Slice& b) const override {...}
9. };
10.
11. int main() {
12.   MyComparator cmp;
13.   Options options;
14.   options.comparator = &cmp;
15.   // Set other fields of options and open DB
16.   ...
17.   return 0;
18. }
```

## Get

`Get()` with a timestamp `ts` specified in `ReadOptions` will return the most recent key/value whose timestamp is smaller than or equal to `ts`. Note that if the database

enables timestamp, then caller of `Get()` should set `ReadOptions.timestamp`.

```
1. ReadOptions read_opts;
2. read_opts.timestamp = &ts;
3. s = db->Get(read_opts, key, &value);
```

## Put

When using `Put` with user timestamp, the user needs to set `WriteOptions.timestamp`.

```
1. WriteOptions write_opts;
2. write_opts.timestamp = &ts;
3. s = db->Put(write_opts, key, value);
```

## Write

When using `Write` API, the user creates a `WriteBatch`. The user does not need to set `WriteOptions.timestamp`. Instead, the user specifies timestamp size when creating `WriteBatch`, and calls `WriteBatch::AssignTimestamp(...)` before calling `Write`. `WriteBatch::AssignTimestamp()` can assign one or multiple timestamps to the key/value pairs in the write batch.

```
1. const size_t timestamp_size = 8; // 8-byte timestamp
2. WriteBatch wb(reserved_bytes, max_bytes, timestamp_size);
3. wb.Put(key1, value1);
4. wb.Delete(key2);
5. wb.AssignTimestamp(ts);
6. s = db->Write(WriteOptions(), &wb);
```

- [Setup Options and Basic Tuning](#)
- [Option String and Option Map](#)
- [RocksDB Options File](#)

Besides writing code using [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](#)):

## Write Buffer Size

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

### 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  $2 \times$  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:

```
1. cf_options.write_buffer_size = 64 << 20;
```

See below for sharing memory across Column Families.

### 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:

```
1. db_options.db_write_buffer_size = 64 << 30;
```

## 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](#)).

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

```

1. auto cache = NewLRUCache(128 << 20);
2.
3. BlockBasedTableOptions table_options;
4. table_options.block_cache = cache;
5.
6. auto table_factory = new BlockBasedTableFactory(table_options);
7. 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](#).

## 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](#).

## 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.

If `Get()` is a common operation for your queries, you can configure the Bloom Filter,

for example with 10 bits per key:

```
1. table_options.filter_policy.reset(NewBloomFilterPolicy(10, false));
```

To learn more about Bloom Filters, see: [Bloom Filter](#).

## 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](#).

**NOTE:** Make sure to pass the same `rate_limiter` object to all the DB's in your process.

## SST File Manager

If you are using flash storage, we recommend users to mount the file system with the `discard` 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](#).

## 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:

```
1. cf_options.level_compaction_dynamic_level_bytes = true;
2. options.max_background_compactions = 4;
3. options.max_background_flushes = 2;
4. options.bytes_per_sync = 1048576;
5. options.compaction_pri = kMinOverlappingRatio;
6. table_options.block_size = 16 * 1024;
7. table_options.cache_index_and_filter_blocks = true;
8. table_options.pin_l0_filter_and_index_blocks_in_cache = true;
```

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

---

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](#).

Users pass options to RocksDB through *Options* class. Other than setting options in the *Options* class, there are two other ways to set it:

1. get an option class from an [option file](#).
2. get it from an option string by calling
3. get it from a string map

## Option String

To get an option from a string, call helper function `GetColumnFamilyOptionsFromString()` or `GetDBOptionsFromString()` with a string containing the information. There is also a special `GetBlockBasedTableOptionsFromString()` and `GetPlainTableOptionsFromString()` to get table specific option.

An example of an option string will like this:

```
1. table_factory=PlainTable;prefix_extractor=rocksdb.CappedPrefix.13;comparator=leveldb.BytewiseComparator;compressi
```

Each option will be given as `<option_name>:<option_value>` separated by `;`. To find the list of options supported, check [the section below](#).

## Options map

Similarly, users can get option classes from a string map, by calling helper function

`GetColumnFamilyOptionsFromMap()`, `GetDBOptionsFromMap()`, `GetBlockBasedTableOptionsFromMap()` or `GetPlainTableOptionsFromMap()`. The string to string map is passed in, which maps from the option name and the option value, as a plain text string. An example of string map for an option is like this:

```
1. std::unordered_map<std::string, std::string> cf_options_map = {
2.     {"write_buffer_size", "1"},
3.     {"max_write_buffer_number", "2"},
4.     {"min_write_buffer_number_to_merge", "3"},
5.     {"max_write_buffer_number_to_maintain", "99"},
6.     {"compression", "kSnappyCompression"},
7.     {"compression_per_level",
8.      "kNoCompression:",
9.      "kSnappyCompression:",
10.     "kZlibCompression:",
11.     "kBZip2Compression:",
12.     "kLZ4Compression:",
13.     "kLZ4HCCompression:",
14.     "kXpressCompression:",
15.     "kZSTD:",
16.     "kZSTDNotFinalCompression"},,
17.     {"bottommost_compression", "kLZ4Compression"},,
18.     {"compression_opts", "4:5:6:7"},,
19.     {"num_levels", "8"},,
```

```

20.      {"level0_file_num_compaction_trigger", "8"},  

21.      {"level0_slowdown_writes_trigger", "9"},  

22.      {"level0_stop_writes_trigger", "10"},  

23.      {"target_file_size_base", "12"},  

24.      {"target_file_size_multiplier", "13"},  

25.      {"max_bytes_for_level_base", "14"},  

26.      {"level_compaction_dynamic_level_bytes", "true"},  

27.      {"max_bytes_for_level_multiplier", "15.0"},  

28.      {"max_bytes_for_level_multiplier_additional", "16:17:18"},  

29.      {"max_compaction_bytes", "21"},  

30.      {"soft_rate_limit", "1.1"},  

31.      {"hard_rate_limit", "2.1"},  

32.      {"hard_pending_compaction_bytes_limit", "211"},  

33.      {"arena_block_size", "22"},  

34.      {"disable_auto_compactions", "true"},  

35.      {"compaction_style", "kCompactionStyleLevel"},  

36.      {"verify_checksums_in_compaction", "false"},  

37.      {"compaction_options_fifo", "23"},  

38.      {"max_sequential_skip_in_iterations", "24"},  

39.      {"inplace_update_support", "true"},  

40.      {"report_bg_io_stats", "true"},  

41.      {"compaction_measure_io_stats", "false"},  

42.      {"inplace_update_num_locks", "25"},  

43.      {"memtable_prefix_bloom_size_ratio", "0.26"},  

44.      {"memtable_huge_page_size", "28"},  

45.      {"bloom_locality", "29"},  

46.      {"max_successive_merges", "30"},  

47.      {"min_partial_merge_operands", "31"},  

48.      {"prefix_extractor", "fixed:31"},  

49.      {"optimize_filters_for_hits", "true"},  

50.    };

```

To find the list of options supported, check [the section below](#).

## Find the supported options in option string and option map

In both of option string and option map, option name maps the variable names in the target class, `DBOptions`, `ColumnFamilyOptions`, `BlockBasedTableOptions`, or `PlainTableOptions`. For `DBOptions` and `ColumnFamilyOptions`, you can find the list of them and their descriptions in two respective classes in the source file `options.h` of the source code of your release. For the other two options, you can find them in file `table.h`

Note, although most of the options in the option class are supported in the option string, there are exceptions. You can find the list of supported options in variable `db_options_type_info`, `cf_options_type_info` and `block_based_table_type_info` in the source file `options/options_helper.h` of the source code of your release.

If the option is a callback class, e.g. comparators, compaction filter, and merge operators, you will usually need to pass the pointer of the callback class as the

value, which will be casted in to an object.

There are exception. Some special callback classes are supported by the option string or map:

- Prefix extractor (option name `prefix_extractor` ), whose value can be passed as `rocksdb.FixedPrefix.<prefix_length>` or `rocksdb.CappedPrefix.<prefix_length>` .
- Filter policy (option name `filter_policy` ), whose value can be passed as `bloomfilter:<bits_per_key>:<use_block_based>`
- Table factory (option name `table_factory` ). The values will be either `BlockBasedTable` or `PlainTable` . Other than that, two special option string names are used to provide the options, `block_based_table_factory` or `plain_table_factory` . The value of the options will be the option string of BlockBasedTableOptions or PlainTableOptions.
- Memtable Factory (option name `memtable_factory` ). It can take value of `skip_list` , `prefix_hash` , `hash_linklist` , `vector` or `cuckoo` .

In RocksDB 4.3, we add a set of features that makes managing RocksDB options easier.

1. Each RocksDB database will now automatically persist its current set of options into a file on every successful call of DB::Open(), SetOptions(), and CreateColumnFamily() / DropColumnFamily().
2. [LoadLatestOptions\(\) / LoadOptionsFromFile\(\)](#): A function that constructs RocksDB options object from an options file.
3. [CheckOptionsCompatibility](#): A function that performs compatibility check on two sets of RocksDB options.

With the above options file support, developers no longer need to maintain the full set of options of a previously-created RocksDB instance. In addition, when changing options is needed, CheckOptionsCompatibility() can further make sure the resulting set of Options can successfully open the same RocksDB database without corrupting the underlying data.

## Example

Here's a running example showing how the new features can make managing RocksDB options easier. A more complete example can be found in [examples/options\\_file\\_example.cc](#).

Suppose we open a RocksDB database, create a new column family on-the-fly while the database is running, and then close the database:

```

1. s = DB::Open(rocksdb_options, path_to_db, &db);
2. ...
3. // Create column family, and rocksdb will persist the options.
4. ColumnFamilyHandle* cf;
5. s = db->CreateColumnFamily(ColumnFamilyOptions(), "new_cf", &cf);
6. ...
7. // close DB
8. delete cf;
9. delete db;

```

Since in RocksDB 4.3 or later, each RocksDB instance will automatically store its latest set of options into a options file, we can use that file to construct the options next time when we want to open the DB. This is different from RocksDB 4.2 or older version where we need to remember all the options of each the column families in order to successfully open a DB. Now let's see how it works.

First, we call LoadLatestOptions() to load the latest set of options used by the target RocksDB database:

```

1. ConfigOptions cfg_opts;
2. DBOptions loaded_db_opt;
3. std::vector<ColumnFamilyDescriptor> loaded_cf_descs;

```

```
4. LoadLatestOptions(cfg_opts, path_to_db, &loaded_db_opt, &loaded_cf_descs);
```

## Unsupported Options

Since C++ does not have reflection, the following user-defined functions and pointer-typed options will only be initialized with default values. Detailed information can be found in `rocksdb/utilities/options_util.h`:

```
1. * env
2. * memtable_factory
3. * compaction_filter_factory
4. * prefix_extractor
5. * comparator
6. * merge_operator
7. * compaction_filter
```

For those un-supported user-defined functions, developers will need to specify them manually. In this example, we initialize Cache in `BlockBasedTableOptions` and `CompactionFilter`:

```
1. for (size_t i = 0; i < loaded_cf_descs.size(); ++i) {
2.     auto* loaded_bbt_opt = loaded_cf_descs[0].options.table_factory->GetOptions<BlockBasedTableOptions>();
3.     loaded_bbt_opt->block_cache = cache;
4. }
5.
6. loaded_cf_descs[0].options.compaction_filter = new MyCompactionFilter();
```

Now we perform sanity check to make sure the set of options is safe to open the target database:

```
1. Status s = CheckOptionsCompatibility(cfg_opts, kDBPath, db_options, loaded_cf_descs);
```

If the return value indicates OK status, we can proceed and use the loaded set of options to open the target RocksDB database:

```
1. s = DB::Open(loaded_db_opt, kDBPath, loaded_cf_descs, &handles, &db);
```

## Ignoring unknown options

In cases where an options file of a newer version is used with an older RocksDB version (say, when downgrading due to a bug), the older RocksDB version might not know about some newer options. `ignore_unknown_options` flag can be used to handle such cases. By setting the `ConfigOptions.ignore_unknown_options=true`, unknown options will be ignored. By default it is set to `false`.

# RocksDB Options File Format

RocksDB options file is a text file that follows the [INI file format](#). Each RocksDB options file has one Version section, one DBOptions section, and one CFOptions and TableOptions section for each column family. Below is an example RocksDB options file. A complete example can be found in [examples/rocksdb\\_option\\_file\\_example.ini](#):

```
1. [Version]
2. rocksdb_version=4.3.0
3. options_file_version=1.1
4. [DBOptions]
5. stats_dump_period_sec=600
6. max_manifest_file_size=18446744073709551615
7. bytes_per_sync=8388608
8. delayed_write_rate=2097152
9. WAL_ttl_seconds=0
10. ...
11. [CFOptions "default"]
12. compaction_style=kCompactionStyleLevel
13. compaction_filter=nullptr
14. num_levels=6
15. table_factory=BlockBasedTable
16. comparator=leveldb.BytewiseComparator

17. compression_per_level=kNoCompression:kNoCompression:kNoCompression:kSnappyCompression:kSnappyCompression:kSnappyCompression
18. ...
19. [TableOptions/BlockBasedTable "default"]
20. format_version=2
21. whole_key_filtering=true
22. skip_table_builder_flush=false
23. no_block_cache=false
24. checksum=kCRC32C
25. filter_policy=rocksdb.BuiltinBloomFilter
26. ....
```

MemTable is an in-memory data-structure holding data before they are flushed to SST files. It serves both read and write - new writes always insert data to memtable, and reads has to query memtable before reading from SST files, because data in memtable is newer. Once a memtable is full, it becomes immutable and replace by a new memtable. A background thread will flush the content of the memtable into a SST file, after which the memtable can be destroyed.

The most important options that affect memtable behavior are:

- `memtable_factory` : The factory object of memtable. By specifying factory object user can change the underlying implementation of memtable, and provide implementation specific options.
- `write_buffer_size` : Size of a single memtable.
- `db_write_buffer_size` : Total size of memtables across column families. This can be used to manage the total memory used by memtables.
- `write_buffer_manager` : Instead of specifying a total size of memtables, user can provide their own write buffer manager to control the overall memtable memory usage. Overrides `db_write_buffer_size`.
- `max_write_buffer_number` : The maximum number of memtables build up in memory, before they flush to SST files.
- `max_write_buffer_size_to_maintain` : The amount of write history to maintain in memory, in bytes. This includes the current memtable size, sealed but unflushed memtables, and flushed memtables that are kept around. RocksDB will try to keep at least this much history in memory - if dropping a flushed memtable would result in history falling below this threshold, it would not be dropped.

The default implementation of memtable is based on skiplist. Other than the default memtable implementation, users can use other types of memtable implementation, for example HashLinkList, HashSkipList or Vector, to speed-up some queries.

## Skiplist MemTable

Skiplist-based memtable provides general good performance to both read and write, random access and sequential scan. Besides, it provides some other useful features that other memtable implementations don't currently support, like [Concurrent Insert](#) and [Insert with Hint](#).

## HashSkipList MemTable

As their names imply, HashSkipList organizes data in a hash table with each hash bucket to be a skip list, while HashLinkList organizes data in a hash table with each hash bucket as a sorted single linked list. Both types are built to reduce number of comparisons when doing queries. One good use case is to combine them with PlainTable SST format and store data in RAMFS.

When doing a look-up or inserting a key, target key's prefix is retrieved using `Options.prefix_extractor`, which is used to find the hash bucket. Inside a hash bucket,

all the comparisons are done using whole (internal) keys, just as SkipList based memtable.

The biggest limitation of the hash based memtables is that doing scan across multiple prefixes requires copy and sort, which is very slow and memory costly.

## Flush

There are three scenarios where memtable flush can be triggered:

1. Memtable size exceeds `write_buffer_size` after a write.
2. Total memtable size across all column families exceeds `db_write_buffer_size`, or `write_buffer_manager` signals a flush. In this scenario the largest memtable will be flushed.
3. Total WAL file size exceeds `max_total_wal_size`. In this scenario the memtable with the oldest data will be flushed, in order to allow the WAL file with data from this memtable to be purged.

As a result, a memtable can be flushed before it is full. This is one reason the generated SST file can be smaller than the corresponding memtable. Compression is another factor to make SST file smaller than corresponding memtable, since data in memtable is uncompressed.

## Concurrent Insert

Without support of concurrent insert to memtables, concurrent writes to RocksDB from multiple threads will apply to memtable sequentially. Concurrent memtable insert is enabled by default and can be turned off via `allow_concurrent_memtable_write` option, although only skiplist-based memtable supports the feature.

## Insert with Hint

## In-place Update

## Comparison

Mem Table Type	SkipList	HashSkipList	HashLinkedList	Vector
Optimized Use Case	General	Range query within a specific key prefix	Range query within a specific key prefix and there are only a small number of rows for each prefix	Random write heavy workload
Index type	binary search	hash + binary search	hash + linear search	linear search
Support		very costly (copy)	very costly (copy)	very costly (copy and

totally ordered full db scan?	naturally	and sort to create a temporary totally-ordered view)	and sort to create a temporary totally-ordered view)	sort to create a temporary totally-ordered view)
Memory Overhead	Average (multiple pointers per entry)	High (Hash Buckets + Skip List Metadata for non-empty buckets + multiple pointers per entry)	Lower (Hash buckets + pointer per entry)	Low (pre-allocated space at the end of vector)
MemTable Flush	Fast with constant extra memory	Slow with high temporary memory usage	Slow with high temporary memory usage	Slow with constant extra memory
Concurrent Insert	Support	Not support	Not support	Not support
Insert with Hint	Support (in case there are no concurrent insert)	Not support	Not support	Not support

## Overview

Every update to RocksDB is written to two places: 1) an in-memory data structure called memtable (to be flushed to SST files later) and 2) write ahead log(WAL) on disk. In the event of a failure, write ahead logs can be used to completely recover the data in the memtable, which is necessary to restore the database to the original state. In the default configuration, RocksDB guarantees process crash consistency by flushing the WAL after every user write.

## Life Cycle of a WAL

Let's use an example to illustrate the life cycle of a WAL. A RocksDB instance `db` is created with two `column families` "new\_cf" and "default". Once the `db` is opened, a new WAL will be created on disk to persist all writes.

```

1. DB* db;
2. std::vector<ColumnFamilyDescriptor> column_families;
3. column_families.push_back(ColumnFamilyDescriptor(
4.     kDefaultColumnName, ColumnFamilyOptions()));
5. column_families.push_back(ColumnFamilyDescriptor(
6.     "new_cf", ColumnFamilyOptions()));
7. std::vector<ColumnFamilyHandle*> handles;
8. s = DB::Open(DBOptions(), kDBPath, column_families, &handles, &db);

```

Some key-value pairs are added to both column families

```

1. db->Put(WriteOptions(), handles[1], Slice("key1"), Slice("value1"));
2. db->Put(WriteOptions(), handles[0], Slice("key2"), Slice("value2"));
3. db->Put(WriteOptions(), handles[1], Slice("key3"), Slice("value3"));
4. db->Put(WriteOptions(), handles[0], Slice("key4"), Slice("value4"));

```

At this point the WAL should have recorded all writes. The WAL will stay open and keep recording future writes until its size reaches `DBOptions::max_total_wal_size`.

If user decides to flush the column family "new\_cf", several things happen: 1) new\_cf's data (key1 and key3) is flushed to a new SST file 2) a new WAL is created and all future writes to all column families now go to the new WAL 3) the older WAL will not accept new writes but the deletion may be delayed.

```

1. db->Flush(FlushOptions(), handles[1]);
2. // key5 and key6 will appear in a new WAL
3. db->Put(WriteOptions(), handles[1], Slice("key5"), Slice("value5"));
4. db->Put(WriteOptions(), handles[0], Slice("key6"), Slice("value6"));

```

At this point there will be two WALS, the older WAL contains key1 through key4 and

newer WAL contains key5 and key6. Because the older WAL still contains live data for at least one column family (“default”), it cannot be deleted yet. Only when user finally decides to flush “default” column family, the older WAL can be archived and purged from disk automatically.

```
1. db->Flush(FlushOptions(), handles[0]);
2. // The older WAL will be archived and purged separately
```

To summarize, a WAL is created when 1) a new DB is opened, 2) a column family is flushed. A WAL is deleted (or archived if archival is enabled) when all column families have flushed beyond the largest sequence number contained in the WAL, or in other words, all data in the WAL have been persisted to SST files. Archived WALs will be moved to a separate location and purged from disk later on. The actual deletion might be delayed due to replication purposes, see Transaction Log Iterator section below.

## WAL Configurations

The following configuration can be found in [options.h](#)

### DBOptions::wal\_dir

`DBOptions::wal_dir` sets the directory where RocksDB stores write-ahead log files, which allows WALs to be stored in a separate directory from the actual data.

### DBOptions::WAL\_ttl\_seconds, DBOptions::WAL\_size\_limit\_MB

These two fields affect how quickly archived WALs will be deleted. Nonzero values indicate the time and disk space threshold to trigger archived WAL deletion. See [options.h](#) for detailed explanation.

### DBOptions::max\_total\_wal\_size

In order to limit the size of WALs, RocksDB uses `DBOptions::max_total_wal_size` as the trigger of column family flush. Once WALs exceed this size, RocksDB will start forcing the flush of column families to allow deletion of some oldest WALs. This config can be useful when column families are updated at non-uniform frequencies. If there’s no size limit, users may need to keep really old WALs when the infrequently-updated column families hasn’t flushed for a while.

### DBOptions::avoid\_flush\_during\_recovery

This config is self explanatory.

### DBOptions::manual\_wal\_flush

`DBOptions::manual_wal_flush` determines whether WAL flush will be automatic after every

write or purely manual (user must invoke `FlushWAL` to trigger a WAL flush).

## DBOptions::wal\_filter

Through `DBOptions::wal_filter`, users can provide a filter object to be invoked while processing WALs during recovery. *Note: Not supported in ROCKSDB\_LITE mode*

## WriteOptions::disableWAL

`WriteOptions::disableWAL` is useful when users rely on other logging or don't care about data loss.

## WAL Filter

---

## Transaction Log Iterator

---

Transaction log iterator provides a way to replicate the data between RocksDB instances. Once a WAL is archived due to column family flush, the WAL is archived instead of immediately deleted. The goal is to allow transaction log iterator to keep reading the WAL and send to slave for replay.

## Related Pages

---

[WAL Recovery Modes](#)

[Write Ahead Log File Format](#)

[WAL Performance](#)

# Overview

Write ahead log (WAL) serializes memtable operations to persistent medium as log files. In the event of a failure, WAL files can be used to recover the database to its consistent state, by reconstructing the memtable from the logs. When a memtable is flushed out to persistent medium safely, the corresponding WAL log(s) become obsolete and are archived. Eventually the archived logs are purged from disk after a certain period of time.

## WAL Manager

WAL files are generated with increasing sequence number in the WAL directory. In order to reconstruct the state of the database, these files are read in the order of sequence number. WAL manager provides the abstraction for reading the WAL files as a single unit. Internally, it opens and reads the files using Reader or Writer abstraction.

## Reader/Writer

Writer provides an abstraction for appending log records to a log file. The medium specific internal details are handled by WriteableFile interface. Similarly, Reader provides an abstraction for sequentially reading log records from the log file. The internal medium specific details are handled by SequentialFile interface.

## Log File Format

Log file consists of a sequence of variable length records. Records are grouped by `kBlockSize` (32k). If a certain record cannot fit into the leftover space, then the leftover space is padded with empty (null) data. The writer writes and the reader reads in chunks of `kBlockSize`.

```

1.      +-----+-----+-----+-----+-----+-----+ ... -----+
2.  File | r0 |       r1 | P | r2 |       r3 | r4 |       |
3.      +-----+-----+-----+-----+-----+-----+ ... -----+
4.      <--- kBlockSize ----->|<-- kBlockSize ----->|
5.
6. rn = variable size records
7. P = Padding

```

## Record Format

The record layout format is as shown below. There are two kinds of record format, Legacy and Recyclable:

## The Legacy Record Format

```

1. +-----+-----+-----+... ---+
2. |CRC (4B) | Size (2B) | Type (1B) | Payload   |
3. +-----+-----+-----+... ---+
4.
5. CRC = 32bit hash computed over the payload using CRC
6. Size = Length of the payload data
7. Type = Type of record
8.      (kZeroType, kFullType, kFirstType, kLastType, kMiddleType )
9.      The type is used to group a bunch of records together to represent
10.     blocks that are larger than kBlockSize
11.     Payload = Byte stream as long as specified by the payload size

```

## The Recyclable Record Format

```

1. +-----+-----+-----+-----+... ---+
2. |CRC (4B) | Size (2B) | Type (1B) | Log number (4B)| Payload   |
3. +-----+-----+-----+-----+... ---+
4. Same as above, with the addition of
5. Log number = 32bit log file number, so that we can distinguish between
6. records written by the most recent log writer vs a previous one.

```

## Record Format Details For Legacy Format

The log file contents are a sequence of 32KB blocks. The only exception is that the tail of the file may contain a partial block.

Each block consists of a sequence of records:

```

1. block := record* trailer?
2. record :=
3.   checksum: uint32    // crc32c of type and data[]
4.   length: uint16
5.   type: uint8        // One of FULL, FIRST, MIDDLE, LAST
6.   data: uint8[length]

```

A record never starts within the last six bytes of a block (since it won't fit). Any leftover bytes here form the trailer, which must consist entirely of zero bytes and must be skipped by readers.

if exactly seven bytes are left in the current block, and a new non-zero length record is added, the writer must emit a FIRST record (which contains zero bytes of user data) to fill up the trailing seven bytes of the block and then emit all of the user data in subsequent blocks.

More types may be added in the future. Some Readers may skip record types they do not understand, others may report that some data was skipped.

```
1. FULL == 1
```

2. FIRST == 2
3. MIDDLE == 3
4. LAST == 4

The `FULL` record contains the contents of an entire user record.

`FIRST`, `MIDDLE`, `LAST` are types used for user records that have been split into multiple fragments (typically because of block boundaries). `FIRST` is the type of the first fragment of a user record, `LAST` is the type of the last fragment of a user record, and `MID` is the type of all interior fragments of a user record.

Example: consider a sequence of user records:

1. A: length 1000
2. B: length 97270
3. C: length 8000

`A` will be stored as a `FULL` record in the first block.

`B` will be split into three fragments: first fragment occupies the rest of the first block, second fragment occupies the entirety of the second block, and the third fragment occupies a prefix of the third block. This will leave six bytes free in the third block, which will be left empty as the trailer.

`C` will be stored as a `FULL` record in the fourth block.

## Benefits

Some benefits over the `recordio` format:

1. We do not need any heuristics for resyncing - just go to next block boundary and scan. If there is a corruption, skip to the next block. As a side-benefit, we do not get confused when part of the contents of one log file are embedded as a record inside another log file.
2. Splitting at approximate boundaries (e.g., for `mapreduce`) is simple: find the next block boundary and skip records until we hit a `FULL` or `FIRST` record.
3. We do not need extra buffering for large records.

## Downsides

Some downsides compared to `recordio` format:

1. No packing of tiny records. This could be fixed by adding a new record type, so it is a shortcoming of the current implementation, not necessarily the format.
2. No compression. Again, this could be fixed by adding new record types.

# Introduction

---

Every application is unique and requires a certain consistency guarantee from RocksDB. Every committed record in RocksDB is persisted. The uncommitted records are recorded in [write-ahead-log \(WAL\)](#). When RocksDB is shutdown cleanly, all uncommitted data is committed before shutdown and hence consistency is always guaranteed. When RocksDB is killed or the machine is restarted, on restart RocksDB needs to restore itself to a consistent state.

One of the important recovery operations is to replay uncommitted records in WAL. The different WAL recovery modes define the behavior of WAL replay.

## WAL Recovery Modes

---

### kTolerateCorruptedTailRecords

---

In this mode, the WAL replay ignores any error discovered at the tail of the log. The rational is that, on unclean shutdown there can be incomplete writes at the tail of the log. This is a heuristic mode, the system cannot differentiate between corruption at the tail of the log and incomplete write. Any other IO error, will be considered as data corruption.

This mode is acceptable for most application since this provides a reasonable tradeoff between starting RocksDB after an unclean shutdown and consistency.

### kAbsoluteConsistency

---

In this mode, any IO error during WAL replay is considered as data corruption. This mode is ideal for application that cannot afford to loose even a single record and/or have other means of recovering uncommitted data.

### kPointInTimeRecovery

---

In this mode, the WAL replay is stopped after encountering an IO error. The system is recovered to a point-in-time where it is consistent. This is ideal for systems with replicas. Data from another replica can be used to replay past the “point-in-time” where the system is recovered to. (This is the default as of version 6.6.)

### kSkipAnyCorruptedRecords

---

In this mode, any IO error while reading the log is ignored. The system tries to recover as much data as possible. This is ideal for disaster recovery.

## Non-Sync Mode

When `WriteOptions.sync = false` (the default), WAL writes are not synchronized to disk. Unless the operating system thinks it must flush the data (e.g. too many dirty pages), users don't need to wait for any I/O for write.

Users who want to even reduce the CPU of latency introduced by writing to OS page cache, can choose `Options.manual_wal_flush = true`. With this option, WAL writes are not even flushed to the file system page cache, but kept in RocksDB. Users need to call `DB::FlushWAL()` to have buffered entries go to the file system.

Users can call `DB::SyncWAL()` to force fsync WAL files. The function will not block writes being executed in other threads.

In this mode, the WAL write is not crash safe.

## Sync Mode

When `WriteOptions.sync = true`, the WAL file is fsync'ed before returning to the user.

## Group Commit

As most other systems relying on logs, RocksDB supports **group commit** to improve WAL writing throughput, as well as write amplification. RocksDB's group commit is implemented in a naive way: when different threads are writing to the same DB at the same time, all outstanding writes that qualify to be combined will be combined together and write to WAL once, with one fsync. In this way, more writes can be completed by the same number of I/Os.

Writes with different write options might disqualify themselves to be combined. The maximum group size is 1MB. RocksDB won't try to increase batch size by proactive delaying the writes.

## Number of I/Os per write

If `Options.recycle_log_file_num = false` (the default). RocksDB always create new files for new WAL segments. Each WAL write will change both of data and size of the file, so every fsync will generate at least two I/Os, one for data and one for metadata. Note that RocksDB calls `fallocate()` to reserve enough space for the file, but it doesn't prevent the metadata I/O in fsync.

`Options.recycle_log_file_num = true` will keep a pool of WAL files and try to reuse them. When writing to an existing log file, random writes are used from size 0. Before writes hit the end of the file, the file size doesn't change, so the I/O for metadata might be

avoided (also depends on file system mount options). Assuming most WAL files will have similar sizes, I/O needed for metadata will be minimal.

## Write Amplification

---

Note that for some use cases, synchronous WAL can introduce non-trivial write amplification. When writes are small, because complete block/page might need to be updated, we may end up with two 4KB writes (one for data and one for metadata) even if the write is very small. If write is only 40 bytes, 8KB is updated, the write amplification is  $8\text{ KB}/40\text{ bytes} \approx 200$ . It can easily be even larger than the write amplification by LSM-tree.

## Overview

RocksDB is file system and storage medium agnostic. File system operations are not atomic, and are susceptible to inconsistencies in the event of system failure. Even with journaling turned on, file systems do not guarantee consistency on unclean restart. POSIX file system does not support atomic batching of operations either. Hence, it is not possible to rely on metadata embedded in RocksDB datastore files to reconstruct the last consistent state of the RocksDB on restart.

RocksDB has a built-in mechanism to overcome these limitations of POSIX file system by keeping a transactional log of RocksDB state changes called the MANIFEST. MANIFEST is used to restore RocksDB to the latest known consistent state on a restart.

## Terminology

- *MANIFEST* refers to the system that keeps track of RocksDB state changes in a transactional log
- *Manifest log* refers to an individual log file that contains RocksDB state snapshot/edits
- *CURRENT* refers to the latest manifest log

## How does it work ?

MANIFEST is a transactional log of the RocksDB state changes. MANIFEST consists of - manifest log files and latest manifest file pointer. Manifest logs are rolling log files named MANIFEST-(seq number). The sequence number is always increasing. CURRENT is a special file that identifies the latest manifest log file.

On system (re)start, the latest manifest log contains the consistent state of RocksDB. Any subsequent change to RocksDB state is logged to the manifest log file. When a manifest log file exceeds a certain size, a new manifest log file is created with the snapshot of the RocksDB state. The latest manifest file pointer is updated and the file system is synced. Upon successful update to CURRENT file, the redundant manifest logs are purged.

1. MANIFEST = { CURRENT, MANIFEST-<seq-no>\* }
2. CURRENT = *File* pointer to the latest manifest log
3. MANIFEST-<seq no> = Contains snapshot of *RocksDB* state and subsequent modifications

## Version Edit

A certain state of RocksDB at any given time is referred to as a version (aka snapshot). Any modification to the version is considered a version edit. A version (or

RocksDB state snapshot) is constructed by joining a sequence of version-edits. Essentially, a manifest log file is a sequence of version edits.

```

1. version-edit      = Any RocksDB state change
2. version          = { version-edit* }
3. manifest-log-file = { version, version-edit* }
4.                  = { version-edit* }

```

## Version Edit Layout

Manifest log is a sequence of version edit records. The version edit record type is identified by the edit identification number.

We use the following datatypes for encoding/decoding.

## Data Types

Simple data types

```

1. VarX - Variable character encoding of intX
2. FixedX - Fixed character encoding of intX

```

Complex data types

```

1. String - Length prefixed string data
2. +-----+-----+
3. | size (n) | content of string |
4. +-----+-----+
5. |<- Var32 -->|<- n           -->|

```

## Version Edit Record Format

Version edit records have the following format. The decoder identifies the record type using the record identification number.

```

1. +-----+----- ....., -----+
2. | Record ID   | Variable size record data |
3. +-----+----- ....., -----+
4. <- Var32 --->|<- varies by type       -->

```

## Version Edit Record Types and Layout

There are a variety of edit records corresponding to different state changes of RocksDB.

## Comparator edit record:

```

1. Captures the comparator name
2.
3. +-----+-----+
4. | kComparator | data      |
5. +-----+-----+
6. <-- Var32 --->|<-- String -->|

```

## Log number edit record:

```

1. Latest WAL log file number
2.
3. +-----+-----+
4. | kLogNumber | log number   |
5. +-----+-----+
6. <-- Var32 --->|<-- Var64 -->|

```

## Previous File Number edit record:

```

1. Previous manifest file number
2.
3. +-----+-----+
4. | kPrevFileName | log number   |
5. +-----+-----+
6. <-- Var32     --->|<-- Var64 -->|

```

## Next File Number edit record:

```

1. Next manifest file number
2.
3. +-----+-----+
4. | kNextFileName | log number   |
5. +-----+-----+
6. <-- Var32     --->|<-- Var64 -->|

```

## Last Sequence Number edit record:

```

1. Last sequence number of RocksDB
2.
3. +-----+-----+
4. | kLastSequence | log number   |
5. +-----+-----+
6. <-- Var32     --->|<-- Var64 -->|

```

## Max Column Family edit record:

```

1. Adjust the maximum number of family columns allowed.
2.

```

```

3. +-----+-----+
4. | kMaxColumnFamily | log number |
5. +-----+-----+
6. <-> Var32      --->|<-> Var32      --->|

```

## Deleted File edit record:

```

1. Mark a file as deleted from database.
2.
3. +-----+-----+
4. | kDeletedFile | level       | file number |
5. +-----+-----+
6. <-> Var32      --->|<-> Var32 --->|<-> Var64 --->|

```

## New File edit record:

Mark a file as newly added to the database and provide RocksDB meta information.

- File edit record with compaction information

```

+-----+-----+-----+-----+-----+-----+-----+-----+
1. -----+
   | kNewFile4 | level       | file number | file size | smallest_key | largest_key | smallest_seqno |
2. largest_seq_no |
+-----+-----+-----+-----+-----+-----+-----+-----+
3. -----+
   | <-> var32 --->|<-> var32 --->|<-> var64 --->|<-> var64 ->|<-> String --->|<-> String --->|<-> var64 --->|<-> var64 --->|<->
4. var64 --->|
5.
+-----+-----+-----+-----+-----+-----+-----+-----+
6. --+
   | CustomTag1 | Field 1 size n1 | field1 | ... | CustomTag(m) | Field m size n(m) | field(m) |
7. kTerminate |
+-----+-----+-----+-----+-----+-----+-----+-----+
8. --+
   <-> var32 --->|<-> var32 --->|<-> n1 ->|       |<-> var32 --->|<-> var32 --->|<-> n(m) ->|<-> var32 -
9. ->|

```

Several Optional customized fields can be written there. The field has a special bit indicating that whether it can be safely ignored. This is for compatibility reason. A RocksDB older release may see a field it can't identify. Checking the bit, RocksDB knows whether it should stop opening the DB, or ignore the field.

Several optional customized fields are supported: `kNeedCompaction` : Whether the file should be compacted to the next level. `kMinLogNumberToKeepHack` : WAL file number that is still in need for recovery after this entry. `kPathID` : The Path ID in which the file lives. This can't be ignored by an old release.

- File edit record backward compatible

```

+-----+-----+-----+-----+-----+-----+-----+-----+
1. -----+

```

```

| kNewFile2      | level       | file number | file size   | smallest_key    | largest_key     | smallest_seqno |
2. largest_seq_no |
+-----+-----+-----+-----+-----+-----+
3. -----
<-> var32 -->|<-> var32 -->|<-> var64 -->|-> var64 ->|<-> String -->|<-> String -->|<-> var64 -->|<->
4. var64 -->|

```

- File edit record with path information

```

1. +-----+-----+-----+-----+-----+-----+
2. | kNewFile3      | level       | file number | Path ID      | file size   | smallest_key    | largest_key     |
3. +-----+-----+-----+-----+-----+-----+
4. |<-> var32 -->|<-> var32 -->|<-> var64 -->|<-> var32 -->|<-> var64 -->|<-> String -->|<-> String -->|
5. +-----+-----+
6. | smallest_seqno | largest_seq_no |
7. +-----+-----+
8. <-> var64 -->|<-> var64 -->|

```

### Column family status edit record:

```

1. Note the status of column family feature (enabled/disabled)
2.
3. +-----+
4. | kColumnFamily      | 0/1           |
5. +-----+
6. <-> Var32 -->|<-> Var32 -->|

```

### Column family add edit record:

```

1. Add a column family
2.
3. +-----+
4. | kColumnFamilyAdd    | cf name      |
5. +-----+
6. <-> Var32 -->|<-> String -->|

```

### Column family drop edit record:

```

1. Drop all column family
2.
3. +-----+
4. | kColumnFamilyDrop   |           |
5. +-----+
6. <-> Var32 -->|

```

### Record as part of an atomic group (since RocksDB 5.16):

There are cases in which ‘all-or-nothing’, multi-column-family version change is desirable. For example, [atomic flush](#) ensures either all or none of the column families get flushed successfully, [multiple column families external SST ingestion](#) guarantees

that either all or none of the column families ingest SSTs successfully. Since writing multiple version edits is not atomic, we need to take extra measure to achieve atomicity (not necessarily *instantaneity* from the user's perspective). Therefore we introduce a new record field `kInAtomicGroup` to indicate that this record is part of a group of version edits that follow the 'all-or-none' property. The format is as follows.

```

1. +-----+-----+
2. | kInAtomicGroup | #remaining version edits in the same group |
3. +-----+-----+
4. |<-- Var32 -->|<-- Var32 -->|

```

During recovery, RocksDB buffers version edits of an atomic group without applying them until the last version edit of the atomic group is decoded successfully from the MANIFEST file. Then RocksDB applies all the version edits in this atomic group. RocksDB never applies partial atomic groups.

## Version Edit ignorable record types

We reserved a special bit in record type. If the bit is set, it can be safely ignored. And the safely ignorable record has a standard general format:

```

1. +-----+-----+
2. | kTag | field length n | fields ... |
3. +-----+-----+
4. <- Var32->|<- var32 -->|<- n >|

```

This is introduced in 6.0 and no customized ignoreable record created yet.

The following types of version edits fall into the ignorable category.

DB ID edit record: introduced since RocksDB 6.5. If `options.write_dbid_to_manifest` is true, then RocksDB writes the DB ID edit record to the MANIFEST file, besides storing in the IDENTITY file.

```

1. +-----+
2. | kDbId | db id |
3. +-----+
4. |<- Var32 ->|<- String ->|

```

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` 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, with each shard has no less than 512k bytes of capacity.

## Usage

Out of box, RocksDB will use LRU-based block cache implementation with 8MB capacity. To set a customized block cache, call `NewLRUCache()` or `NewClockCache()` to create a cache object, and set it to block based table options. Users can also have their own cache implementation by implementing the `Cache` interface.

```
1. std::shared_ptr<Cache> cache = NewLRUCache(capacity);
2. BlockBasedTableOptions table_options;
3. table_options.block_cache = cache;
4. Options options;
5. options.table_factory.reset(NewBlockBasedTableFactory(table_options));
```

To set compressed block cache:

```
1. table_options.block_cache_compressed = another_cache;
```

RocksDB will create the default block cache if `block_cache` is set to `nullptr`. To disable block cache completely:

```
1. table_options.no_block_cache = true;
```

## LRU Cache

Out of box, RocksDB will use LRU-based block cache implementation with 8MB capacity. Each shard of the cache maintains its own LRU list and its own hash table for lookup. Synchronization is done via a per-shard mutex. Both lookup and insert to the cache would require a locking mutex of the shard. User can create a LRU cache by calling `NewLRUCache()`. The function provides several useful options to set to the cache:

- `capacity` : Total size of the cache.
- `num_shard_bits` : The number of bits from cache keys to be use as shard id. The cache

- will be sharded into `2^num_shard_bits` shards.
- `strict_capacity_limit` : In rare case, block cache size can go larger than its capacity. This is when ongoing reads or iterations over DB pin blocks in block cache, and the total size of pinned blocks exceeds the capacity. If there are further reads which try to insert blocks into block cache, if `strict_capacity_limit=false` (default), the cache will fail to respect its capacity limit and allow the insertion. This can create undesired OOM error that crashes the DB if the host don't have enough memory. Setting the option to `true` will reject further insertion to the cache and fail the read or iteration. The option works on per-shard basis, means it is possible one shard is rejecting insert when it is full, while another shard still have extra unpinned space.
  - `high_pri_pool_ratio` : The ratio of capacity reserved for high priority blocks. See [Caching Index, Filter, and Compression Dictionary Blocks](#) section below for more information.

## Clock Cache

`ClockCache` implements the [CLOCK algorithm](#). Each shard of clock cache maintains a circular list of cache entries. A clock handle runs over the circular list looking for unpinned entries to evict, but also giving each entry a second chance to stay in cache if it has been used since last scan. A `tbb::concurrent_hash_map` is used for lookup.

The benefit over `LRUCache` is it has finer-granularity locking. In case of LRU cache, the per-shard mutex has to be locked even on lookup, since it needs to update its LRU-list. Looking up from a clock cache won't require locking per-shard mutex, but only looking up the concurrent hash map, which has fine-granularity locking. Only inserts needs to lock the per-shard mutex. With clock cache we see boost of read throughput over LRU cache in contented environment (see inline comments in `cache/clock_cache.cc` for benchmark setup):

1.	Threads Cache		Cache		ClockCache		LRUCache	
	Size	Index/Filter	Throughput(MB/s)	Hit	Throughput(MB/s)	Hit		
3.	32	2GB	yes	466.7	85.9%	433.7	86.5%	
4.	32	2GB	no	529.9	72.7%	532.7	73.9%	
5.	32	64GB	yes	649.9	99.9%	507.9	99.9%	
6.	32	64GB	no	740.4	99.9%	662.8	99.9%	
7.	16	2GB	yes	278.4	85.9%	283.4	86.5%	
8.	16	2GB	no	318.6	72.7%	335.8	73.9%	
9.	16	64GB	yes	391.9	99.9%	353.3	99.9%	
10.	16	64GB	no	433.8	99.8%	419.4	99.8%	

To create a clock cache, call `NewClockCache()`. To make clock cache available, RocksDB needs to be linked with [Intel TBB](#) library. Again there are several options users can set when creating a clock cache:

- `capacity` : Same as LRUCache.
- `num_shard_bits` : Same as LRUCache.
- `strict_capacity_limit` : Same as LRUCache.

# Caching Index, Filter, and Compression Dictionary Blocks

By default index, filter, and compression dictionary blocks (with the exception of the partitions of `partitioned indexes/filters`) are cached outside of block cache, and users won't be able to control how much memory should be used to cache these blocks, other than setting `max_open_files`. Users can opt to cache index and filter blocks in block cache, which allows for better control of memory used by RocksDB. To cache index, filter, and compression dictionary blocks in block cache:

```
1. BlockBasedTableOptions table_options;
2. table_options.cache_index_and_filter_blocks = true;
```

Note that the partitions of `partitioned indexes/filters` are as a rule stored in the block cache, regardless of the value of the above option.

By putting index, filter, and compression dictionary blocks in block cache, these blocks have to compete against data blocks for staying in cache. Although index and filter blocks are being accessed more frequently than data blocks, there are scenarios where these blocks can be thrashing. This is undesired because index and filter blocks tend to be much larger than data blocks, and they are usually of higher value to stay in cache (the latter is also true for compression dictionary blocks). There are two options to tune to mitigate the problem:

- `cache_index_and_filter_blocks_with_high_priority` : Set priority to high for index, filter, and compression dictionary blocks in block cache. For `partitioned indexes/filters`, this affects the priority of the partitions as well. It only affect `LRUCache` so far, and need to use together with `high_pri_pool_ratio` when calling `NewLRUCache()`. If the feature is enabled, LRU-list in LRU cache will be split into two parts, one for high-pri blocks and one for low-pri blocks. Data blocks will be inserted to the head of low-pri pool. Index, filter, and compression dictionary blocks will be inserted to the head of high-pri pool. If the total usage in the high-pri pool exceed `capacity * high_pri_pool_ratio`, the block at the tail of high-pri pool will overflow to the head of low-pri pool, after which it will compete against data blocks to stay in cache. Eviction will start from the tail of low-pri pool.
- `pin_l0_filter_and_index_blocks_in_cache` : Pin level-0 file's index and filter blocks in block cache, to avoid them from being evicted. Starting with RocksDB version 6.4, this option also affects compression dictionary blocks. Level-0 index and filters are typically accessed more frequently. Also they tend to be smaller in size so hopefully pinning them in cache won't consume too much capacity.
- `pin_top_level_index_and_filter` : only applicable to `partitioned indexes/filters`. If `true`, the top level of the `partitioned index/filter` structure will be pinned in the cache, regardless of the LSM tree level (that is, unlike the previous option, this affects files on all LSM tree levels, not just L0).

## Simulated Cache

`SimCache` is an utility to predict cache hit rate if cache capacity or number of shards is changed. It wraps around the real `Cache` object that the DB is using, and runs a shadow LRU cache simulating the given capacity and number of shards, and measure cache hits and misses of the shadow cache. The utility is useful when user wants to open a DB with, say, 4GB cache size, but would like to know what the cache hit rate will become if cache size enlarge to, say, 64GB. To create a simulated cache:

```

1. // This cache is the actual cache use by the DB.
2. std::shared_ptr<Cache> cache = NewLRUCache(capacity);
3. // This is the simulated cache.
4. std::shared_ptr<Cache> sim_cache = NewSimCache(cache, sim_capacity, sim_num_shard_bits);
5. BlockBasedTableOptions table_options;
6. table_options.block_cache = sim_cache;

```

The extra memory overhead of the simulated cache is less than 2% of `sim_capacity`.

## Statistics

A list of block cache counters can be accessed through `Options.statistics` if it is non-null.

```

1. // total block cache misses
2. // REQUIRES: BLOCK_CACHE_MISS == BLOCK_CACHE_INDEX_MISS +
3. //           BLOCK_CACHE_FILTER_MISS +
4. //           BLOCK_CACHE_DATA_MISS;
5. BLOCK_CACHE_MISS = 0,
6. // total block cache hit
7. // REQUIRES: BLOCK_CACHE_HIT == BLOCK_CACHE_INDEX_HIT +
8. //           BLOCK_CACHE_FILTER_HIT +
9. //           BLOCK_CACHE_DATA_HIT;
10. BLOCK_CACHE_HIT,
11. // # of blocks added to block cache.
12. BLOCK_CACHE_ADD,
13. // # of failures when adding blocks to block cache.
14. BLOCK_CACHE_ADD_FAILURES,
15. // # of times cache miss when accessing index block from block cache.
16. BLOCK_CACHE_INDEX_MISS,
17. // # of times cache hit when accessing index block from block cache.
18. BLOCK_CACHE_INDEX_HIT,
19. // # of times cache miss when accessing filter block from block cache.
20. BLOCK_CACHE_FILTER_MISS,
21. // # of times cache hit when accessing filter block from block cache.
22. BLOCK_CACHE_FILTER_HIT,
23. // # of times cache miss when accessing data block from block cache.
24. BLOCK_CACHE_DATA_MISS,
25. // # of times cache hit when accessing data block from block cache.
26. BLOCK_CACHE_DATA_HIT,
27. // # of bytes read from cache.

```

```
28. BLOCK_CACHE_BYTES_READ,  
29. // # of bytes written into cache.  
30. BLOCK_CACHE_BYTES_WRITE,
```

See also: [Memory-usage-in-RocksDB#block-cache](#)

Write buffer manager helps users control the total memory used by memtables across multiple column families and/or DB instances. Users can enable this control by 2 ways:

1. Limit the total memtable usage across multiple column families and DBs under a threshold.
2. Cost the memtable memory usage to block cache so that memory of RocksDB can be capped by the single limit.

The usage of a write buffer manager is similar to `rate_limiter` and `sst_file_manager`. Users can create one write buffer manager object and pass it to all the options of column families or DBs whose memtable size they want to be controlled by this object.

For more details refer, [write\\_buffer\\_manager option](#) and [write\\_buffer\\_manager.h](#)

## Limit total memory of memtables

---

A memory limit is given when creating the write buffer manager object. RocksDB will try to limit the total memory to under this limit.

In version 5.6 or higher, a flush will be triggered on one column family of the DB you are inserting to,

1. If mutable memtable size exceeds about 90% of the limit,
2. If the total memory is over the limit, more aggressive flush may also be triggered only if the mutable memtable size also exceeds 50% of the limit. Both checks are needed because if already more than half memory is being flushed, triggering more flush may not help.

Before version 5.6, a flush will be triggered if the total mutable memtable size exceeds the limit.

In version 5.6 or higher, the total memory is counted as total memory allocated in the arena, even if some of that may not yet be used by memtable. In earlier versions, the memory is counted as memory actually used by memtables.

## Cost memory used in memtable to block cache

---

Since version 5.6, users can set up RocksDB to cost memory used by memtables to block cache. This can happen no matter whether you enable memtable memory limit or not. This option is added to manage memory (memtables + block cache) under a single limit.

In most cases, blocks that are actually used in block cache are just a smaller percentage than data cached in block cache, so when users enable this feature, the block cache capacity will cover the memory usage for both block cache and memtables. If users also enable `cache_index_and_filter_blocks`, then the three major types of memory of RocksDB (`cache_index_and_filter_blocks`, memtables and data blocks cached) will be capped by the single cap.

## Implementation:

For every, let's say 1MB memory allocated memtable, WriteBufferManager will put dummy 1MB entries (empty) to the block cache so that the block cache can track the size correctly for memtables and evict blocks to make room if needed. In case the memory used by the memtable shrinks, WriteBufferManager will not immediately remove the dummy blocks but gradually shrink memory cost in the block cache if the total memory used by memtables is less than 3/4 of what we reserve in the block cache. We do this because we don't want to free the memory costed in the block cache immediately when a memtable is freed, as block cache insertion is expensive so we make sure we shrink the memory cost in block cache over time.

To enable this feature,

- pass the block cache you are using to the WriteBufferManager you are going to use.
- still pass the parameter of WriteBufferManager as the maximum memory you want RocksDB to use for memtables.
- set the capacity of your block cache to be the sum of the memory used for cached data blocks and memtables.

```
WriteBufferManager(size_t buffer_size, std::shared_ptr<Cache> cache = {})
```

Compaction algorithms constrain the LSM tree shape. They determine which sorted runs can be merged by it and which sorted runs need to be accessed for a read operation. You can read more on RocksDB Compactions here: [Multi-threaded compactions](#)

## LSM terminology and metaphors

Let us first establish the different, sometimes mixed, metaphors and terminology used in describing LSM levels and structure.

- L0 can be called the **lowest**, **top**, or **first** level, or level with the **newest** data. (Confusing because “low” and “top” are typically opposite, and “first” and “newest” are typically opposite.)
- The highest-numbered level with data, which might not be Lmax, is **bottom-most**, with the **oldest** data.
- Lmax is the **highest** or **last** level.

## Overview of Compaction algorithms

Source: <https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html>

Here we present a taxonomy of compaction algorithms: Classic Leveled, Tiered, Tiered+Leveled, Leveled-N, FIFO. Out of them, Rocksdb implements Tiered+Leveled, termed Level Compaction in the code, Tiered termed Universal in the code, and FIFO.

### Classic Leveled

Classic Leveled compaction, introduced by LSM-tree paper by O’Neil et al, minimizes space amplification at the cost of read and write amplification.

The LSM tree is a sequence of levels. Each level is one sorted run that can be range partitioned into many files. Each level is many times larger than the previous level. The size ratio of adjacent levels is sometimes called the fanout and write amplification is minimized when the same fanout is used between all levels. Compaction into level N (Ln) merges data from Ln-1 into Ln. Compaction into Ln rewrites data that was previously merged into Ln. The per-level write amplification is equal to the fanout in the worst case, but it tends to be less than the fanout in practice as explained in this paper by Hyeontaek Lim et al. Compaction in the original LSM paper was all-to-all – all data from Ln-1 is merged with all data from Ln. It is some-to-some for LevelDB and RocksDB – some data from Ln-1 is merged with some (the overlapping) data in Ln.

While write amplification is usually worse with leveled than with tiered, there are a few cases where leveled is competitive. The first is key-order inserts and a RocksDB optimization greatly reduces write-amp in that case. The second one is skewed writes where only a small fraction of the keys are likely to be updated. With the right value for compaction priority in RocksDB compaction should stop at the smallest level that

is large enough to capture the write working set – it won't go all the way to the max level. When leveled compaction is some-to-some then compaction is only done for the slices of the LSM tree that overlap the written keys, which can generate less write amplification than all-to-all compaction.

## Leveled-N

Leveled-N compaction is like leveled compaction but with less write and more read amplification. It allows more than one sorted run per level. Compaction merges all sorted runs from  $L_{n-1}$  into one sorted run from  $L_n$ , which is leveled. And then "-N" is added to the name to indicate there can be n sorted runs per level. The Dostoevsky paper defined a compaction algorithm named Fluid LSM in which the max level has 1 sorted run but the non-max levels can have more than 1 sorted run. Leveled compaction is done into the max level.

## Tiered

Tiered compaction minimizes write amplification at the cost of read and space amplification.

The LSM tree can still be viewed as a sequence of levels as explained in the Dostoevsky paper by Niv Dayan and Stratos Idreos. Each level has N sorted runs. Each sorted run in  $L_n$  is  $\sim N$  times larger than a sorted run in  $L_{n-1}$ . Compaction merges all sorted runs in one level to create a new sorted run in the next level. N in this case is similar to fanout for leveled compaction. Compaction does not read/rewrite sorted runs in  $L_n$  when merging into  $L_{n+1}$ . The per-level write amplification is 1 which is much less than for leveled where it was fanout.

A common approach for tiered is to merge sorted runs of similar size, without having the notion of levels (which imply a target for the number of sorted runs of specific sizes). Most include some notion of major compaction that includes the largest sorted run and conditions that trigger major and non-major compaction. Too many files and too many bytes are typical conditions.

There are a few challenges with tiered compaction:

- Transient space amplification is large when compaction includes a sorted run from the max level.
- The block index and bloom filter for large sorted runs will be large. Splitting them into smaller parts is a good idea.
- Compaction for large sorted runs takes a long time. Multi-threading would help.
- Compaction is all-to-all. When there is skew and most of the keys don't get updates, large sorted runs might get rewritten because compaction is all-to-all. In a traditional tiered algorithm there is no way to rewrite a subset of a large sorted run.

For tiered compaction the notion of levels are usually a concept to reason about the

shape of the LSM tree and estimate write amplification. With RocksDB they are also an implementation detail. The levels of the LSM tree beyond L0 can be used to store the larger sorted runs. The benefit from this is to partition large sorted runs into smaller SSTs. This reduces the size of the largest bloom filter and block index chunks – which is friendlier to the block cache – and was a big deal before partitioned index/filter was supported. With subcompactions this enables multi-threaded compaction of the largest sorted runs. Note that RocksDB used the name universal rather than tiered.

Tiered compaction in RocksDB code base is termed Universal Compaction.

## Tiered+Leveled

Tiered+Leveled has less write amplification than leveled and less space amplification than tiered.

The tiered+leveled approach is a hybrid that uses tiered for the smaller levels and leveled for the larger levels. It is flexible about the level at which the LSM tree switches from tiered to leveled. For now I assume that if Ln is leveled then all levels that follow (Ln+1, Ln+2, ...) must be leveled.

SlimDB from VLDB 2018 is an example of tiered+leveled although it might allow Lk to be tiered when Ln is leveled for k > n. Fluid LSM is described as tiered+leveled but I think it is leveled-N.

Leveled compaction in RocksDB is also tiered+leveled. There can be N sorted runs at the memtable level courtesy of the max\_write\_buffer\_number option – only one is active for writes, the rest are read-only waiting to be flushed. A memtable flush is similar to tiered compaction – the memtable output creates a new sorted run in L0 and doesn't read/rewrite existing sorted runs in L0. There can be N sorted runs in level 0 (L0) courtesy of level0\_file\_num\_compaction\_trigger. So the L0 is tiered. Compaction isn't done into the memtable level so it doesn't have to be labeled as tiered or leveled. Subcompactions in the RocksDB L0 makes this even more interesting, but that is a topic for another post.

## FIFO

The FIFOStyle Compaction drops oldest file when obsolete and can be used for cache-like data.

## Options

---

Here we give overview of the options that impact behavior of Compactions:

- `Options::compaction_style` - RocksDB currently supports two compaction algorithms - Universal style and Level style. This option switches between the two. Can be `kCompactionStyleUniversal` or `kCompactionStyleLevel`. If this is

- `kCompactionStyleUniversal`, then you can configure universal style parameters with `Options::compaction_options_universal`.
- `Options::disable_auto_compactions` - Disable automatic compactions. Manual compactions can still be issued on this database.
  - `Options::compaction_filter` - Allows an application to modify/delete a key-value during background compaction. The client must provide `compaction_filter_factory` if it requires a new compaction filter to be used for different compaction processes. Client should specify only one of filter or factory.
  - `Options::compaction_filter_factory` - a factory that provides compaction filter objects which allow an application to modify/delete a key-value during background compaction.

Other options impacting performance of compactions and when they get triggered are:

- `Options::access_hint_on_compaction_start` - Specify the file access pattern once a compaction is started. It will be applied to all input files of a compaction. Default: NORMAL
- `Options::level0_file_num_compaction_trigger` - Number of files to trigger level-0 compaction. A negative value means that level-0 compaction will not be triggered by number of files at all.
- `Options::target_file_size_base` and `Options::target_file_size_multiplier` - Target file size for compaction. `target_file_size_base` is per-file size for level-1. Target file size for level L can be calculated by `target_file_size_base * (target_file_size_multiplier ^ (L-1))` For example, if `target_file_size_base` is 2MB and `target_file_size_multiplier` is 10, then each file on level-1 will be 2MB, and each file on level 2 will be 20MB, and each file on level-3 will be 200MB. Default `target_file_size_base` is 64MB and default `target_file_size_multiplier` is 1.
- `Options::max_compaction_bytes` - Maximum number of bytes in all compacted files. We avoid expanding the lower level file set of a compaction if it would make the total compaction cover more than this amount.
- `Options::max_background_compactions` - Maximum number of concurrent background jobs, submitted to the default LOW priority thread pool
- `Options::compaction_readahead_size` - If non-zero, we perform bigger reads when doing compaction. If you're running RocksDB on spinning disks, you should set this to at least 2MB. We enforce it to be 2MB if you don't set it with direct I/O.

Compaction can also be manually triggered. See [Manual Compaction](#)

You can learn more about all of those options in [rocksdb/options.h](#)

## Leveled style compaction

---

See [Leveled Compaction](#).

## Universal style compaction

---

For description about universal style compaction, see [Universal compaction style](#)

If you're using Universal style compaction, there is an object `CompactionOptionsUniversal` that holds all the different options for that compaction. The exact definition is in `rocksdb/universal_compaction.h` and you can set it in `Options::compaction_options_universal`. Here we give a short overview of options in `CompactionOptionsUniversal` :

- `CompactionOptionsUniversal::size_ratio` - Percentage flexibility while comparing file size. If the candidate file(s) size is 1% smaller than the next file's size, then include next file into this candidate set. Default: 1
- `CompactionOptionsUniversal::min_merge_width` - The minimum number of files in a single compaction run. Default: 2
- `CompactionOptionsUniversal::max_merge_width` - The maximum number of files in a single compaction run. Default: `UINT_MAX`
- `CompactionOptionsUniversal::max_size_amplification_percent` - The size amplification is defined as the amount (in percentage) of additional storage needed to store a single byte of data in the database. For example, a size amplification of 2% means that a database that contains 100 bytes of user-data may occupy upto 102 bytes of physical storage. By this definition, a fully compacted database has a size amplification of 0%. Rocksdb uses the following heuristic to calculate size amplification: it assumes that all files excluding the earliest file contribute to the size amplification. Default: 200, which means that a 100 byte database could require upto 300 bytes of storage.
- `CompactionOptionsUniversal::compression_size_percent` - If this option is set to be -1 (the default value), all the output files will follow compression type specified. If this option is not negative, we will try to make sure compressed size is just above this value. In normal cases, at least this percentage of data will be compressed. When we are compacting to a new file, here is the criteria whether it needs to be compressed: assuming here are the list of files sorted by generation time: [ A1...An B1...Bm C1...Ct ], where A1 is the newest and Ct is the oldest, and we are going to compact B1...Bm, we calculate the total size of all the files as `total_size`, as well as the total size of C1...Ct as `total_C`, the compaction output file will be compressed iff `total_C / total_size < this percentage`
- `CompactionOptionsUniversal::stop_style` - The algorithm used to stop picking files into a single compaction run. Can be `kCompactionStopStyleSimilarSize` (pick files of similar size) or `kCompactionStopStyleTotalSize` (total size of picked files > next file).

Default: `kCompactionStopStyleTotalSize`

## FIFO Compaction Style

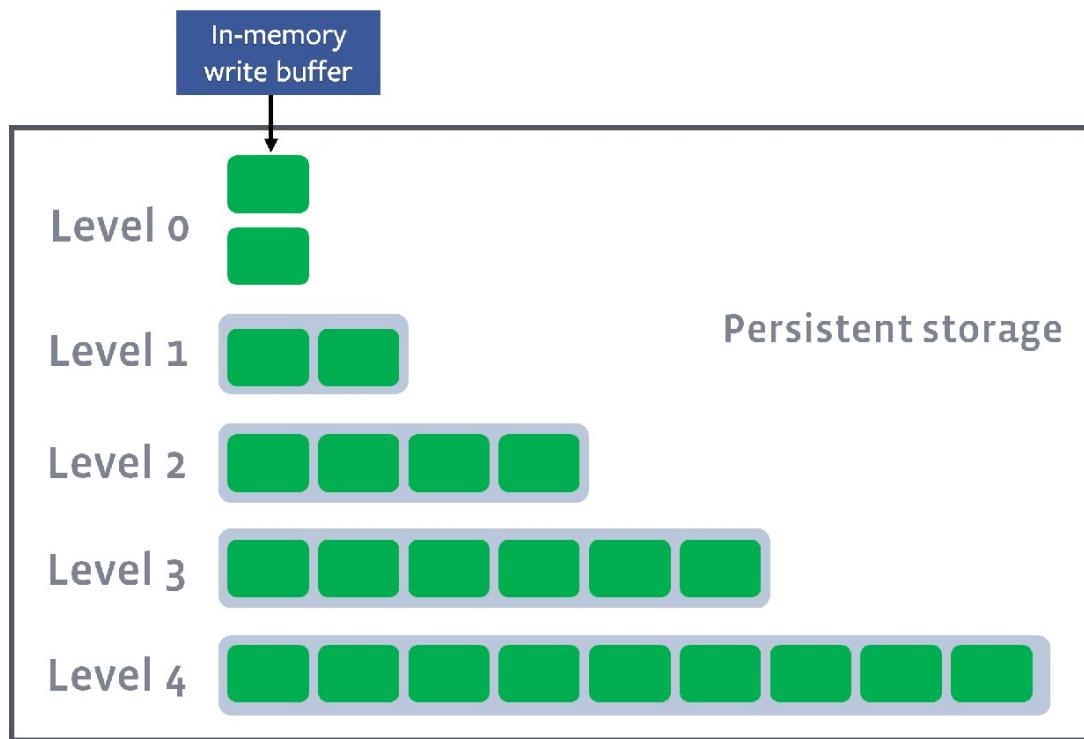
See [FIFO compaction style](#)

## Thread pools

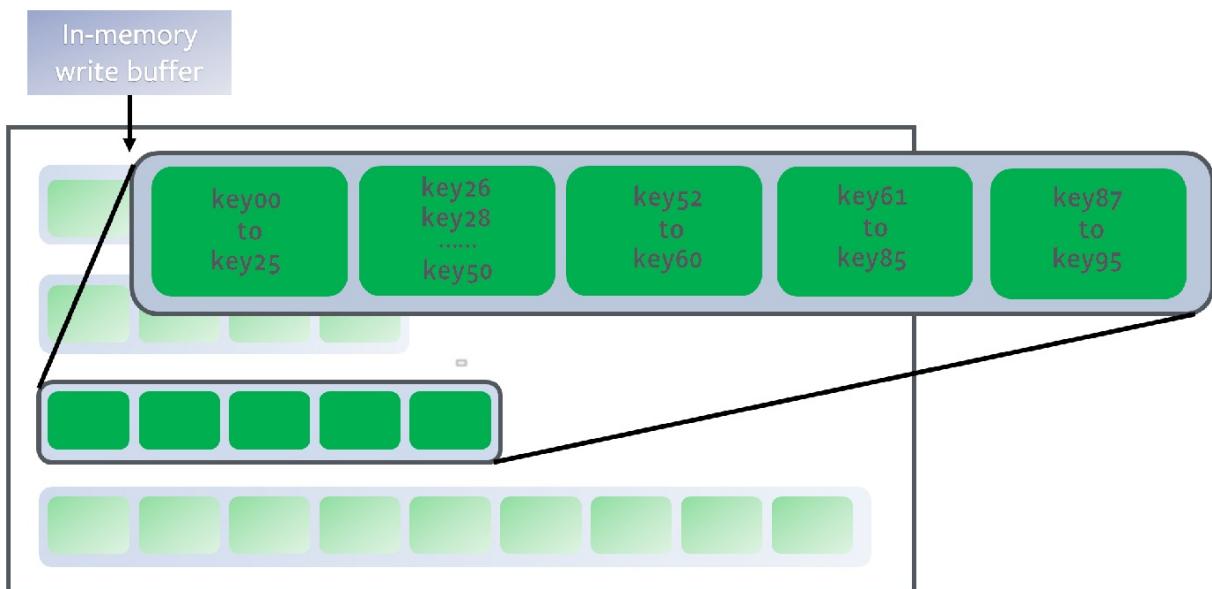
Compactions are executed in thread pools. See [Thread Pool](#).

# Structure of the files

Files on disk are organized in multiple levels. We call them level-1, level-2, etc, or L1, L2, etc, for short. A special level-0 (or L0 for short) contains files just flushed from in-memory write buffer (memtable). Each level (except level 0) is one data sorted run:



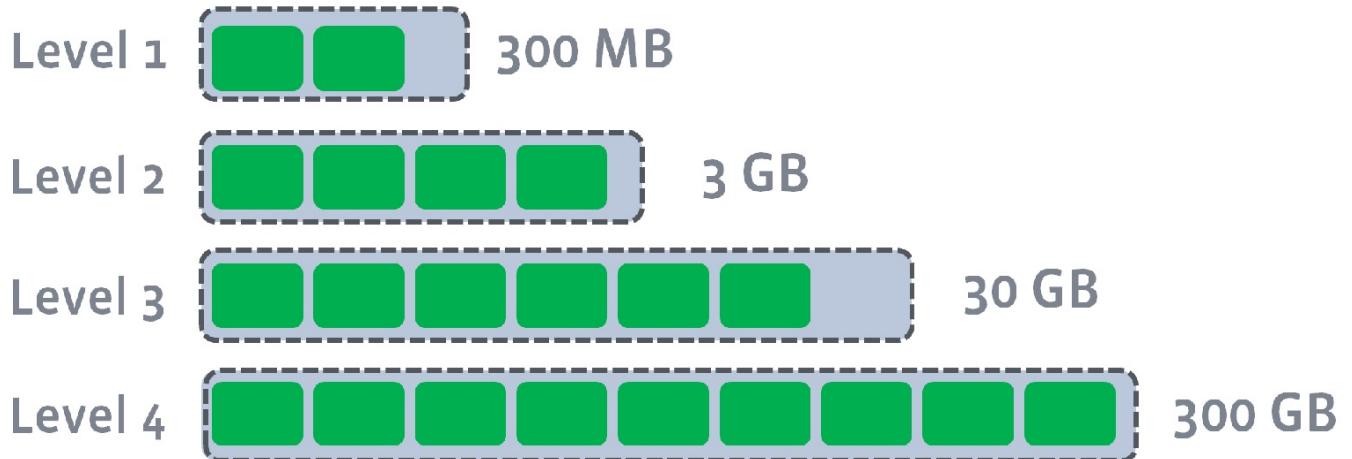
Inside each level (except level 0), data is range partitioned into multiple SST files:



The level is a sorted run because keys in each SST file are sorted (See [Block-based](#)

[Table Format](#) as an example). To identify a position for a key, we first binary search the start/end key of all files to identify which file possibly contains the key, and then binary search inside the file to locate the exact position. In all, it is a full binary search across all the keys in the level.

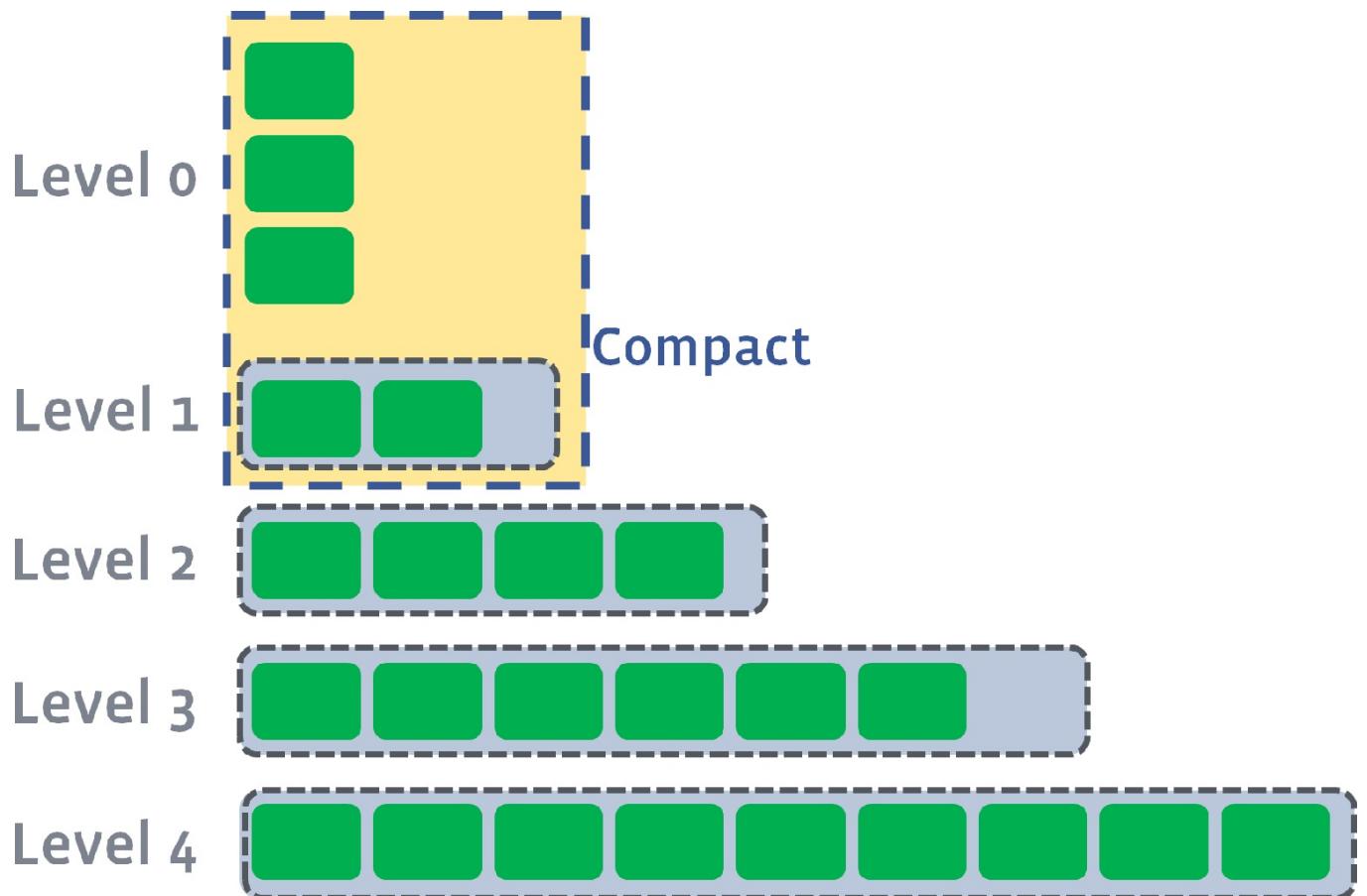
All non-0 levels have target sizes. Compaction's goal will be to restrict data size of those levels to be under the target. The size targets are usually exponentially increasing:



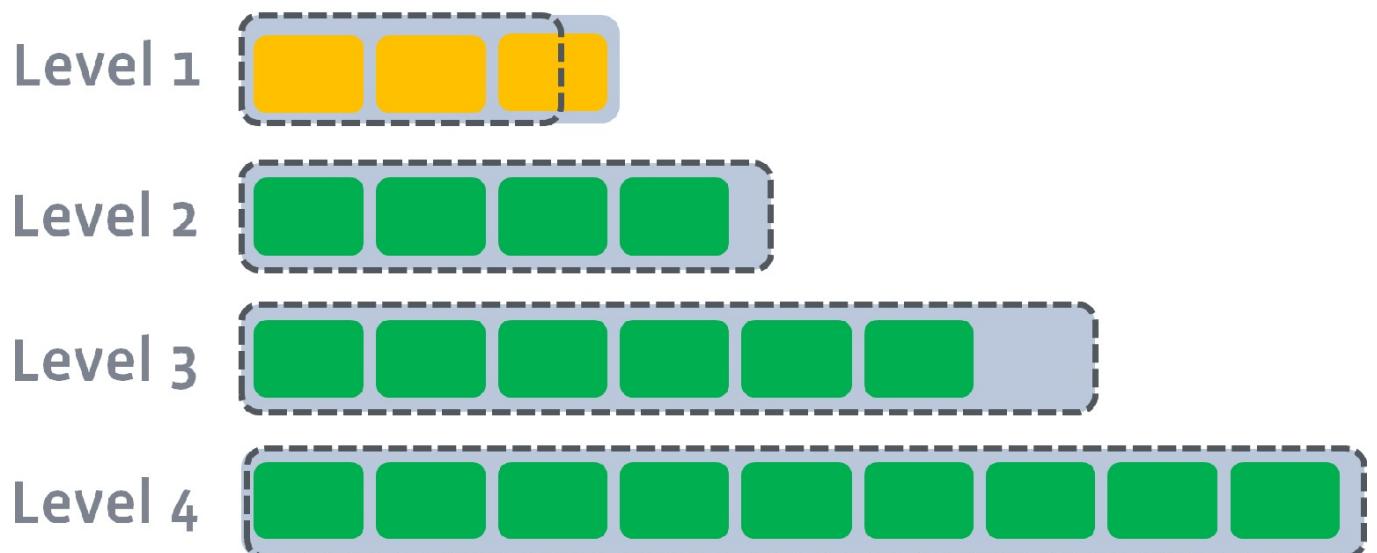
## Compactions

---

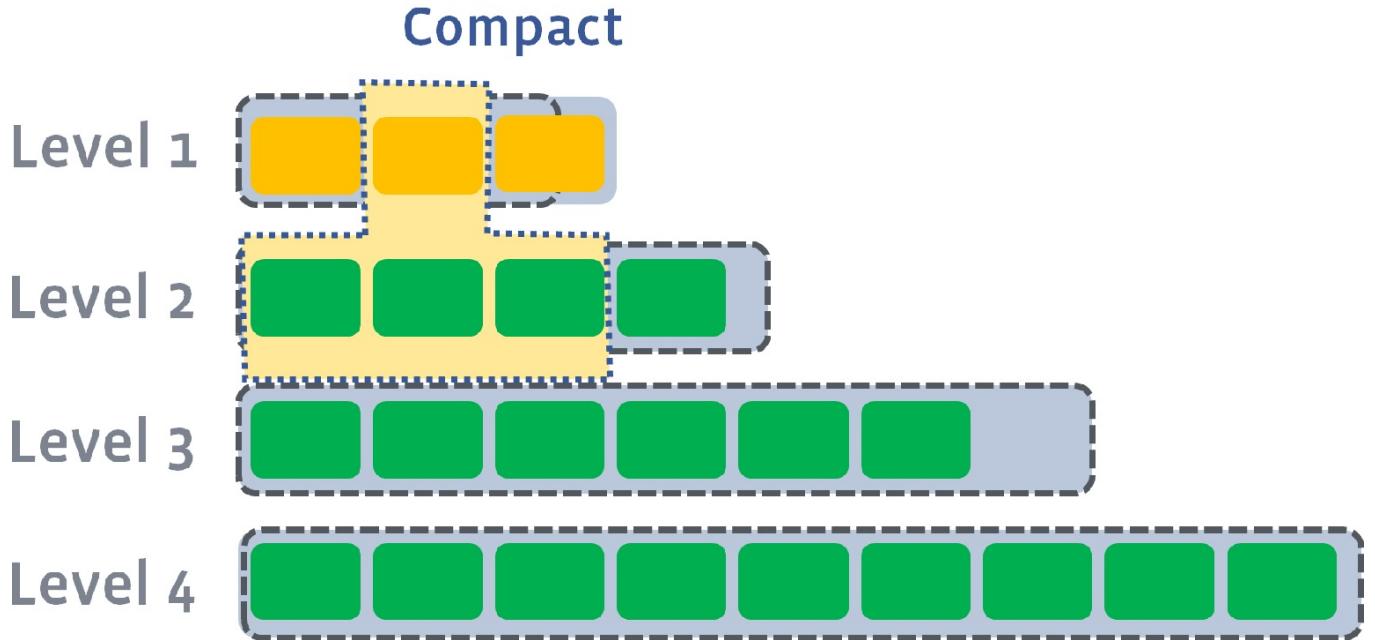
Compaction triggers when number of L0 files reaches `level0_file_num_compaction_trigger`, files of L0 will be merged into L1. Normally we have to pick up all the L0 files because they usually are overlapping:



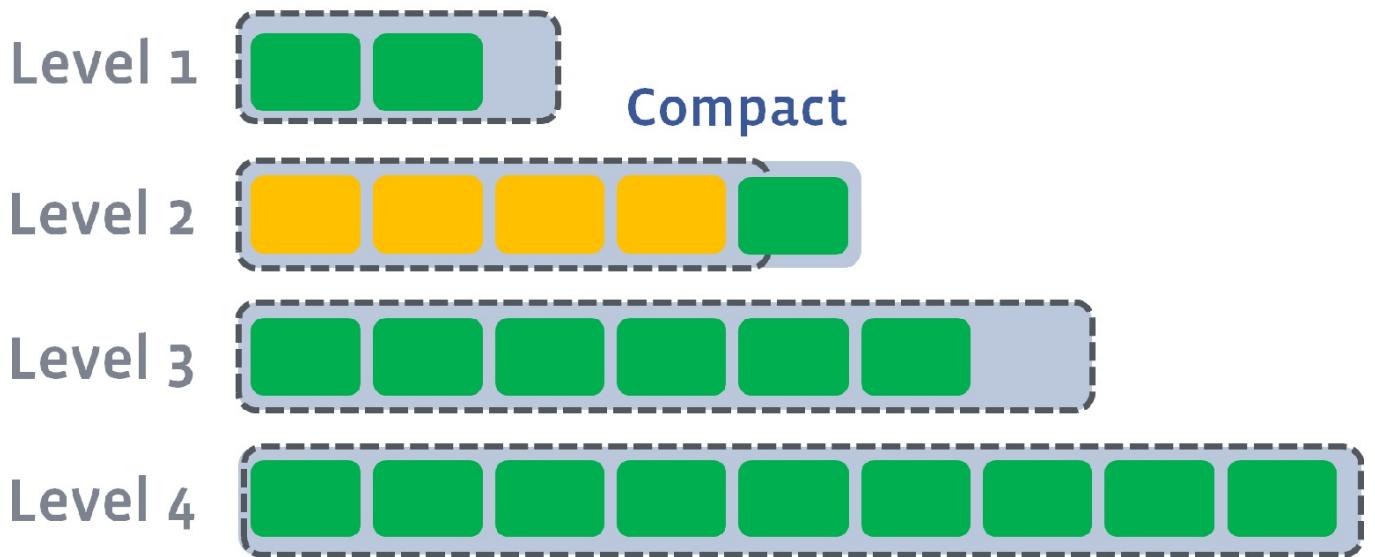
After the compaction, it may push the size of L1 to exceed its target:



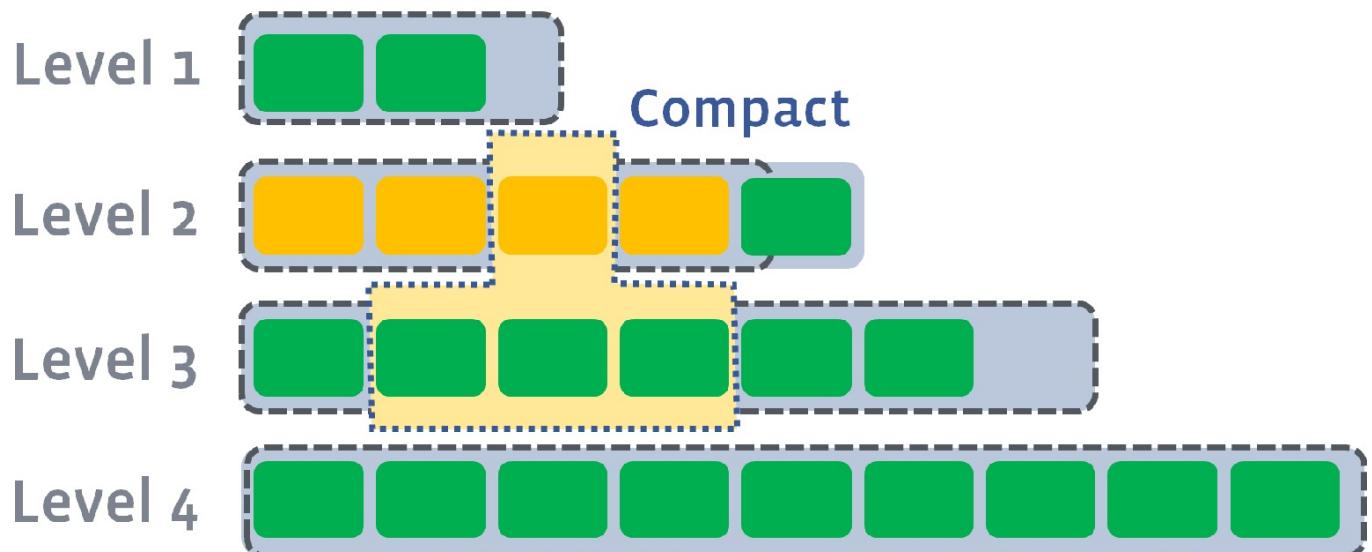
In this case, we will pick at least one file from L1 and merge it with the overlapping range of L2. The result files will be placed in L2:



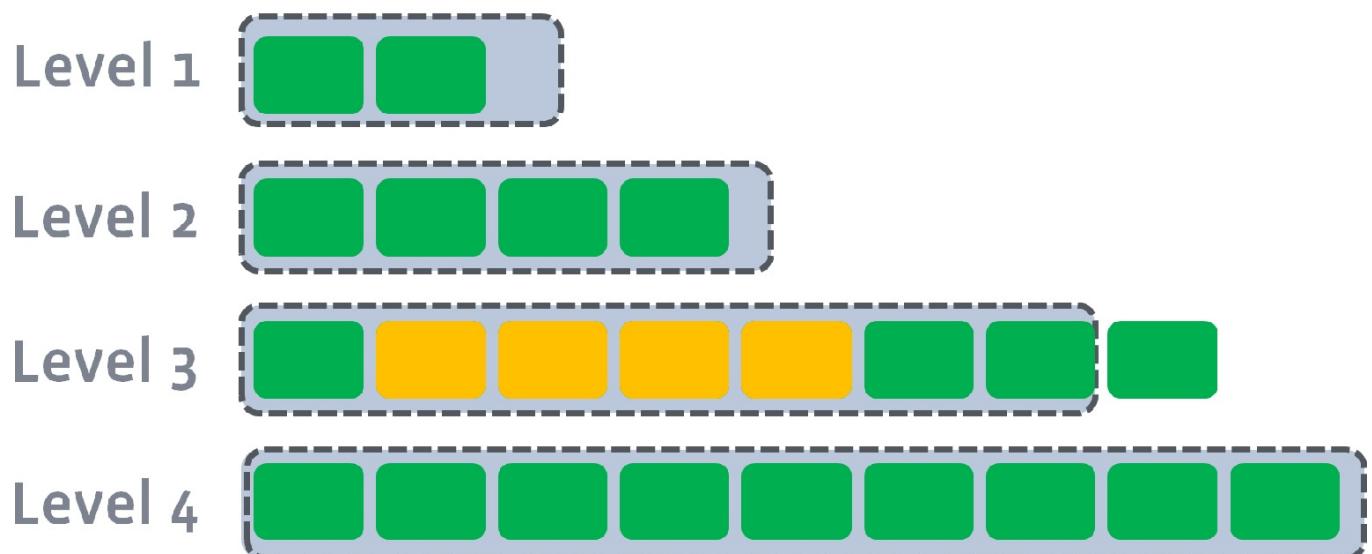
If the results push the next level's size exceeds the target, we do the same as previously – pick up a file and merge it into the next level:



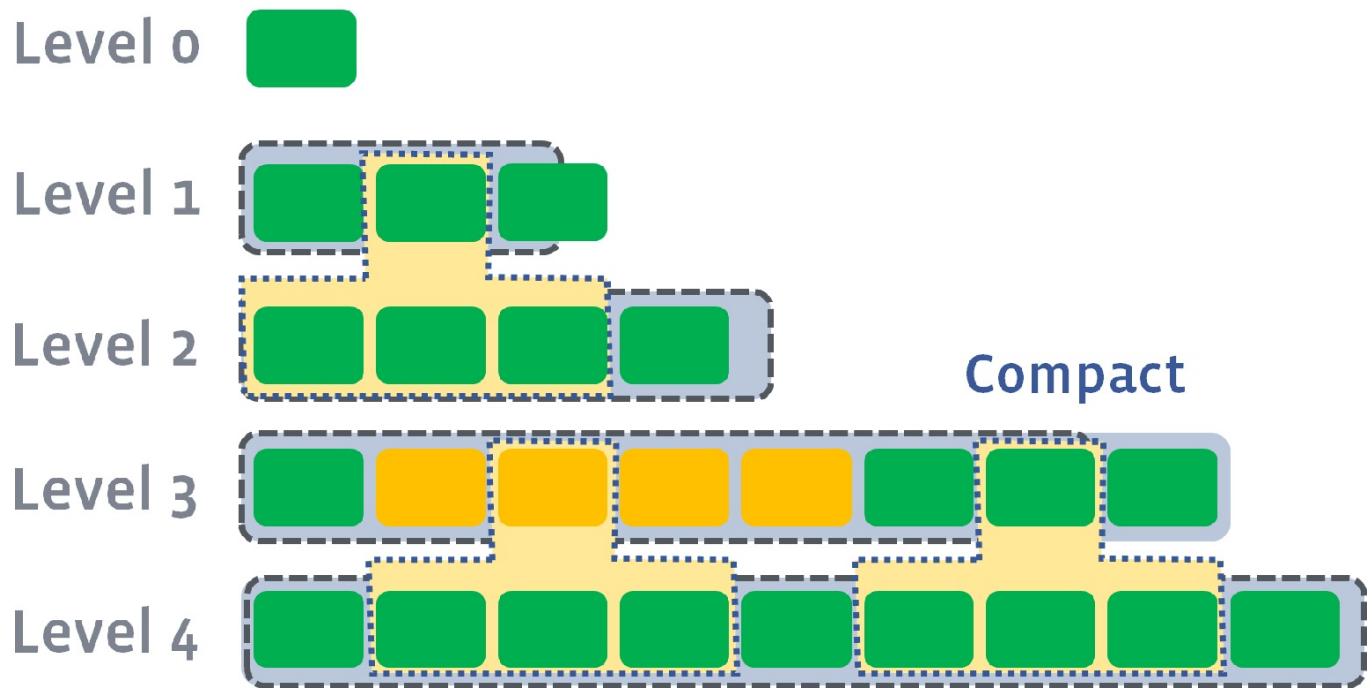
and then



and then

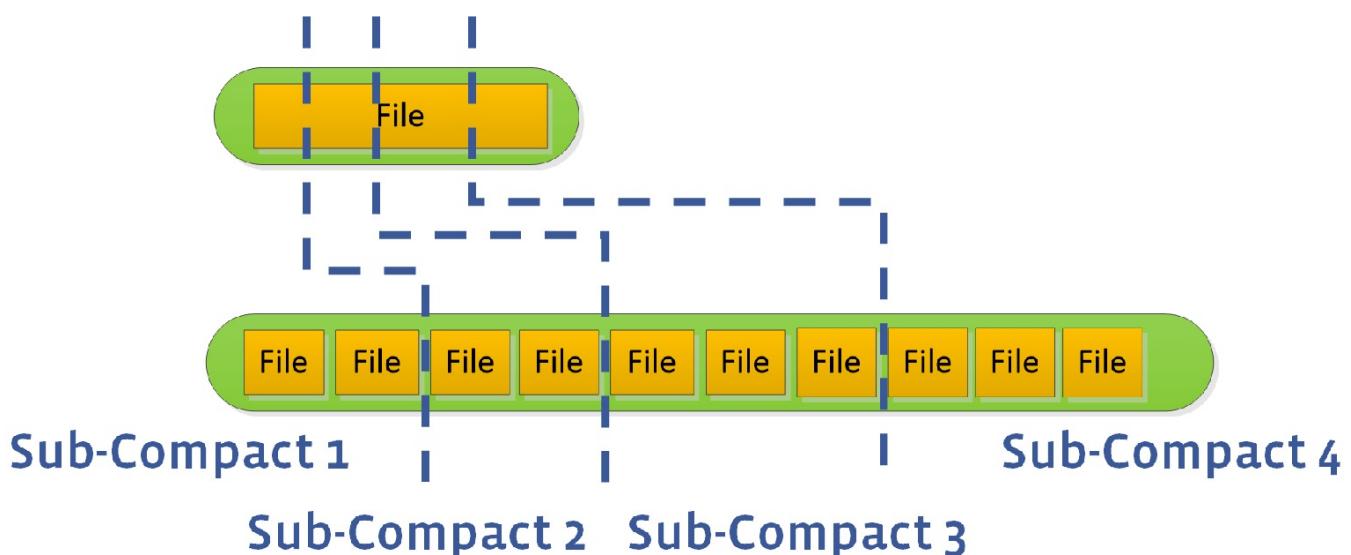


Multiple compactations can be executed in parallel if needed:



Maximum number of compactions allowed is controlled by `max_background_compactions` .

However, L0 to L1 compaction cannot be parallelized. In some cases, it may become a bottleneck that limit the total compaction speed. In this case, users can set `max_subcompactions` to more than 1. In this case, we'll try to partition the range and use multiple threads to execute it:



## Compaction Picking

When multiple levels trigger the compaction condition, RocksDB needs to pick which level to compact first. A score is generated for each level:

- For non-zero levels, the score is total size of the level divided by the target size. If there are already files picked that are being compacted into the next

level, the size of those files is not included into the total size, because they will soon go away.

- for level-0, the score is the total number of files, divided by `level0_file_num_compaction_trigger`, or total size over `max_bytes_for_level_base`, which ever is larger. (if the file size is smaller than `level0_file_num_compaction_trigger`, compaction won't trigger from level 0, no matter how big the score is.)

We compare the score of each level, and the level with highest score takes the priority to compact.

Which file(s) to compact from the level are explained in [Choose Level Compaction Files](#).

## Levels' Target Size

---

`level_compaction_dynamic_level_bytes` is `false`

If `level_compaction_dynamic_level_bytes` is `false`, then level targets are determined as following: L1's target will be `max_bytes_for_level_base`. And then `Target_Size(Ln+1) = Target_Size(Ln) * max_bytes_for_level_multiplier * max_bytes_for_level_multiplier_additional[n]`. `max_bytes_for_level_multiplier_additional` is by default all 1.

For example, if `max_bytes_for_level_base = 16384`, `max_bytes_for_level_multiplier = 10` and `max_bytes_for_level_multiplier_additional` is not set, then size of L1, L2, L3 and L4 will be 16384, 163840, 1638400, and 16384000, respectively.

`level_compaction_dynamic_level_bytes` is `true`

Target size of the last level (`num_levels - 1`) will always be actual size of the level. And then `Target_Size(Ln-1) = Target_Size(Ln) / max_bytes_for_level_multiplier`. We won't fill any level whose target will be lower than `max_bytes_for_level_base / max_bytes_for_level_multiplier`. These levels will be kept empty and all L0 compaction will skip those levels and directly go to the first level with valid target size.

For example, if `max_bytes_for_level_base` is 1GB, `num_levels=6` and the actual size of last level is 276GB, then the target size of L1-L6 will be 0, 0, 0.276GB, 2.76GB, 27.6GB and 276GB, respectively.

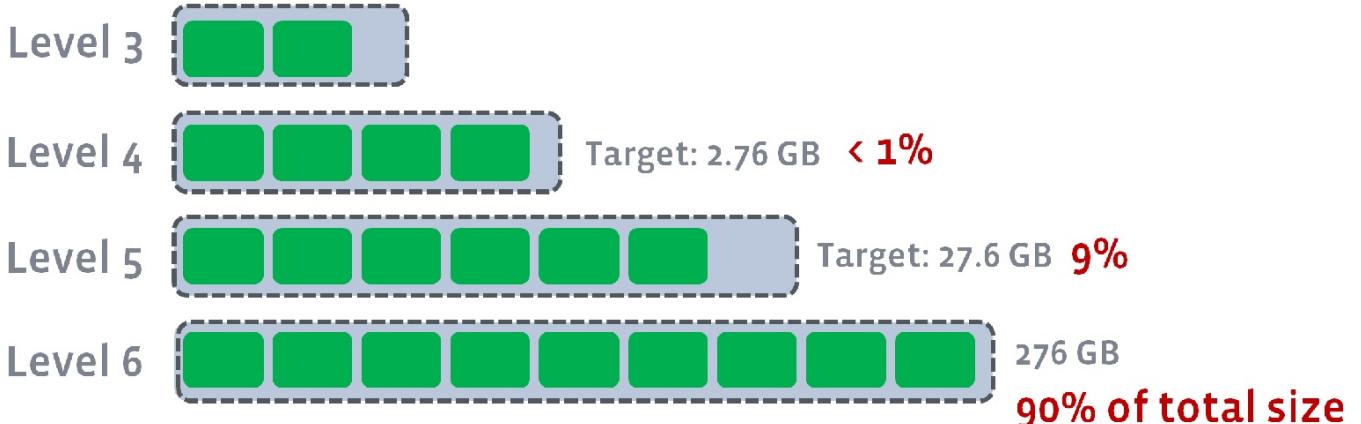
This is to guarantee a stable LSM-tree structure, which can't be guaranteed if `level_compaction_dynamic_level_bytes` is `false`. For example, in the previous example:

## Level 0



## Level 1

## Level 2



We can guarantee 90% of data is stored in the last level, 9% data in the second last level. There will be multiple benefits to it.

## When L0 files piled up

Sometimes writes are heavy, temporarily or permanently, so that number of L0 files piled up before they can be compacted to lower levels. When it happens, the behavior of leveled compaction changes:

### Intra-L0 Compaction

Too many L0 files hurt read performance in most queries. To address the issue, RocksDB may choose to compact some L0 files together to a larger file. This sacrifices write amplification by one but may significantly improve read amplification in L0 and in turn increase the capability RocksDB can hold data in L0. This would generate other benefits which would be explained below. Additional write amplification of 1 is far smaller than the usual write amplification of leveled compaction, which is often larger than 10. So we believe it is a good trade-off. Maximum size of Intra-L0 compaction is also bounded by `options.max_compaction_bytes`. If the option takes a reasonable value, total L0 size will still be bounded, even with Intra-L0 files.

### Adjust level targets

If total L0 size grows too large, it can be even larger than target size of L1, or even lower levels. It doesn't make sense to continue following this configured targets for each level. Instead, for dynamic level, target levels are adjusted. Size of L1 will be adjusted to actual size of L0. And all levels between L1 and the last level

will have adjusted target sizes, so that levels will have the same multiplier. The motivation is to make compaction down to lower levels to happen slower. If data stuck in L0->L1 compaction, it is wasteful to still aggressively compacting lower levels, which competes I/O with higher level compactations.

For example, if configured multiplier is 10, configured base level size is 1GB, and actual L1 to L4 size is 640MB, 6.4GB, 64GB, 640GB, accordingly. If a spike of writes come, and push total L0 size up to 10GB. L1 size will be adjusted to 10GB, and size target of L1 to L4 becomes 10GB, 40GB, 160GB, 640GB. If it is a temporary spike, then actual file size of lower levels are still close to the previous size, which means lower level compaction almost stops, and all the resource is used for L0 => L1 and L1 => L2 compactations, so that it can clear L0 files sooner. In case the high write rate becomes permanent. The adjusted targets's write amplification (expected 14) is better than the configured one (expected 34), so it's a good move.

The goal for this feature is for leveled compaction to handle temporary spike of writes more smoothly. Note that leveled compaction still cannot efficiently handle write rate that is too much higher than capacity based on the configuration. Works on going to further improve it.

## TTL

---

A file could exist in the LSM tree without going through the compaction process for a really long time if there are no updates to the data in the file's key range. For example, in certain use cases, the keys are "soft deleted" – set the values to be empty instead of actually issuing a Delete. There might not be any more writes to this "deleted" key range, and if so, such data could remain in the LSM for a really long time resulting in wasted space.

A dynamic `ttl` column-family option has been introduced to solve this problem. Files (and, in turn, data) older than TTL will be scheduled for compaction when there is no other background work. This will make the data go through the regular compaction process and get rid of old unwanted data. This also has the (good) side-effect of all the data in the non-bottommost level being newer than `ttl`, and all data in the bottommost level older than `ttl`. Note that it could lead to more writes as RocksDB would schedule more compactations.

## Periodic compaction

---

If compaction filter is present, RocksDB ensures that data go through compaction filter after a certain amount of time. This is achieved via

`options.periodic_compaction_seconds` . Setting it to 0 disables this feature. Leaving it the default value, i.e. `UINT64_MAX - 1`, indicates that RocksDB controls the feature. At the moment, RocksDB will change the value to 30 days. Whenever RocksDB tries to pick a compaction, files older than 30 days will be eligible for compaction and be compacted to the same level.

Universal Compaction Style is a compaction style, targeting the use cases requiring lower write amplification, trading off read amplification and space amplification.

## Conceptual Basis

---

As introduced by multiple authors and systems, there are two main types of LSM-tree compaction strategies:

- leveled compaction, as the default compaction style in RocksDB
- an alternative compaction strategy, sometimes called “size tiered” [1] or “tiered” [2].

The key difference between the two strategies is that leveled compaction tends to aggressively merge a smaller sorted run into a larger one, while “tiered” waits for several sorted runs with similar size and merge them together.

It is generally regarded that the second strategy provides far better write amplification with worse read amplification [2][3]. An intuitive way to think about it: in tiered storage, every time an update is compacted, it tends to be moved from a smaller sorted run to a much larger one. Every compaction is likely to make the update exponentially closer to the final sorted run, which is the largest. In leveled compaction, however, an update is compacted more as a part of the larger sorted run where a smaller sorted run is merged into, than as a part of the smaller sorted run. As a result, in most of the times an update is compacted, it is not moved to a larger sorted run, so it doesn’t make much progress towards the final largest run.

The benefit of “tiered” compaction is not without downside. The worse case number of sorted runs is far higher than leveled compaction. It may cause higher I/O costs and/or higher CPU costs during reads. The lazy nature of the compaction scheduling also makes the compaction traffic much more spiky, the number of sorted runs greatly varies over time, hence large variation of performance.

Nevertheless, RocksDB provides a Universal compaction in the “tiered” family. Users may try this compaction style if leveled compaction is not able to handle the required write rate.

[1] The term is used by Cassandra. See their [doc](#).

[2] N. Dayan, M. Athanassoulis, and S. Idreos, “[Monkey: Optimal Navigable Key-Value Store](#),” in ACM SIGMOD International Conference on Management of Data, 2017.

[3] <https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html>

## Overview And the Basic Idea

---

When using this compaction style, all the SST files are organized as sorted runs covering the whole key ranges. One sorted run covers data generated during a time

range. Different sorted runs never overlap on their time ranges. Compaction can only happen among two or more sorted runs of adjacent time ranges. The output is a single sorted run whose time range is the combination of input sorted runs. After any compaction, the condition that sorted runs never overlap on their time ranges still holds. A sorted run can be implemented as an L0 file, or a “level” in which data is stored as key range partitioned files.

The basic idea of the compaction style: with a threshold of number of sorted runs N, we only start compaction when number of sorted runs reaches N. When it happens, we try to pick files to compact so that number of sorted runs is reduced in the most economic way: (1) it starts from the smallest file; (2) one more sorted run is included if its size is no larger than the existing compaction size. The strategy assumes and itself tries to maintain that the sorted run containing more recent data is smaller than ones containing older data.

## Limitations

---

### Double Size Issue

In universal style compaction, sometimes full compaction is needed. In this case, output data size is similar to input size. During compaction, both of input files and the output file need to be kept, so the DB will be temporarily double the disk space usage. Be sure to keep enough free space for full compaction.

### DB (Column Family) Size if num\_levels=1

When using Universal Compaction, if num\_levels = 1, all data of the DB (or Column Family to be precise) is sometimes compacted to one single SST file. There is a limitation of size of one single SST file. In RocksDB, a block cannot exceed 4GB (to allow size to be uint32). The index block can exceed the limit if the single SST file is too big. The size of index block depends on your data. In one of our use cases, we would observe the DB to reach the limitation when the DB grows to about 250GB, using 4K data block size. To mitigate this limitation you can use [partitioned indexes](#).

This problem is mitigated if users set num\_levels to be much larger than 1. In that case, larger “files” will be put in larger “levels” with files divided into smaller files (more explanation below). L0->L0 compaction can still happen for parallel compactions, but most likely files in L0 are much smaller.

## Data Layout and Placement

---

### Sorted Runs

As mentioned above, data is organized as sorted runs. Sorted runs are laid out by updated time of the data in it and stored as either files in L0 or a whole “level”.

Here is an example of a typical file layout:

```

1. Level 0: File0_0, File0_1, File0_2
2. Level 1: (empty)
3. Level 2: (empty)
4. Level 3: (empty)
5. Level 4: File4_0, File4_1, File4_2, File4_3
6. Level 5: File5_0, File5_1, File5_2, File5_3, File5_4, File5_5, File5_6, File5_7

```

Levels with a larger number contain older sorted run than levels of smaller numbers. In this example, there are 5 sorted runs: three files in level 0, level 4 and 5. Level 5 is the oldest sorted run, level 4 is newer, and the level 0 files are the newest.

## Placement of Compaction Outputs

Compaction is always scheduled for sorted runs with consecutive time ranges and the outputs are always another sorted run. We always place compaction outputs to the highest possible level, following the rule of older data on levels with larger numbers.

Use the same example shown above. We have following sorted runs:

```

1. File0_0
2. File0_1
3. File0_2
4. Level 4: File4_0, File4_1, File4_2, File4_3
5. Level 5: File5_0, File5_1, File5_2, File5_3, File5_4, File5_5, File5_6, File5_7

```

If we compact all the data, the output sorted run will be placed in level 5. so it becomes:

```
1. Level 5: File5_0', File5_1', File5_2', File5_3', File5_4', File5_5', File5_6', File5_7'
```

Starting from this state, let's see how to place output sorted runs if we schedule different compactions:

If we compact File0\_1, File0\_2 and Level 4, the output sorted run will be placed in level 4.

```

1. Level 0: File0_0
2. Level 1: (empty)
3. Level 2: (empty)
4. Level 3: (empty)
5. Level 4: File4_0', File4_1', File4_2', File4_3'
6. Level 5: File5_0, File5_1, File5_2, File5_3, File5_4, File5_5, File5_6, File5_7

```

If we compact File0\_0, File0\_1 and File0\_2, the output sorted run will be placed in level 3.

```

1. Level 0: (empty)
2. Level 1: (empty)
3. Level 2: (empty)
4. Level 3: File3_0, File3_1, File3_2
5. Level 4: File4_0, File4_1, File4_2, File4_3
6. Level 5: File5_0, File5_1, File5_2, File5_3, File5_4, File5_5, File5_6, File5_7

```

If we compact File0\_0 and File0\_1, the output sorted run will still be placed in level 0.

```

1. Level 0: File0_0', File0_2
2. Level 1: (empty)
3. Level 2: (empty)
4. Level 3: (empty)
5. Level 4: File4_0, File4_1, File4_2, File4_3
6. Level 5: File5_0, File5_1, File5_2, File5_3, File5_4, File5_5, File5_6, File5_7

```

## Special case options.num\_levels=1

If options.num\_levels=1, we still follow the same placement rule. It means all the files will be placed under level 0 and each file is a sorted run. The behavior will be the same as initial universal compaction, so it can be used as a backward compatible mode.

## Compaction Picking Algorithm

Assuming we have sorted runs

```
1. R1, R2, R3, ..., Rn
```

where R1 containing data of most recent updates to the DB and Rn containing the data of oldest updates of the DB. Note it is sorted by age of the data inside the sorted run, not the sorted run itself. According to this sorting order, after compaction, the output sorted run is always placed into the place where the inputs were.

How is all compactions are picked up:

Precondition:  $n \geq$   
`options.level0_file_num_compaction_trigger`

Unless number of sorted runs reaches this threshold, no compaction will be triggered at all.

(Note although the option name uses word “file”, the trigger is for “sorted run” for historical reason. For the names of all options mentioned below, “file” also means sorted run for the same reason.)

If pre-condition is satisfied, there are three conditions. Each of them can trigger a compaction:

## 1. Compaction Triggered by Space Amplification

If the estimated *size amplification ratio* is larger than `options.compaction_options_universal.max_size_amplification_percent / 100`, all files will be compacted to one single sorted run.

Here is how *size amplification ratio* is estimated:

```
1. size amplification ratio = (size(R1) + size(R2) + ... size(Rn-1)) / size(Rn)
```

Please note, size of Rn is not included, which means 0 is the perfect size amplification and 100 means DB size is double the space of live data, and 200 means triple.

The reason we estimate size amplification in this way: in a stable sized DB, incoming rate of deletion should be similar to rate of insertion, which means for any of the sorted runs except Rn should include similar number of insertion and deletion. After applying R1, R2 ... Rn-1, to Rn, the size effects of them will cancel each other, so the output should also be size of Rn, which is the size of live data, which is used as the base of size amplification.

If `options.compaction_options_universal.max_size_amplification_percent = 25`, which means we will keep total space of DB less than 125% of total size of live data. Let's use this value in an example. Assuming compaction is only triggered by space amplification, `options.level0_file_num_compaction_trigger = 1`, file size after each mem table flush is always 1, and compacted size always equals to total input sizes. After two flushes, we have two files size of 1, while  $1/1 > 25\%$  so we'll need to do a full compaction:

```
1. 1 1 => 2
```

After another mem table flush we have

```
1. 1 2 => 3
```

Which again triggers a full compaction because  $1/2 > 25\%$ . And in the same way:

```
1. 1 3 => 4
```

But after another flush, the compaction is not triggered:

```
1. 1 4
```

Because  $1/4 \leq 25\%$ . Another mem table flush will trigger another compaction:

```
1. 1 1 4 => 6
```

Because  $(1+1) / 4 > 25\%$ .

And it keeps going like this:

```
1. 1
2. 1 1 => 2
3. 1 2 => 3
4. 1 3 => 4
5. 1 4
6. 1 1 4 => 6
7. 1 6
8. 1 1 6 => 8
9. 1 8
10. 1 1 8
11. 1 1 1 8 => 11
12. 1 11
13. 1 1 11
14. 1 1 1 11 => 14
15. 1 14
16. 1 1 14
17. 1 1 1 14
18. 1 1 1 1 14 => 18
```

## 2. Compaction Triggered by Individual Size Ratio

We calculated a value of *size ratio trigger* as

```
1. size_ratio_trigger = (100 + options.compaction_options_universal.size_ratio) / 100
```

Usually `options.compaction_options_universal.size_ratio` is close to 0 so *size ratio trigger* is close to 1.

We start from R1, if  $\text{size}(R2) / \text{size}(R1) \leq \text{size ratio trigger}$ , then (R1, R2) are qualified to be compacted together. We continue from here to determine whether R3 can be added too. If  $\text{size}(R3) / \text{size}(R1 + R2) \leq \text{size ratio trigger}$ , we would include (R1, R2, R3). Then we do the same for R4. We keep comparing total existing size to the next sorted run until the *size ratio trigger* condition doesn't hold any more.

Here is an example to make it easier to understand. Assuming `options.compaction_options_universal.size_ratio = 0`, total mem table flush size is always 1, compacted size always equals to total input sizes, compaction is only triggered by space amplification and `options.level0_file_num_compaction_trigger = 5`. Starting from an empty DB, after 5 mem table flushes, we have 5 files size of 1, which triggers a compaction of all files because  $1/1 \leq 1$ ,  $1/(1+1) \leq 1$ ,  $1/(1+1+1) \leq 1$  and

```
1/(1+1+1+1) <= 1:
```

```
1. 1 1 1 1 => 5
```

After 4 mem table flushes make it 5 files again. First 4 files qualifies for merging:  
 $1/1 \leq 1$ ,  $1/(1+1) \leq 1$ ,  $1/(1+1+1) \leq 1$ . While the 5th one doesn't:  $5/(1+1+1+1) > 1$ :

```
1. 1 1 1 5 => 4 5
```

They go on like that for several rounds:

```
1. 1 1 1 1 1 => 5
2. 1 5 (no compaction triggered)
3. 1 1 5 (no compaction triggered)
4. 1 1 1 5 (no compaction triggered)
5. 1 1 1 1 5 => 4 5
6. 1 4 5 (no compaction triggered)
7. 1 1 4 5 (no compaction triggered)
8. 1 1 1 4 5 => 3 4 5
9. 1 3 4 5 (no compaction triggered)
10. 1 1 3 4 5 => 2 3 4 5
```

Another flush brings it to be like

```
1. 1 2 3 4 5
```

And no compaction is triggered, so we hold the compaction. Only when another flush comes, all files are qualified to compact together:

```
1. 1 1 2 3 4 5 => 16
```

Because  $1/1 \leq 1$ ,  $2/(1+1) \leq 1$ ,  $3/(1+1+2) \leq 1$ ,  $4/(1+1+2+3) \leq 1$  and  $5/(1+1+2+3+4) \leq 1$ . And we continue from there:

```
1. 1 16 (no compaction triggered)
2. 1 1 16 (no compaction triggered)
3. 1 1 1 16 (no compaction triggered)
4. 1 1 1 1 16 => 4 16
5. 1 4 16 (no compaction triggered)
6. 1 1 4 16 (no compaction triggered)
7. 1 1 1 4 16 => 3 4 16
8. 1 3 4 16 (no compaction triggered)
9. 1 1 3 4 16 => 2 3 4 16
10. 1 2 3 4 16 (no compaction triggered)
11. 1 1 2 3 4 16 => 11 16
```

Compaction is only triggered when number of input sorted runs would be at least

`options.compaction_options_universal.min_merge_width` and number of sorted runs as inputs will be capped as no more than `options.compaction_options_universal.max_merge_width`.

### 3. Compaction Triggered by number of sorted runs

If for every time we try to schedule a compaction, neither of 1 and 2 are triggered, we will try to compact whatever sorted runs from R1, R2..., so that after the compaction the total number of sorted runs is not greater than `options.level0_file_num_compaction_trigger`. If we need to compact more than `options.compaction_options_universal.max_merge_width` number of sorted runs, we cap it to `options.compaction_options_universal.max_merge_width`.

“Try to schedule” I mentioned below will happen when after flushing a memtable, finished a compaction. Sometimes duplicated attempts are scheduled.

See [Universal Style Compaction Example](#) as an example of how output sorted run sizes look like for a common setting.

Parallel compactions are possible if `options.max_background_compactions > 1`. Same as all other compaction styles, parallel compactions will not work on the same sorted run.

### 4. Compaction triggered by age of data

For universal style compaction, the aging-based triggering criterion has the highest priority since it is a hard requirement. When trying to pick a compaction, this condition is checked first. If the condition meets (there are files older than `options.periodic_compaction_seconds`), then RocksDB proceeds to pick sorted runs for compaction. RocksDB picks sorted runs from oldest to youngest until encountering a sorted run that is being compacted by another compaction. These files will be compacted to bottommost level unless bottommost level is reserved for ingestion behind. In this case, files will be compacted to second bottommost level.

## Subcompaction

---

Subcompaction is supported in universal compaction. If the output level of a compaction is not “level” 0, we will try to range partition the inputs and use number of threads of `options.max_subcompaction` to compact them in parallel. It will help with the problem that full compaction of universal compaction takes too long.

## Options to Tune

---

Following are options affecting universal compactions:

- `options.compaction_options_universal`: various options mentioned above
- `options.level0_file_num_compaction_trigger` the triggering condition of any

- compaction. It also means after all compactions' finishing, number of sorted runs will be under options.level0\_file\_num\_compaction\_trigger+1
- options.level0\_slowdown\_writes\_trigger: if number of sorted runs exceeds this value, writes will be artificially slowed down.
  - options.level0\_stop\_writes\_trigger: if number of sorted runs exceeds this value, writes will stop until compaction finishes and number of sorted runs turns under this threshold.
  - options.num\_levels: if this value is 1, all sorted runs will be stored as level 0 files. Otherwise, we will try to fill non-zero levels as much as possible. The larger num\_levels is, the less likely we will have large files on level 0.
  - options.target\_file\_size\_base: effective if options.num\_levels > 1. Files of levels other than level 0 will be cut to file size not larger than this threshold.
  - options.target\_file\_size\_multiplier: it is effective, but we don't know a way to use this option in universal compaction that makes sense. So we don't recommend you to tune it.

Following options **DO NOT** affect universal compactions:

- options.max\_bytes\_for\_level\_base: only for level-based compaction
- options.level\_compaction\_dynamic\_level\_bytes: only for level-based compaction
- options.max\_bytes\_for\_level\_multiplier and  
options.max\_bytes\_for\_level\_multiplier\_additional: only for level-based compaction
- options.expanded\_compaction\_factor: only for level-based compactions
- options.source\_compaction\_factor: only for level-based compactions
- options.max\_grandparent\_overlap\_factor: only for level-based compactions
- options.soft\_rate\_limit and options.hard\_rate\_limit: deprecated
- options.hard\_pending\_compaction\_bytes\_limit: only for level-based compaction
- options.compaction\_pri: only for level-based compaction

## Estimate Write Amplification

---

Estimating write amplification will be very helpful to users to tune universal compaction. This, however, is hard. Since universal compaction always makes locally optimized decision, the shape of the LSM-tree is hard to predict. You can see it from the [example](#) mentioned above. We still don't have a good Math model to predict the write amplification.

Here is a not-so-good estimation.

The estimation based on the simplicity that every time an update is compacted, the size of output sorted run is doubled of the original one (this is a wild unproven estimation), with the exception of the first or last compaction it experienced, where sorted runs of similar sizes are compacted together.

Take an example, if

`options.compaction_options_universal.max_size_amplification_percent = 25`, last sorted run's size is 256GB, and SST file size is 256MB after flushed from memtable, and `options.level0_file_num_compaction_trigger = 11`. Then in a stable stage, the file sizes are like this:

1. 256MB
2. 256MB
3. 256MB
4. 256MB
5. 2GB
6. 4GB
7. 8GB
8. 16GB
9. 16GB
10. 16GB
11. 256GB

Compaction stages are like following with its write amp:

1. 256MB
2. 256MB
3. 256MB (write amp 1)
4. 256MB
5. -----
6. 2GB (write amp 1)
7. -----
8. 4GB (write amp 1)
9. -----
10. 8GB (write amp 1)
11. -----
12. 16GB
13. 16GB (write amp 1)
14. 16GB
15. -----
16. 256GB (write amp 4)

So the total write amp is estimated to be 9.

Here is how the write amplification is estimated:

`options.compaction_options_universal.max_size_amplification_percent` always introduces write amplification by itself it is much lower than 100. This write amplification is estimated to be

$WA1 = 100 / options.compaction_options_universal.max_size_amplification_percent$ .

If this is not lower than 100, estimate

$WA1 = 0$

Estimate total data size other than the last sorted run, S. If

*options.compaction\_options\_universal.max\_size\_amplification\_percent < 100*, estimate it using

```
S = total_size \
(options.compaction_options_universal.max_size_amplification_percent/100)*
```

Otherwise

```
S = total_size
```

Estimate SST file size flushed from memtable to be  $M$ . And we estimate maximum number of compactations for an update to reach maximum as:

```
p = log(2, S/M)
```

It is recommended that  $options.level0_file_num_compaction_trigger > p$ . And then we estimate the write amplification because of individual size ratio using:

```
WA2 = p - log(2, options.level0_file_num_compaction_trigger - p)
```

And then the total estimated write amplification would be  $WA1 + WA2$ .

FIFO compaction style is the simplest compaction strategy. It is suited for keeping event log data with very low overhead (query log for example). It periodically deletes the old data, so it's basically a TTL compaction style.

In FIFO compaction, all files are in level 0. When total size of the data exceeds configured size (`CompactionOptionsFIFO::max_table_files_size`), we delete the oldest table file. This means that write amplification of data is always 1 (in addition to WAL write amplification).

Currently, `CompactRange()` function just manually triggers the compaction and deletes the old table files if there is need. It ignores the parameters of the function (begin and end keys).

Since we never rewrite the key-value pair, we also don't ever apply the compaction filter on the keys.

Please use FIFO compaction style with caution. Unlike other compaction style, it can drop data without informing users.

## Compaction

---

FIFO compactations can end up with lots of L0 files. Queries can be slow because we need to search all those files in the worst case in a query. Even Bloom filters may not be able to offset all of them. With typical 1% false positive, 1000 L0 files will cause 10 false positive cases in average, and generate 10 I/Os per query in the worst case. More Bloom bits per key, such as 20, is highly recommended to reduce the false positive rate, especially with `format_version=5 Bloom filter`, but more Bloom bits does use more memory. In some cases, even the CPU overhead of Bloom filter checking is too high.

To solve this case, users can choose to enable some lightweight compactations to happen. This will potentially double the write I/O, but can significantly reduce number of L0 files. This sometimes is the right trade-off for users.

The feature is introduced in 5.5 release. Users can enable it using

```
CompactionOptionsFIFO.allow_compaction = true . And it tries to pick up at least
level0_file_num_compaction_trigger files flushed from memtable and compact them together.
```

To be specific, we always start with the latest `level0_file_num_compaction_trigger` files and try to include more files in the compaction. We calculated compacted bytes per file deducted using `total_compaction_size / (number_files_in_compaction - 1)`. We always pick the files so that this number is minimized and is no more than `options.write_buffer_size`. In typical workloads, it will always compact `level0_file_num_compaction_trigger` freshly flushed files.

For example, if `level0_file_num_compaction_trigger = 8` and every flushed file is 100MB. Then as soon as there is 8 files, they are compacted to one 800MB file. And after we have 8 new 100MB files, they are compacted in the second 800MB, and so on. Eventually we'll have a list of 800MB files and no more than 8 100MB files.

Please note that, since the oldest files are compacted, the file to be deleted by FIFO is also larger, so potentially slightly less data is stored than without compaction.

## FIFO Compaction with TTL

---

A new feature called FIFO compaction with TTL has been introduced starting in RocksDB 5.7. ([2480](#))

So far, FIFO compaction worked by just taking the total file size into consideration, like: drop the oldest files if the db size exceeds `compaction_options_fifo.max_table_files_size`, one at a time until the total size drops below the set threshold. This sometimes makes it tricky to enable in production as use cases can have organic growth.

A new option, `ttl`, has been introduced for this to delete SST files for which the TTL has expired. This feature enables users to drop files based on time rather than always based on size, say, drop all SST files older than a week or a month. [`ttl` option before 6.0 is part of `ColumnFamilyOptions.compaction_options_fifo` struct. It moved into `ColumnFamilyOptions` from 6.0 onwards].

Constraints:

- This currently works only for Block based table format and when `max_open_files` is set to -1 since it requires all table files to be open.
- FIFO with TTL still works within the bounds of the configured size, i.e. RocksDB temporarily falls back to FIFO deletion based on size if it observes that deleting expired files does not bring the total size to be less than the configured max size.

Compaction can be triggered manually by calling the `DB::CompactRange` or `DB::CompactFiles` method. This is meant to be used by advanced users to implement custom compaction strategies, including but not limited to the following use cases -

1. Optimize for read heavy workloads by compacting to lowest level after ingesting a large amount of data
2. Force the data to go through the compaction filter in order to consolidate it
3. Migrate to a new compaction configuration. For example, if changing number of levels, `CompactRange` can be called to compact to bottommost level and then move the files to a target level

The example code below shows how to use the APIs.

```

1. Options dbOptions;
2.
3. DB* db;
4. Status s = DB::Open(dbOptions, "/tmp/rocksdb", &db);
5.
6. // Write some data
7. ...
8. Slice begin("key1");
9. Slice end("key100");
10. CompactRangeOptions options;
11.
12. s = db->CompactRange(options, &begin, &end);

```

Or

```

1. CompactionOptions options;
2. std::vector<std::string> input_file_names;
3. int output_level;
4. ...
5. Status s = db->CompactFiles(options, input_file_names, output_level);

```

## CompactRange

The `begin` and `end` arguments define the key range to be compacted. The behavior varies depending on the compaction style being used by the db. In case of universal and FIFO compaction styles, the `begin` and `end` arguments are ignored and all files are compacted. Also, files in each level are compacted and left in the same level. For leveled compaction style, all files containing keys in the given range are compacted to the last level containing files. If either `begin` or `end` are NULL, it is taken to mean the key before all keys in the db or the key after all keys respectively.

If more than one thread calls manual compaction, only one will actually schedule it while the other threads will simply wait for the scheduled manual compaction to complete. If `CompactRangeOptions::exclusive_manual_compaction` is set to true, the call will disable scheduling of automatic compaction jobs and wait for existing automatic

compaction jobs to finish.

`DB::CompactRange` waits while compaction is performed on the background threads and thus is a blocking call.

The `CompactRangeOptions` supports the following options -

- `CompactRangeOptions::exclusive_manual_compaction` When set to true, no other compaction will run when this manual compaction is running. Default value is `true`
- `CompactRangeOptions::change_level` , `CompactRangeOptions::target_level` Together, these options control the level where the compacted files will be placed. If `target_level` is -1, the compacted files will be moved to the minimum level whose computed `max_bytes` is still large enough to hold the files. Intermediate levels must be empty. For example, if the files were initially compacted to L5 and L2 is the minimum level large enough to hold the files, they will be placed in L2 if L3 and L4 are empty or in L4 if L3 is non-empty. If `target_level` is positive, the compacted files will be placed in that level provided intermediate levels are empty. If any any of the intermediate levels are not empty, the compacted files will be left where they are.
- `CompactRangeOptions::target_path_id` Compaction outputs will be placed in `options.db_paths[target_path_id]` directory.
- `CompactRangeOptions::bottommost_level_compaction` When set to `BottommostLevelCompaction::kSkip` , or when set to `BottommostLevelCompaction::kIfHaveCompactionFilter` and a compaction filter is defined for the column family, the bottommost level files are not compacted.

## CompactFiles

---

This API compacts all the input files into a set of output files in the `output_level` . The number of output files is determined by the size of the data and the setting of `CompactionOptions::output_file_size_limit` . This API is not supported in ROCKSDB\_LITE.

Here we explain the sub-compaction that is used in both Leveled and Universal compactions.

## Goal

The goal of sub-compaction is to speed up a compaction job by partitioning it among multiple threads.

## When

It is employed when one of the following conditions holds:

- $L0 \rightarrow L0$  where  $o > 0$ 
  - Why:  $L0 \rightarrow L0$  cannot be run in parallel with another  $L0 \rightarrow L0$ , hence partitioning is the only way to speed it up.
- Universal Compaction, except  $L0 \rightarrow L0$ .
- Manual Leveled Compaction:  $L_i \rightarrow L_0$  where  $o > 0$ 
  - Why: Manual compaction invoked by the user is usually not parallelized hence benefits from partitioning.

Note: sub-compaction is disabled for leveled if it is not merged with any file from the target level  $L_0$ . Refer to `Compaction::ShouldFormSubcompactions` for details.

## How

It currently is based on a heuristic that worked out well so far. The heuristic could be improved in many ways.

1. Select boundaries based on the natural boundary of input levels/files.
  - first and last key of  $L_0$  files
  - first and last key of non- $0$ , non-last levels
  - first key of each SST file of the last level
2. Use `Versions::ApproximateSize` to estimate the data size in each boundary.
3. Merge boundaries to eliminate empty and smaller-than-average ranges.
  - find the average size in each range
  - starting from beginning greedily merge adjacent ranges until their total size exceeds the average

# Introduction

Level compaction style is the default compaction style of RocksDB, so it is also the most widely used compaction style among users. Sometimes users are curious that how level compaction chooses which files to be compacted in each compaction. In this wiki,



we will elaborate more on this topic to save your time in reading code.

## Steps

1. Go from level 0 to highest level to pick the level,  $L_b$ , that the score of this level is the largest and is larger than 1 as the compaction base level.
2. Determine the compaction output level  $L_o = L_b + 1$
3. According to different [compaction priority options](#), find the first file that should be compacted with the highest priority. If the file or its parents on  $L_o$ (The files the key ranges of which overlap) are being compacted by another compaction job, skip to the file with the secondary highest priority until find **one** candidate file. Adds this file into compaction **inputs**.
4. Keep expanding inputs until we are sure that there is a “clean cut” boundary between the files in input and the surrounding files. This will ensure that no parts of a key are lost during compaction. For example, we have five files with key range:

```
1. f1[a1 a2] f2[a3 a4] f3[a4 a6] f4[a6 a7] f5[a8 a9]
```

If we choose  $f_3$  in step 3, then in step 4, we have to expand inputs from  $\{f_3\}$  to  $\{f_2, f_3, f_4\}$  because the boundary of  $f_2$  and  $f_3$  are continuous, as are  $f_3$  and  $f_4$ . The reason that two files may share the same user key boundary is that RocksDB store InternalKey in files which consists of user key, sequence number of key type. So files may store multiple InternalKeys with the same user key. Therefore, if compaction happens, all the InternalKeys with the same user key have to be compacted altogether.

5. Check that the files in current **inputs** don't overlap with any files being compacted. Otherwise, try to find any manual compaction available. If not, abort this compaction pick job.
6. Find all the files on  $L_o$  that overlap with the files in **inputs** and expand them as the same way in step 4 until we find a “clean cut” of files on  $L_o$ . If anyone of these files are being compacted, abort this compaction pick job. Otherwise, put

them into **output\_level\_inputs**.

7. An optional optimization step. See if we can further grow the number of inputs on Lb without changing the number of Lo files we pick up. We also choose NOT to expand if this would cause Lb to include InternalKey for some user key, while excluding other InternalKey for the same user key, i.e., "a non-clean cut". This can happen when one user key spans multiple files. The previous sentence may be confusing by words, so here I give an example to illustrate this expansion optimization.

Consider such a case:

- ```
1. Lb: f1[B E] f2[F G] f3[H I] f4[J M]  
2. Lo: f5[A C] f6[D K] f7[L O]
```

If we initially pick f2 in step 3, now we will compact f2(**inputs**) and f6(**output\_level\_inputs** in step 4). But we can safely compact f2, f3 and f6 without expanding the output level.

8. The files in **inputs** and **output\_level\_inputs** are the candidate files for this level compaction.



Cheers!

# Overview

---

It is possible to cap the total amount of disk space used by a RocksDB database instance, or multiple instances in aggregate. This might be useful in cases where a file system is shared by multiple applications and the other applications need to be insulated from unlimited database growth.

The tracking of disk space utilization and, optionally, limiting the database size is done by `rocksdb::SstFileManager`. It is allocated by calling `NewSstFileManager()` and the returned object is assigned to `DBOptions::sst_file_manager`. It is possible for multiple DB instances to share an `SstFileManager`.

## Usage

---

### Tracking DB Size

A caller can get the total disk space used by the DB by calling `SstFileManager::GetTotalSize()`. It returns the total size, in bytes, of all the SST files. WAL files are not included in the calculation. If the same `SstFileManager` object is shared by multiple DB instances, `GetTotalSize()` will report the size across all the instances.

### Limiting the DB Size

By calling `SstFileManager::SetMaxAllowedSpaceUsage()` and, optionally, `SstFileManager::SetCompactionBufferSize()`, it is possible to set limits on how much disk space is used. Both functions accept a single argument specifying the desired size in bytes. The former sets a hard limit on the DB size, and the latter specifies headroom that should be reserved before deciding whether to allow a compaction to proceed or not.

Setting the max DB size limit can impact the operation of the DB in the following ways -

1. Every time a new SST file is created, either by flush or compaction, `SstFileManager::OnAddFile()` is called to update the total size used. If this causes the total size to go above the limit, the `ErrorHandler::bg_error_` variable is set to `Status::SpaceLimit()` and the DB instance that created the SST file goes into read-only mode. For more information on how to recover from this situation, see [Background Error Handling](#).
2. Before starting a compaction, RocksDB will check if there is enough room to create the output SST files. This is done by calling `SstFileManager::EnoughRoomForCompaction()`. This function conservatively estimates the output size as the sum of the sizes of all the input SST files to the compaction.

If the output size, plus the compaction buffer size if its set, will cause the total size to exceed the limit set by `SstFileManager::SetMaxAllowedSpaceUsage()`, the compaction is not allowed to proceed. The compaction thread will sleep for 1 second before adding the column family back to the compaction queue. So this effectively throttles the compaction rate.

- [Block-based Table Format](#)
- [PlainTable Format](#)
- [CuckooTable Format](#)
- [Index Block Format](#)
- [Bloom Filter](#)
- [Data Block Hash Index](#)

This page is forked from LevelDB's document on [table format](#), and reflects changes we have made during the development of RocksDB.

`BlockBasedTable` is the default SST table format in RocksDB.

## File format

```

1. <beginning_of_file>
2. [data block 1]
3. [data block 2]
4. ...
5. [data block N]
6. [meta block 1: filter block]           (see section: "filter" Meta Block)
7. [meta block 2: index block]
8. [meta block 3: compression dictionary block] (see section: "compression dictionary" Meta Block)
9. [meta block 4: range deletion block]     (see section: "range deletion" Meta Block)
10. [meta block 5: stats block]            (see section: "properties" Meta Block)
11. ...
12. [meta block K: future extended block] (we may add more meta blocks in the future)
13. [metaindex block]
14. [Footer]                            (fixed size; starts at file_size - sizeof(Footer))
15. <end_of_file>
```

The file contains internal pointers, called `BlockHandles`, containing the following information:

```

1. offset:      varint64
2. size:        varint64
```

See [this document](#) for an explanation of varint64 format.

(1) The sequence of key/value pairs in the file are stored in sorted order and partitioned into a sequence of data blocks. These blocks come one after another at the beginning of the file. Each data block is formatted according to the code in `block_builder.cc` (see code comments in the file), and then optionally compressed.

(2) After the data blocks, we store a bunch of meta blocks. The supported meta block types are described below. More meta block types may be added in the future. Each meta block is again formatted using `block_builder.cc` and then optionally compressed.

(3) A `metaindex` block contains one entry for every meta block, where the key is the name of the meta block and the value is a `BlockHandle` pointing to that meta block.

(4) At the very end of the file is a fixed length footer that contains the `BlockHandle` of the `metaindex` and index blocks as well as a magic number.

```

1. metaindex_handle: char[p];    // Block handle for metaindex
2. index_handle:      char[q];    // Block handle for index
3. padding:          char[40-p-q]; // zeroed bytes to make fixed length
```

```

4.                                     // (40==2*BlockHandle::kMaxEncodedLength)
5.     magic:           fixed64;      // 0x88e241b785f4cff7 (little-endian)

```

## Index Block

Index blocks are used to look up a data block containing the range including a lookup key. It is a binary search data structure. A file may contain one index block, or a list of partitioned index blocks (see [Partitioned Index Filters](#)). Index block format is documented here: [Index Block Format](#).

## Filter Meta Block

Note: `format_version \=5` (Since RocksDB 6.6) uses a faster and more accurate [Bloom filter implementation](#) for full and partitioned filters.

### Full filter

In this filter there is one filter block for the entire SST file.

### Partitioned Filter

The full filter is partitioned into multiple blocks. A top-level index block is added to map keys to corresponding filter partitions. Read more [here](#).

### Block-based filter

Note: the below explains block based filter, which is deprecated.

If a “FilterPolicy” was specified when the database was opened, a filter block is stored in each table. The “metaindex” block contains an entry that maps from “filter.” to the BlockHandle for the filter block, where “” is the string returned by the filter policy’s `Name()` method.

The filter block stores a sequence of filters, where filter `i` contains the output of `FilterPolicy::CreateFilter()` on all keys that are stored in a block whose file offset falls within the range

```
1. [ i*base ... (i+1)*base-1 ]
```

Currently, “base” is 2KB. So, for example, if blocks X and Y start in the range [ 0KB .. 2KB-1 ], all of the keys in X and Y will be converted to a filter by calling `FilterPolicy::CreateFilter()`, and the resulting filter will be stored as the first filter in the filter block.

The filter block is formatted as follows:

```

1. [filter 0]
2. [filter 1]
3. [filter 2]
4. ...

```

```

5. [filter N-1]
6.
7. [offset of filter 0] : 4 bytes
8. [offset of filter 1] : 4 bytes
9. [offset of filter 2] : 4 bytes
10. ...
11. [offset of filter N-1] : 4 bytes
12.
13. [offset of beginning of offset array] : 4 bytes
14. lg(base) : 1 byte

```

The offset array at the end of the filter block allows efficient mapping from a data block offset to the corresponding filter.

## Properties Meta Block

This meta block contains a bunch of properties. The key is the name of the property. The value is the property.

The stats block is formatted as follows:

```

1. [prop1] (Each property is a key/value pair)
2. [prop2]
3. ...
4. [propN]

```

Properties are guaranteed to sort with no duplication.

By default, each table provides the following properties.

```

1. data size           // the total size of all data blocks.
2. index size         // the size of the index block.
3. filter size        // the size of the filter block.
4. raw key size       // the size of all keys before any processing.
5. raw value size     // the size of all value before any processing.
6. number of entries
7. number of data blocks

```

RocksDB also provides users the “callback” to collect their interested properties about this table. Please refer to [UserDefinedPropertiesCollector](#).

## Compression Dictionary Meta Block

This metablock contains the dictionary used to prime the compression library before compressing/decompressing each block. Its purpose is to address a fundamental problem with dynamic dictionary compression algorithms on small data blocks: the dictionary is built during a single pass over the block, so small data blocks always have small and thus ineffective dictionaries.

Our solution is to initialize the compression library with a dictionary built from data sampled from previously seen blocks. This dictionary is then stored in a file-level meta-block for use during decompression. The upper-bound on the size of this dictionary is configurable via `CompressionOptions::max_dict_bytes`. By default it is zero, i.e., the block is not generated or stored. Currently this feature is supported with `kZlibCompression`, `kLZ4Compression`, `kLZ4HCCompression`, and `kZSTDNotFinalCompression`.

More specifically, the compression dictionary is built only during compaction to the bottommost level, where the data is largest and most stable. To avoid iterating over input data multiple times, the dictionary includes samples from the subcompaction's first output file only. Then, the dictionary is applied to and stored in meta-blocks of all subsequent output files. Note the dictionary is not applied to or stored in the first file since its contents are not finalized until that file has been fully processed.

Currently the sampling is uniformly random and each sample is 64 bytes. We do not know in advance the size of the output file when selecting the sample offsets, so we assume it'll reach the maximum size, which is usually true since it's the first file in the subcompaction. In case the file is smaller, some sample intervals will refer to offsets beyond EOF, which just means the dictionary will be a bit smaller than `CompressionOptions::max_dict_bytes`.

## `Range Deletion` Meta Block

This metablock contains the range deletions in the file's key-range and seqnum-range. Range deletions cannot be inlined in the data blocks together with point data since the ranges would then not be binary searchable.

The block format is the standard key-value format. A range deletion is encoded as follows:

- User key: the range's begin key
- Sequence number: the sequence number at which the range deletion was inserted to the DB
- Value type: `kTypeRangeDeletion`
- Value: the range's end key

Range deletions are assigned sequence numbers when inserted using the same mechanism as non-range data types (puts, deletes, etc.). They also traverse through the LSM using the same flush/compaction mechanism as point data. They can be obsoleted (i.e., dropped) only during compaction to the bottommost level.

# Introduction

PlainTable is a RocksDB's SST file format optimized for low query latency on pure-memory or really low-latency media.

Advantages:

- An in-memory index is built to replace plain binary search with hash + binary search
- Bypassing block cache to avoid the overhead of block copy and LRU cache maintenance.
- Avoid any memory copy when querying (mmap)

Limitations:

- File size needs to be smaller than 31 bits integer.
- Data compression is not supported
- Delta encoding is not supported
- `Iterator.Prev()` is not supported
- Non-prefix-based `Seek()` is not supported
- Table loading is slower for building indexes
- Only support mmap mode.

We have plan to reduce some of the limitations.

# Usage

You can call two factory functions `NewPlainTableFactory()` or `NewTotalOrderPlainTableFactory()` in `table.h` to generate a table factory for plain table with your parameters and pass it to `Options.table_factory` or `ColumnFamilyOptions.table_factory`. You need to specify a prefix extractor if you use the former function. Examples:

```
1. options.table_factory.reset(NewPlainTableFactory());
2. options.prefix_extractor.reset(NewFixedPrefixTransform(8));
```

or

```
1. options.table_factory.reset(NewTotalOrderPlainTableFactory());
```

See comments of the two functions in `include/rocksdb/table.h` for explanation to the parameters.

`NewPlainTableFactory()` creates a plain table factory for plain tables with hash-based index using key prefixes. It is what `PlainTable` is optimized for.

While `NewTotalOrderPlainTableFactory()` doesn't require a prefix extractor and uses a totally binary index. This function is mainly to make `PlainTable` feature-complete. We haven't

yet highly optimized query performance in this case.

```

1. ### File Format
2. ##### Basic
3. <beginning_of_file>
4. [data row1]
5. [data row1]
6. [data row1]
7. ...
8. [data rowN]
9. [Property Block]
10. [Footer]                                (fixed size; starts at file_size - sizeof(Footer))
11. <end_of_file>
```

Format of property block and footer is the same as [BlockBasedTable format](#)

See [Row Format](#) for the format of each data row.

Two properties in property block are used to read data:

```

1. data_size: the end of data part of the file
2.
3. fixed_key_len: length of the key if all keys have the same length, 0 otherwise.
```

## Row Format

Each data row is encoded as:

```

1. <beginning of a row>
2. encoded key
3. length of value: varint32
4. value bytes
5. <end of a row>
```

See [Key Encoding](#) for format of encoded key.

## Key Encoding

There are two encoding types to the key: `kPlain` and `kPrefix`, which can be specified when creating the plain table factory.

### Plain Encoding

If fixed key length is given, the plain internal key is encoded.

If the fixed key length is not given, the key is variable length and will be encoded as

```
1. [length of key: varint32] + user key + internal bytes
```

See [Internal Bytes Encoding](#) for details of internal bytes.

### Prefix Encoding

kPrefix encoding type is a special delta encoding, in which if a row shows the same prefix (determined by prefix extractor given by the user) as the previous key, we can avoid to repeat the prefix part of the key.

In that case, there are three cases to encode a key (remember all keys are sorted so the keys sharing the same prefixes are together):

- the first key of a prefix, where a full key need to be written
- the second key of a prefix, where the prefix length needs to be recorded, as well as bytes other than prefix (to simplify, we call it suffix here)
- the third or later key of a prefix, where only the suffix part needs to be written.

We defined three flags for indicate full key, a prefix, as well as a suffix. For all the three cases, we need a size with it. They are encoded in this format:

The 8 bits of the first byte:

```

1. +-----+
2. | Type   | Size
3. +-----+

```

The first 2 bits indicate full key (00), prefix (01), or suffix (02). The last 6 bits are for size. If the size bits are not all 1, it means the size of the key. Otherwise, varint32 is written after this byte. This varint32 value + 0x3F (the value of all 1) will be the key size. In this way, shorter keys only need one byte.

Here are the formats for the three cases mentioned:

#### (1) Full Key

```

1. +-----+
2. | Full Key Flag + Size | Full User Key | Internal Bytes |
3. +-----+

```

#### (2) The second key of the prefix

```

1. +-----+
2. | Prefix Flag + Size | Suffix Flag + Size | Key Suffix | Internal Bytes |
3. +-----+

```

#### (3) The third and later key of the prefix:

```

1. +-----+
2. | Suffix Flag + Size | Key Suffix | Internal Bytes |

```

```
3. +-----+-----+-----+
```

See [Internal Bytes Encoding](#) for details of internal bytes for all the three cases.

With this format, without knowing the prefix, rows key only be seek using file offset of a full key. So if there are too many keys in a prefix, plain table builder might determine to rewrite the full key again, even if it is not the first key of the prefix, to make seek easier.

Here is an example, we for following keys (prefix and suffix are separated by spaces):

```
1. AAAA AAAB
2. AAAA AAABA
3. AAAA AAAC
4. AAABB AA
5. AAAC AAAB
```

Will be encoded like this:

```
1. FK 8 AAAAAAAB
2. PF 4 SF 5 AAABA
3. SF 4 AAAC
4. FK 7 AAABBAA
5. FK 8 AAACAAAB
```

(where FK means full key flag, PF means prefix flag and SF means suffix flag.)

### Internal Bytes Encoding

In both of Plain and Prefix encoding type, internal bytes of the internal key are encoded in the same way. In RocksDB, Internal bytes of a key include a row type (value, delete, merge, etc) and a sequence ID. Normally, a key is laid out in this format:

```
1. +----- . . . . . +-----+-----+-----+-----+
2. |       user key           |type|      sequence ID      |
3. +----- . . . . . +-----+-----+-----+-----+
```

where type takes one byte and sequence ID takes 7 bytes.

In Plain Table format, it is also the normal way one key can be optimized. Furthermore, we have an optimization to save some extra bytes for a common case: value type with sequence ID 0. In RocksDB, we have an optimization to fill sequence ID to be 0 for a key when we are sure there is no previous value for this key in the system (to be specifically, the first key of the last level for level-style compaction or last file in universal style) to enable better compression or encoding. In PlainTable, we use 1 byte "0x80" instead of 8 bytes for the internal bytes:

```

1. +-----+-----+
2. |       user key          | 0x80 |
3. +-----+-----+

```

## In-Memory Index Format

### Basic Idea

In-memory Index is built to be as compact as possible. On top level, the index is a hash table with each bucket to be either offset in the file or a binary search index. The binary search is needed in two cases:

- (1) Hash collisions: two or more prefixes are hashed to the same bucket.
- (2) Too many keys for one prefix: need to speed-up the look-up inside the prefix.

### Format

The index consists of two pieces of memory: an array for hash buckets, and a buffer containing the binary searchable indexes (call it “binary search buffer” below, and “file” as the SST file).

Key is hashed to buckets based on hash of its prefix (extracted using Options.prefix\_extractor).

```

1. +-----+
2. | Flag (1 bit) | Offset to binary search buffer or file (31 bits)   +
3. +-----+

```

If Flag = 0 and offset field equals to the offset of end of the data of the file, it means null - no data for this bucket; if the offset is smaller, it means there is only one prefix for the bucket, starting from that file offset. If Flag = 1, it means the offset is for binary search buffer. The format from that offset is shown below.

Starting from the offset of binary search buffer, a binary search index is encoded as following:

```

1. <begin>
2. number_of_records: varint32
3. record 1 file offset: fixedint32
4. record 2 file offset: fixedint32
5. ....
6. record N file offset: fixedint32
7. <end>

```

where N = number\_of\_records. The offsets are in ascending order.

The reason for only storing 31-bit offset and use 1-bit to identify whether a binary

search is needed is to make the index compact.

## An Example of Index

Let's assume here are the contents of a file:

```
1. +-----+ <= offset_0003_0000 = 0
2. | row (key: "0003 0000") |
3. +-----+ <= offset_0005_0000
4. | row (key: "0005 0000") |
5. +-----+
6. | row (key: "0005 0001") |
7. +-----+
8. | row (key: "0005 0002") |
9. +-----+
10. |
11. | .... |
12. |
13. +-----+
14. | row (key: "0005 000F") |
15. +-----+ <= offset_0005_0010
16. | row (key: "0005 0010") |
17. +-----+
18. |
19. | .... |
20. |
21. +-----+
22. | row (key: "0005 001F") |
23. +-----+ <= offset_0005_0020
24. | row (key: "0005 0020") |
25. +-----+
26. | row (key: "0005 0021") |
27. +-----+
28. | row (key: "0005 0022") |
29. +-----+ <= offset_0007_0000
30. | row (key: "0007 0000") |
31. +-----+
32. | row (key: "0007 0001") |
33. +-----+ <= offset_0008_0000
34. | row (key: "0008 0000") |
35. +-----+
36. | row (key: "0008 0001") |
37. +-----+
38. | row (key: "0008 0002") |
39. +-----+
40. |
41. | .... |
42. |
43. +-----+
44. | row (key: "0008 000F") |
45. +-----+ <= offset_0008_0010
46. | row (key: "0008 0010") |
```

```

47. +-----+ <== offset_end_data
48. |
49. | property block and footer |
50. |
51. +-----+

```

Let's assume in the example, we use 2 bytes fixed length prefix and in each prefix, rows are always incremented by 1.

Now we are building index for the file. By scanning the file, we know there are 4 distinct prefixes ("0003", "0005", "0007" and "0008") and assume we pick to use 5 hash buckets and based on the hash function, prefixes are hashed into the buckets:

```

1. bucket 0: 0005
2. bucket 1: empty
3. bucket 2: 0007
4. bucket 3: 0003 0008
5. bucket 4: empty

```

Bucket 2 doesn't need binary search since there is only one prefix in it ("0007") and it has only 2 (<16) rows.

Bucket 0 needs binary search because prefix 0005 has more than 16 rows.

Bucket 3 needs binary search because it contains more than one prefix.

We need to allocate binary search indexes for bucket 0 and 3. Here are the result:

```

1. +-----+ <== bs_offset_bucket_0
2. + 2 (in varint32)   |
3. +-----+-----+
4. + offset_0005_0000 (in fixedint32)   |
5. +-----+-----+
6. + offset_0005_0010 (in fixedint32)   |
7. +-----+-----+ <== bs_offset_bucket_3
8. + 3 (in varint32)   |
9. +-----+-----+
10. + offset_0003_0000 (in fixedint32)   |
11. +-----+-----+
12. + offset_0008_0000 (in fixedint32)   |
13. +-----+-----+
14. + offset_0008_0010 (in fixedint32)   |
15. +-----+

```

Then here are the data in hash buckets:

```

1. +-----+
2. | 1 | bs_offset_bucket_0 (31 bits)   | <== bucket 0
3. +-----+
4. | 0 | offset_end_data   (31 bits)   | <== bucket 1

```

```

5. +---+-----+
6. | 0 |    offset_0007_0000  (31 bits)      | <== bucket 2
7. +---+-----+
8. | 1 |    bs_offset_bucket_3 (31 bits)      | <== bucket 3
9. +---+-----+
10. | 0 |   offset_end_data    (31 bits)      | <== bucket 4
11. +---+-----+

```

## Index Look-up

To look up a key, first calculate prefix of the key using Options.prefix\_extractor, and find the bucket for the prefix. If the bucket has no record on it (Flag=0 and offset is the offset of data end in file), the key is not found. Otherwise,

If Flag=0, it means there is only one prefix for the bucket and there are not many keys for the prefix, so the offset field points to the file offset of the prefix. We just need to do linear search from there.

If Flag=1, a binary search is needed for this bucket. The binary search index can be retrieved from the offset field. After the binary search, do the linear search from the offset found by the binary search.

## Building the Index

When building indexes, scan the file. For each key, calculate its prefix, remember (hash value of the prefix, offset) information for the  $(16n+1)$ th row of each prefix ( $n=0,1,2\dots$ ), starting from the first one. 16 is the maximum number of rows that need to be checked in the linear search following the binary search. By increasing the number, we would save memory consumption for indexes but paying more costs for linear search. Decreasing the number vise versa. Based on the number of prefixes, determine an optimal bucket size. Allocate exact buckets and binary search buffer needed and fill in the indexes according to the bucket size.

## Bloom Filter

A bloom filter on prefixes can be configured for queries. User can config how many bits are allocated for every prefix. When doing the query (Seek() or Get()), bloom filter is checked and filter out non-existing prefixes before looking up the indexes.

## Future Plan

- May consider to materialize the index to be a part of the SST file.
- Add an option to remove the restriction of file size, by trading off memory consumption of indexes.
- May build extra more sparse indexes to enable general iterator seeking.

We introduce a new SST file format based on [Cuckoo Hashing](#) which is optimized for very high point lookup rates. Applications which don't use range scan but require very fast point lookups can use this new table format. See [here](#) for a detailed description of algorithm.

Advantages:

- For most lookups, only one memory access required per lookup.
- The database is smaller by 8 bytes per key after compaction as we don't store sequence number and value type in the key.

Limitations:

- Because the file format is hashing based, range scan is very slow
- Key and value lengths are fixed
- Does not support Merge Operator
- Does not support Snapshots
- File must be mmaped
- The last SST file in the database may have a utilization of only 50% in worst case.

We have plans to reduce some of the limitations in future.

## Usage

A new table factory can be created by calling `NewCuckooTableFactory()` function in `table.h`. See comments in `include/rocksdb/table.h` or [blogpost](#) for description of parameters.

Examples:

```
1. options.table_factory.reset(NewCuckooTableFactory());
```

or

```
1. options.table_factory.reset(NewCuckooTableFactory(
2.     0.9 /* hash_table_ratio */,
3.     100 /* max_search_depth */,
4.     5 /* cuckoo_block_size */);
```

## File Format

The Cuckoo SST file comprises of:

- Hash table containing key value pairs. Empty buckets are filled using a special key outside the key range in the table.
- Property Block containing Table properties
- Metadata and footer.

```

1.      <beginning_of_file>
2.      <beginning_of_hash_table>
3.          [key1 value1]
4.          [key2 value2]
5.          [key3 value3]
6.          ...
7.          [More key-values including empty buckets]
8.          ...
9.          [keyN valueN]
10.         <end_of_hash_table>
11.         [Property Block]
12.         [Footer]                                (fixed size; starts at file_size - sizeof(Footer))
13.         <end_of_file>

```

It must be noted that for optimizing hash computation, we fixed the number of buckets in a hash table to be a power of two. So, file sizes can only be a fixed set of values and hence the size of files may be considerably smaller than the values determined by `options.target_file_size_base` and `options.target_file_size_multiplier` parameters.

## Performance Results

We filled the db with 100M records in 8 files with key length of 8 bytes and value length of 4 bytes. We chose the default Cuckoo hash parameter: `hash_table_ratio=0.9, max_search_depth=100, cuckoo_block_size=5`.

Command to create and compact DB:

```

./db_bench --disable_seek_compaction=1 --statistics=1 --histogram=1 --cache_size=1048576 --bloom_bits=10 --
cache_numshardbits=4 --open_files=500000 --verify_checksum=1 --write_buffer_size=1073741824 --
max_write_buffer_number=2 --level0_slowdown_writes_trigger=16 --level0_stop_writes_trigger=24 --
delete_obsolete_files_period_micros=300000000 --max_grandparent_overlap_factor=10 --stats_per_interval=1 --
stats_interval=10000000 --compression_type=none --compression_ratio=1 --memtablerep=vector --sync=0 --
disable_data_sync=1 --key_size=8 --value_size=4 --num_levels=7 --threads=1 --mmap_read=1 --mmap_write=0 --
max_bytes_for_level_base=4294967296 --target_file_size_base=201327616 --level0_file_num_compaction_trigger=10 --
--max_background_compactions=20 --use_existing_db=0 --disable_wal=1 --db=/mnt/tmp/cuckoo/2M100 --
use_cuckoo_table=1 --use_uint64_comparator --benchmarks=fillunique,random,compact --cuckoo_hash_ratio=0.9 --
1. num=100000000

```

Random Read:

```

1. readrandom : 0.371 microseconds/op 2698931 ops/sec; (809679999 of 809679999 found)
./db_bench --stats_interval=10000000 --open_files=-1 --key_size=8 --value_size=4 --num_levels=7 --threads=1 --
-mmap_read=1 --mmap_write=0 --use_existing_db=1 --disable_wal=1 --db=/mnt/tmp/cuckoo/2M100 --
2. use_cuckoo_table=1 --benchmarks=readrandom --num=100000000 --readonly --use_uint64_comparator --duration=300

```

Multi Random Read:

```

1. multireadrandom : 0.278 microseconds/op 3601345 ops/sec; (1080449950 of 1080449950 found)

```

```
./db_bench --stats_interval=10000000 --open_files=-1 --key_size=8 --value_size=4 --num_levels=7 --threads=1  
--mmap_read=1 --mmap_write=0 --use_existing_db=1 --disable_wal=1 --db=/mnt/tmp/cuckoo/2M100 --  
use_cuckoo_table=1 --benchmarks=multireadrandom --num=10000000 --duration=300 --readonly --batch_size=50 --  
2. use_uint64_comparator
```

Note that the MultiGet experiment contains an implementation of MultiGet() in readonly mode which is not yet checked in. The performance is likely to improve further as we submit more optimizations that we have identified.

An `index` block contains one entry per data block, where the key is a string `>=` last key in that data block and before the first key in the successive data block. The value is the `BlockHandle` (file offset and length) for the data block.

## Partitioned Index

If `kTwoLevelIndexSearch` is used as `IndexType`, the `index` block is a 2nd level index on index partitions, i.e., each entry points to another `index` block that contains one entry per data block. In this case, the format will be

1. [index block - 1st level]
2. [index block - 1st level]
3. ...
4. [index block - 1st level]
5. [index block - 2nd level]

## Key in index blocks

As described above, the key stored for a block is between the last key of the block and the first key of the next block. There are usually a range of potential keys met this condition. Choosing a smaller one can reduce the index size. If

`BlockBasedTableOptions.index_shortening` is set to `kShortenSeparators` or `kShortenSeparatorsAndSuccessor`, the last key of the block, and the first key of the next block will be passed to `Comparator::FindShortestSeparator()` to find out the shortest separator key. This function is implemented in builtin byte-wise and reverse byte-wise comparators. User comparators need to implement the function to take advantage this feature.

Similarly the index key for the last block is determined by `Comparator::FindShortSuccessor()`, which provides any key that is greater or equal to the last key of the last block. Users also need to implement this function for customized comparator to take advantage of the memory saving feature.

The default value for the option is `kShortenSeparators`, which shortens index key for all blocks but the last one. It's because that the last key has minimal impacts to index size while can have positive impact of preventing the last data blocks to be read.

Value `kNoShortening` can also be used together with `index_type = kBinarySearchWithFirstKey` to prevent reading some blocks using a special function, which is explained below.

## Individual Index Block

Up to RocksDB version 5.14, `BlockBasedTableOptions::format_version \=2`, the format of index and data blocks are the same, where the index blocks use same key format of `< user_key , seq >` but special values, `< offset , size >`, that point to data blocks. Different from data blocks, the option controlling restart block size is `BlockBasedTableOptions.index_block_restart_interval`, rather than

`BlockBasedTableOptions.block_restart_interval`. The default value is 1, rather than 16 for data blocks. So the default is relatively memory costly. Setting the value to 8 or 16 can usually shrink index block size by half, but the CPU overhead might increase based on workloads. `format_version=3,4` further optimized size, yet forward-incompatible format for index blocks.

- `format_version \=3` (Since RocksDB 5.15): In most of the cases the sequence number `seq` is not necessary for keys in the index blocks. In such cases, this `format_version` skips encoding the sequence number and sets `index_key_is_user_key` in `TableProperties`, which is used by the reader to know how to decode the index block.
- `format_version \=4` (Since RocksDB 5.16): Changes the format of index blocks by delta encoding the index values, which are the block handles. This saves the encoding of `BlockHandle::offset` of the non-head index entries in each restart interval. If used, `TableProperties::index_value_is_delta_encoded` is set, which is used by the reader to know how to decode the index block. The format of each key is (`shared_size`, `non_shared_size`, `shared`, `non_shared`). The format of each value, i.e., block handle, is (`offset`, `size`) whenever the `shared_size` is 0, which included the first entry in each restart point. Otherwise the format is `delta-size = block handle size - size of last block handle`.

The index format in `format_version=4` would be as follows:

```

1. restart_point 0: k, v (off, sz), k, v (delta-sz), ..., k, v (delta-sz)
2. restart_point 1: k, v (off, sz), k, v (delta-sz), ..., k, v (delta-sz)
3. ...
4. restart_point n-1: k, v (off, sz), k, v (delta-sz), ..., k, v (delta-sz)
5. where, k is key, v is value, and its encoding is in parenthesis.

```

The format applies to the case when `index_type != kBinarySearchWithFirstKey`. The `kBinarySearchWithFirstKey` case is described in the next section.

## index\_type == kBinarySearchWithFirstKey

The feature of `index_type == kBinarySearchWithFirstKey` is to allow RocksDB to see first key of a data block without reading it from the disk. With this feature, RocksDB knows the key of the first block, so it doesn't have to read the data block immediately. Only when users call `Iterator::value()`, the block can be loaded. This can effectively prevent I/O and other overhead for some special workloads. For example, when data block usually occupy the whole data block, a `Get()` can skip data block reads for files not containing the keys.

If this option is used, for each entry in the index block, following the `BlockHandle` (`offset, size`) part, the first key of the block is stored, in the form of length prefixed string. For example, `version_format = 5` will take the format as:

```

1. restart_point 0:

```

```
2.     k (block_key), v (block_offset, block_size, size_of_first_key, first_key)
3.     k (block_key), v (delta_size, size_of_first_key, first_key)
4.     k (block_key), v (delta_size, size_of_first_key, first_key)
5. restart_point  1:
6.     k (block_key), v (block_offset, block_size, size_of_first_key, first_key)
7.     k (block_key), v (delta_size, size_of_first_key, first_key)
8.     k (block_key), v (delta_size, size_of_first_key, first_key)
9. ...
```

It's similar to other format versions and restart block size.

## What is a Bloom Filter?

For any arbitrary set of keys, an algorithm may be applied to create a bit array called a Bloom filter. Given an arbitrary key, this bit array may be used to determine if the key *may exist* or *definitely does not exist* in the key set. For a more detailed explanation of how Bloom filters work, see this [Wikipedia article](#).

In RocksDB, when the filter policy is set, every newly created SST file will contain a Bloom filter, which is used to determine if the file may contain the key we're looking for. The filter is essentially a bit array. Multiple hash functions are applied to the given key, each specifying a bit in the array that will be set to 1. At read time also the same hash functions are applied on the search key, the bits are checked, i.e., probe, and the key definitely does not exist if at least one of the probes return 0.

The example of setting up a bloom filter:

```

1. rocksdb::BlockBasedTableOptions table_options;
2. table_options.filter_policy.reset(rocksdb::NewBloomFilterPolicy(10, false));
3. my_cf_options.table_factory.reset(
4.     rocksdb::NewBlockBasedTableFactory(table_options));

```

## Life Cycle

When configured in RocksDB, each SST file is created with a Bloom filter, embedded in the SST file itself. Bloom filters are generally constructed for files in all levels in the same way, except that the last level can be skipped by setting

`optimize_filters_for_hits`, and overriding `FilterPolicy::GetBuilderWithContext` offers customizability.

Because of different sizes and other Bloom filter limitations, SST Bloom filters are not generally compatible with each other for optimized AND/OR composition. Even when we combine two SST files, a new Bloom filter is created from scratch with the keys of the new file, in part to ensure predictable bits/key and, thus, false positive rate.

When we open an SST file, the corresponding Bloom filter is also opened and loaded in memory. When the SST file is closed, the Bloom filter is removed from memory. To otherwise cache the Bloom filter in block cache, use:

```
BlockBasedTableOptions::cache_index_and_filter_blocks=true, .
```

## Block-based Bloom Filter (old format)

With this API, a Bloom filter could only be built if all keys fit in memory. On the other hand, sharding keys across bloom filters does not affect the overall false positive rate if each filter is sized for the number of keys added. Therefore, to alleviate the memory pressure when creating the SST file, in the old format a separate bloom filter is created per each 2KB block of key values. Details for the format can

be found [here](#). At the end an array of the offsets of individual bloom blocks is stored in SST file.

At read time, the offset of the block that might contain the key/value is obtained from the SST index. Based on the offset the corresponding bloom filter is then loaded. If the filter suggests that the key might exist, then it searches the actual data block for the key.

## Full Filters (new format)

The individual filter blocks in the old format are not cache aligned and could result into a lot of cache misses during lookup. Moreover although negative responses (i.e., key does not exist) of the filter saves the search from the data block, the index is already loaded and looked into. The new format, full filter, addresses these issues by creating one filter for the entire SST file. The tradeoff is that more memory is required to cache a hash of every key in the file (versus only keys of a 2KB block in the old format).

Full filter limits the probe bits for a key to be all within the same CPU cache line. This ensures fast lookups and maximizes CPU cache effectiveness by limiting the CPU cache misses to one per key (per filter). Note that this is essentially sharding the bloom space and has only a small effect on the false positive rate as long as there are many keys. Refer to “The Math” section below for more details.

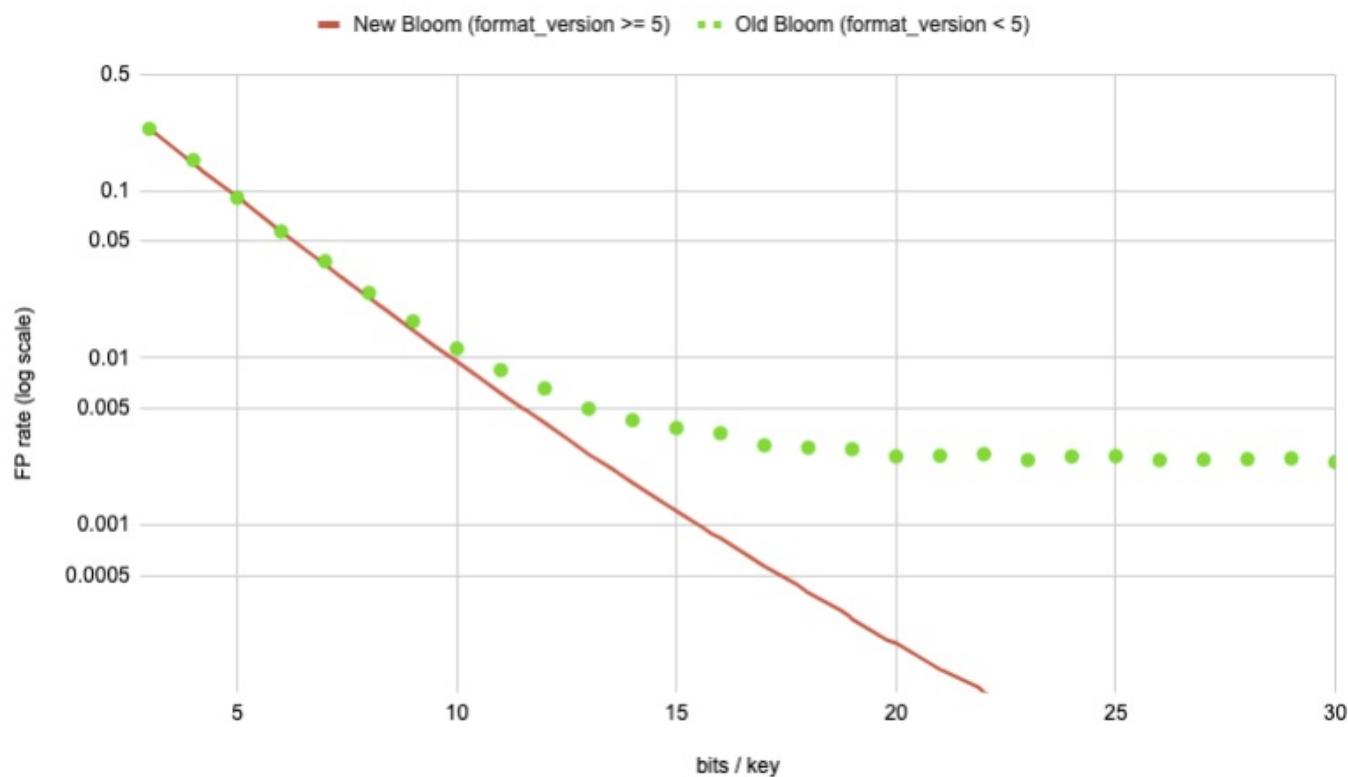
At read time, RocksDB uses the same format that was used when creating the SST file. Users can specify the format for the newly created SST files by setting `filter_policy` in options. The helper function `NewBloomFilterPolicy` can be used to create both old block-based and new full filter (the default).

```
1. extern const FilterPolicy* NewBloomFilterPolicy(
2.     double bits_per_key, bool use_block_based_builder = false);
3. }
```

The first underlying Bloom filter implementation used for full filters (and partitioned filters) had some flaws, and a new implementation is used with `format_version \=5` (version 6.6) and later. First, the original full filter could not get an FP rate better than about 0.1%, even at 100 bits/key. The new implementation is below 0.1% FP rate with only 16 bits/key. Second, possibly unlikely, the original full filter would have degraded FP rates with millions of keys in a single filter, because of inherent limitations of 32-bit hashing. The new implementation easily scales to many billions of keys in a single filter, thanks to a 64-bit hash. Aside from extra intermediate data during construction (64 bits per key rather than 32), the new implementation is generally faster also.

The following graph can be used to pick a `bits_per_key` setting to achieve a certain false positive rate in the Bloom filter, old or new implementation. You might also consult this for migrating to `format_version=5` if you wish to save filter space and

keep the same FP rate. (Note that FP rates are not percentages here. For example, 0.001 means 1 in 1000 queries of keys not in the SST file will falsely return true, leading to extra work by RocksDB to determine the key is not actually in the SST.)



## Prefix vs. whole key

By default a hash of every whole key is added to the bloom filter. This can be disabled by setting `BlockBasedTableOptions::whole_key_filtering` to false. When `Options.prefix_extractor` is set, a hash of the prefix is also added to the bloom. Since there are less unique prefixes than unique whole keys, storing only the prefixes in bloom will result into smaller blooms with the down side of having larger false positive rate. Moreover the prefix blooms can be optionally (using `check_filter` when creating the iterator) also used during `::Seek` and `::SeekForPrev` whereas the whole key blooms are only used for point lookups.

## Statistic

Here are the statistics that can be used to gain insight of how well your full bloom filter settings are performing in production:

The following stats are updated after each `::Seek` and `::SeekForPrev` if prefix is enabled and `check_filter` is set.

- `rocksdb.bloom.filter.prefix.checked` : `seek_negatives + seek_positives`
- `rocksdb.bloom.filter.prefix.useful` : `seek_negatives`

The following stats are updated after each point lookup. If `whole_key_filtering` is set, this is the result of checking the bloom of the whole key, otherwise this is the

result of checking the bloom of the prefix.

- `rocksdb.bloom.filter.useful` : [true] negatives
- `rocksdb.bloom.filter.full.positive` : positives
- `rocksdb.bloom.filter.full.true.positive` : true positives

So the observed false positive rate of point lookups is computed by  $(\text{positives} - \text{true positives}) / (\text{positives} - \text{true positives} + \text{true negatives})$ , which is  $(\text{number of queries returning positive on keys or prefixes not in the SST}) / (\text{number of queries on keys or prefixes not in the SST})$ .

Pay attention to these confusing scenarios:

1. If both `whole_key_filtering` and `prefix` are set, `prefix` are not checked during point lookups.
2. If only the `prefix` is set, the total number of times `prefix bloom` is checked is the sum of the stats of point lookup and seeks. Due to absence of true positive stats in seeks, we then cannot have the total false positive rate: only that of point lookups.

## Customize your own FilterPolicy

`FilterPolicy` (`include/rocksdb/filter_policy.h`) can be extended to define custom filters. The two main functions to implement are:

- ```
1. FilterBitsBuilder* GetFilterBitsBuilder()
2. FilterBitsReader* GetFilterBitsReader(const Slice& contents)
```

In this way, the new filter policy would function as a factory for `FilterBitsBuilder` and `FilterBitsReader`. `FilterBitsBuilder` provides interface for key storage and filter generation and `FilterBitsReader` provides interface to check if a key may exist in filter.

A newer alternative to `GetFilterBitsBuilder()` is also available (override only one of the two):

- ```
1. FilterBitsBuilder* GetBuilderWithContext(const FilterBuildingContext&)
```

This variant allows custom filter configuration based on contextual information such as table level, compaction style, and column family. See `LevelAndStyleCustomFilterPolicy` for an example selecting between configurations of built-in filters.

Notice: This builder/reader interface only works for full and partitioned filters (new format).

## Partitioned Bloom Filter

Partitioned filters use the same filter block format as full filters, but use many filter blocks per SST file partitioned by key range. This feature increases the average CPU cost of an SST Bloom query because of key range partitioning, but (with `cache_index_and_filter_blocks=true`) greatly reduces worst-case data loaded into the block cache for a single small-data op. This can mitigate and smooth a block cache thrashing “cliff” and improve tail latencies. Read [here](#).

## The math

Refer e.g. to [the wikipedia article](#) for standard false positive rate of a Bloom filter.

The CPU cache-friendly Bloom filter variant is presented and analyzed in [this 2007 paper](#)(page 4, formula 3). In short, the Bloom filter is sharded into  $s$  shards, each of size  $m/s$ . If  $s$  were much smaller than  $n$  (number of keys), each shard would have approximately  $n/s$  keys mapping to it, and the false positive rate is like a standard Bloom filter using  $m/s$  for  $m$  and  $n/s$  for  $n$ . This yields essentially the same false positive rate as for a standard Bloom filter with the original  $m$  and  $n$ .

For CPU-cache locality of probes,  $s$  is within a couple orders of magnitude of  $n$ , which leads to noticeable variance in the number of keys mapped to each shard. (This is the phenomenon behind “clustering” in hashing.) This matters because the false positive rate is non-linear in the number of keys added. Even though the median false positive rate of a shard is the same as a standard Bloom filter, what matters is the mean false positive rate, or the expected value of the false positive rate of a shard.

This can be understood using the [Poisson distribution \(as an approximation of a binomial distribution\)](#), using parameter  $n/s$ , the expected number of keys mapping to each shard. You can compute the expected/effective false positive rate using a summation based on the pdf of a Poisson distribution (in 2007 paper referenced above), or you can use an approximation I have found to be very good:

- The expected false positive rate is close to the average of the false positive rate for one standard deviation above and below the expected number of keys per shard.

The standard deviation is the square root of that  $n/s$  parameter. In a standard configuration,  $m/s = 512$  and  $m/n = 10$ , so  $n/s = 51.2$  and  $\sqrt{n/s} = 7.16$ . With  $k=6$ , we use the standard Bloom filter false positive rate formula, substituting  $51.2 +/- 7.16$  for  $n$  and  $512$  for  $m$ , yielding 0.43% and 1.48%, which averages to 0.95% for the CPU cache local Bloom filter. The same Bloom filter without cache locality has false positive rate 0.84%.

This modest increase in false positive rate, here a factor of 1.13 higher, is considered quite acceptable for the speed boost, especially in environments heavily utilizing many cores.

RocksDB does a binary search when performing point lookup the keys in data blocks. However, in order to find the right location where the key may reside, multiple key parsing and comparison are needed. Each binary search branching triggers CPU cache miss, causing much CPU utilization. We have seen that this binary search takes up considerable CPU in production use-cases.

A hash index is designed and implemented in RocksDB data blocks to improve the CPU efficiency of point lookup. Benchmarks with `db_bench` show the CPU utilization of one of the main functions in the point lookup code path, `DataBlockIter::Seek()`, is reduced by 21.8%, and the overall RocksDB throughput is increased by 10% under purely cached workloads, at an overhead of 4.6% more space.

## How to use it

Two new options are added as part of this feature: `BlockBasedTableOptions::data_block_index_type` and `BlockBasedTableOptions::data_block_hash_table_util_ratio`.

The hash index is disabled by default unless `BlockBasedTableOptions::data_block_index_type` is set to `data_block_index_type = kDataBlockBinaryAndHash`. The hash table utilization ratio is adjustable using `BlockBasedTableOptions::data_block_hash_table_util_ratio`, which is valid only if `data_block_index_type = kDataBlockBinaryAndHash`.

```

1. // the definitions can be found in include/rocksdb/table.h
2.
3. // The index type that will be used for the data block.
4. enum DataBlockIndexType : char {
5.     kDataBlockBinarySearch = 0, // traditional block type
6.     kDataBlockBinaryAndHash = 1, // additional hash index
7. };
8.
9. DataBlockIndexType data_block_index_type = kDataBlockBinarySearch;
10.
11. // #entries/#buckets. It is valid only when data_block_index_type is
12. // kDataBlockBinaryAndHash.
13. double data_block_hash_table_util_ratio = 0.75;
```

## Things that Need Attention

### Customized Comparator

Hash index will hash different keys (keys with different content, or byte sequence) into different hash values. This assumes the comparator will not treat different keys as equal if they have different content.

The default bytewise comparator orders the keys in alphabetical order and works well with hash index, as different keys will never be regarded as equal. However, some specially crafted comparators will do. For example, say, a `StringToIntComparator` can convert a string into an integer, and use the integer to perform the comparison. Key

string "16" and "0x10" is equal to each other as seen by this `StringToIntComparator`, but they probably hash to different value. Later queries to one form of the key will not be able to find the existing key been stored in the other format.

We add a new function member to the comparator interface:

```
1. virtual bool CanKeysWithDifferentByteContentsBeEqual() const { return true; }
```

Every comparator implementation should override this function and specify the behavior of the comparator. If a comparator can regard different keys equal, the function returns true, and as a result the hash index feature will not be enabled, and vice versa.

NOTE: to use the hash index feature, one should 1) have a comparator that can never treat different keys as equal; and 2) override the `CanKeysWithDifferentByteContentsBeEqual()` function to return `false`, so the hash index can be enabled.

### Util Ratio's Impact on Data Block Cache

Adding the hash index to the end of the data block essentially takes up the data block cache space, making the effective data block cache size smaller and increasing the data block cache miss ratio. Therefore, a very small util ratio will result in a large data block cache miss ratio, and the extra I/O may drag down the throughput gain achieved by the hash index lookup. Besides, when compression is enabled, cache miss also incurs data block decompression, which is CPU-consuming. Therefore the CPU may even increase if using a too small util ratio. The best util ratio depends on workloads, cache to data ratio, disk bandwidth/latency etc. In our experiment, we found util ratio = 0.5 ~ 1 is a good range to explore that brings both CPU and throughput gains.

## Limitations

As we use `uint8_t` to store binary seek index, i.e. restart interval index, the total number of restart intervals cannot be more than 253 (we reserved 255 and 254 as special flags). For blocks having a larger number of restart intervals, the hash index will not be created and the point lookup will be done by traditional binary seek.

Data block hash index only supports point lookup. We do not support range lookup. Range lookup request will fall back to BinarySeek.

RocksDB supports many types of records, such as `Put`, `Delete`, `Merge`, etc (visit [here](#) for more information). Currently we only support `Put` and `Delete`, but not `Merge`. Internally we have a limited set of supported record types:

```
1. kPutRecord,      <==== supported
2. kDeleteRecord,  <==== supported
3. kSingleDeleteRecord, <==== supported
4. kTypeBlobIndex, <==== supported
```

For records not supported, the searching process will fall back to the traditional binary seek.

RocksDB provides a list of options for users to hint how I/Os should be executed.

## Control Write I/O

---

### Range Sync

RocksDB's data files are usually generated in an appending way. File system may choose to buffer the write until the dirty pages hit a threshold and write out all of those pages all together. This can create a burst of write I/O and cause the online I/Os to wait too long and cause long query latency. Rather, you can ask RocksDB to periodically hint OS to write out outstanding dirty pages by setting

`options.bytes_per_sync` for SST files and `options.wal_bytes_per_sync` for WAL files. Underlying it calls `sync_file_range()` on Linux every time a file is appended for such size. The most recent pages are not included in the range sync.

### Rate Limiter

You can control the total rate RocksDB writes to data files through `options.rate_limiter`, in order to reserve enough I/O bandwidth to online queries. See [Rate Limiter](#) for details.

### Write Max Buffer

When appending a file, RocksDB has internal buffering of files before writing to the file system, unless an explicit `fsync` is needed. The max size of this buffer can be controlled by `options.writable_file_max_buffer_size`. Tuning this parameter is more critical in [[Direct IO] mode or to a file system without page cache. With non-direct I/O mode, enlarging this buffer only reduces number of `write()` system calls and is unlikely to change the I/O behavior, so unless this is what you want, it may be desirable to keep the default value 0 to save the memory.

### File Deletion

Deletion of obsolete DB files can be rate limited by configuring the delete scheduler. This is especially useful on flash devices to limit read latency spikes due to a burst of deletions. See [Delete Scheduler](#) for details.

## Control Read I/O

---

### fadvise

While opening an SST file for reads, users can decide whether RocksDB will call `fadvise` with `FADV_RANDOM`, by setting `options.advise_random_on_open = true` (default). If the value is `false`, no `fadvise` will be called while opening a file. Setting the option to

be `true` usually works well if the dominating queries are either `Get()` or iterating a very short range, because read-ahead is not helpful in these cases anyway. Otherwise, `options.advise_random_on_open = false` can usually improve performance to hint the file system to do underlying read-ahead.

Unfortunately, there isn't a good setting for mixed workload. There is an ongoing project to address this, by doing read-ahead for iterators inside RocksDB.

## Compaction inputs

Compaction inputs are special. They are long sequential reads, so applying the same `fadvise` hint as user reads is usually not optimal. Also, usually, compaction input files are often going to be deleted soon, though there is no guarantee. RocksDB provides multiple ways to deal with that:

### `fadvise` hint

RocksDB will call `fadvise` to any compaction input file according to

`options.access_hint_on_compaction_start`. This can override the `fadvise` random setting since a file is picked as a compaction input.

### Use a different file descriptor for compaction inputs

If `options.new_table_reader_for_compaction_inputs = true`, RocksDB will use different file descriptors for compaction inputs. This can avoid the mixture of `fadvise` setting for normal data files and compaction inputs. The limitation of the setting is that, RocksDB does not just create a new file descriptor, but read index, filter and other meta blocks again and store them in memory, so that it takes extra I/O and use more memory.

### readahead for compaction inputs

You can do its own readahead following `options.compaction_readahead_size` if it is not 0. `options.new_table_reader_for_compaction_inputs` is automatically switched to `true` if the option is set. This setting can allow users to keep `options.access_hint_on_compaction_start` to `NONE`.

It is critical to set the option if `Direct IO` is on or the file system doesn't support readahead.

## Direct I/O

---

Rather than control the I/O through file system hints shown above, you can enable direct I/O in RocksDB to allow RocksDB to directly control I/O, using option `use_direct_reads` and/or `use_direct_io_for_flush_and_compaction`. If direct I/O is enabled, some or all of the options introduced above will not be applicable. See more details in [Direct IO](#).

# Memory Mapping

`options.allow_mmap_reads` and `options.allow_mmap_writes` make RocksDB mmap the whole data file while doing read or write, respectively. The benefit of the approach is to reduce the file system calls doing `pread()` and `write()`, and in many cases, reduce the memory copying too. `options.allow_mmap_reads` can usually significantly improve performance if the DB is run on ramfs. They can be used on file systems backed by block device too. However, based on our previous experience, file systems aren't usually doing a perfect job maintaining this kind of memory mapping, and some times cause slow queries. In this case, we advise you try out this option only when necessary and with caution.

# Avoid Blocking IO

Cleanup on `Iterator` destruction and cleanup on column family destruction by default will try to delete obsolete files in the context of the thread calling the destructor, subject to deletion rate limits. This can result in an unexpected long latency in completing the operation. To avoid the long latency and defer the deletion of obsolete files to background threads, the following options are provided -

- `ReadOptions::background_purge_on_iterator_cleanup` - This option, when set in the call to `DB::NewIterator()`, will schedule the deletion of obsolete files in a background thread on iterator destruction.
- `DBOptions::avoid_unnecessary_blocking_io` - This option is a DB wide option. When set, both iterator destructor and `ColumnFamilyHandle` destructors will schedule obsolete file deletion in a background thread.

When using RocksDB, users maybe want to throttle the maximum write speed within a certain limit for lots of reasons. For example, flash writes cause terrible spikes in read latency if they exceed a certain threshold. Since you've been reading this site, I believe you already know why you need a rate limiter. Actually, RocksDB contains a native `RateLimiter` which should be adequate for most use cases.

## How to use

Create a `RateLimiter` object by calling `NewGenericRateLimiter` , which can be created separately for each RocksDB instance or by shared among RocksDB instances to control the aggregated write rate of flush and compaction.

```
1. RateLimiter* rate_limiter = NewGenericRateLimiter(
2.     rate_bytes_per_sec /* int64_t */,
3.     refill_period_us /* int64_t */,
4.     fairness /* int32_t */);
```

Params:

- `rate_bytes_per_sec` : this is the only parameter you want to set most of the time. It controls the total write rate of compaction and flush in bytes per second. Currently, RocksDB does not enforce rate limit for anything other than flush and compaction, e.g. write to WAL
- `refill_period_us` : this controls how often tokens are refilled. For example, when `rate_bytes_per_sec` is set to 10MB/s and `refill_period_us` is set to 100ms, then 1MB is refilled every 100ms internally. Larger value can lead to burst writes while smaller value introduces more CPU overhead. The default value 100,000 should work for most cases.
- `fairness` : `RateLimiter` accepts high-pri requests and low-pri requests. A low-pri request is usually blocked in favor of hi-pri request. Currently, RocksDB assigns low-pri to request from compaction and high-pri to request from flush. Low-pri requests can get blocked if flush requests come in continuously. This fairness parameter grants low-pri requests permission by 1/fairness chance even though high-pri requests exist to avoid starvation. You should be good by leaving it at default 10.

Although tokens are refilled with a certain interval set by `refill_period_us` , the maximum bytes that can be granted in a single burst have to be bounded since we are not happy to see that tokens are accumulated for a long time and then consumed by a single burst request which definitely does not agree with our intention.

`GetSingleBurstBytes()` returns this upper bound of tokens.

Then each time token should be requested before writes happen. If this request can not be satisfied now, the call will be blocked until tokens get refilled to fulfill the request. For example,

```
1. // block if tokens are not enough
```

```
2. rate_limiter->Request(1024 /* bytes */, rocksdb::Env::IO_HIGH);
3. Status s = db->Flush();
```

Users could also dynamically change rate limiter's bytes per second with `SetBytesPerSecond()` when they need. see [include/rocksdb/rate\\_limiter.h](#) for more API details.

## Customization

For the users whose requirements are beyond the functions provided by RocksDB native Ratelimiter, they can implement their own Ratelimiter by extending [include/rocksdb/rate\\_limiter.h](#)

The `SstFileManager` class manages the physical SST file disk space utilization and deletion. It can be configured by the user in `Options::sst_file_manager`. This class, even though declared in a public header file, is not extensible and the only way to allocate a new instance is by calling `NewSstFileManager()`. One `SstFileManager` object can be shared by multiple DB instances. The limits configured in the `SstFileManager` would be shared by all the DBs.

`SstFileManager` provides various options to control and limit the disk space utilization of SST files in the DB. See [Managing Disk Space Utilization](#) for more details.

The other key functionality provided by `SstFileManager` is slow deletion (a.k.a rate limited deletion). On SSDs, enabling slow deletion helps reduce the rate of TRIM commands to the SSD, thereby improving read/write latencies. See [Slow Deletion](#) for more details.

# Introduction

Direct I/O is a system-wide feature that supports direct reads/writes from/to storage device to/from user memory space bypassing system page cache. Buffered I/O is usually the default I/O mode enabled by most operating systems.

## Why do we need it?

With buffered I/O, the data is copied twice between storage and memory because of the page cache as the proxy between the two. In most cases, the introduction of page cache could achieve better performance. But for self-caching applications such as RocksDB, the application itself should have a better knowledge of the logical semantics of the data than OS, which provides a chance that the applications could implement more efficient replacement algorithm for cache with any application-defined data block as a unit by leveraging their knowledge of data semantics. On the other hand, in some situations, we want some data to opt-out of system cache. At this time, direct I/O would be a better choice.

## Implementation

The way to enable direct I/O depends on the operating system and the support of direct I/O depends on the file system. Before using this feature, please check whether the file system supports direct I/O. RocksDB has dealt with these OS-dependent complications for you, but we are glad to share some implementation details here.

### 1. File Open

For LINUX, the `O_DIRECT` flag has to be included. For Mac OSX, `O_DIRECT` is not available. Instead, `fcntl(fd, F_NOCACHE, 1)` looks to be the canonical solution where `fd` is the file descriptor of the file. For Windows, there is a flag called `FILE_FLAG_NO_BUFFERING` as the counterpart in Windows of `O_DIRECT`.

### 2. File R/W

Direct I/O requires file R/W to be aligned, which means, the position indicator (offset), #bytes and the buffer address must be aligned to the *logical sector size* of the underlying storage device. So the position indicator should and the buffer pointer must be aligned on a *logical sector size* boundary and the number of bytes to be read or written must be in multiples of the *logical sector size*. RocksDB implements all the alignment logic inside `FileReader/FileWriter`, one layer higher abstraction on top of File classes to make the alignment ignorant to the OS. Thus, different OSs could have their own implementations of File Classes.

## API

It is easy to use Direct I/O as two new options are provided in `options.h`:

```

1. // Enable direct I/O mode for read/write
2. // they may or may not improve performance depending on the use case
3. //
4. // Files will be opened in "direct I/O" mode
5. // which means that data r/w from the disk will not be cached or
6. // buffered. The hardware buffer of the devices may however still
7. // be used. Memory mapped files are not impacted by these parameters.
8.
9. // Use O_DIRECT for user and compaction reads.
10. // When true, we also force new_table_reader_for_compaction_inputs to true.
11. // Default: false
12. // Not supported in ROCKSDB_LITE mode!
13. bool use_direct_reads = false;
14.
15. // Use O_DIRECT for writes in background flush and compactions.
16. // Default: false
17. // Not supported in ROCKSDB_LITE mode!
18. bool use_direct_io_for_flush_and_compaction = false;
```

The code is self-explanatory.

You may also need other options to optimize direct I/O performance.

```

1. // options.h
2. // Option to enable readahead in compaction
3. // If not set, it will be set to 2MB internally
4. size_t compaction_readahead_size = 2 * 1024 * 1024; // recommend at least 2MB
5. // Option to tune write buffer for direct writes
6. size_t writable_file_max_buffer_size = 1024 * 1024; // 1MB by default
```

```

1. // DEPRECATED!
2. // table.h
3. // If true, block will not be explicitly flushed to disk during building
4. // a SstTable. Instead, buffer in WritableFileWriter will take
5. // care of the flushing when it is full.
6. // This option is deprecated and always be true
7. bbto.skip_table_builder_flush = true;
```

Recent releases have these options automatically set if direct I/O is enabled.

## Notes

- `allow_mmap_reads` cannot be used with `use_direct_reads` or `use_direct_io_for_flush_and_compaction`. `allow_mmap_writes` cannot be used with `use_direct_io_for_flush_and_compaction`, i.e., they cannot be set to true at the same time.
- `use_direct_io_for_flush_and_compaction` and `use_direct_reads` will only be applied to SST file I/O but not WAL I/O or MANIFEST I/O. Direct I/O for WAL and Manifest files is not

supported yet.

3. After enable direct I/O, compaction writes will no longer be in the OS page cache, so first read will do real IO. Some users may know RocksDB has a feature called compressed block cache which is supposed to be able to replace page cache with direct I/O enabled. But please read the following comments before enable it:
    - Fragmentation. RocksDB's compressed block is not aligned to page size. A compressed block resides in malloc'ed memory in RocksDB's compressed block cache. It usually means a fragmentation in memory usage. OS page cache does slightly better, since it caches the whole physical page. If some continuous blocks are all hot, OS page cache uses less memory to cache them.
    - OS page cache provides read ahead. By default this is turned off in RocksDB but users can choose turn it on. This is going to be useful in range-loop dominated workloads. RocksDB compressed cache doesn't have anything to match the functionality.
    - Possible bugs. The RocksDB compressed block cache code has never been used before. We did see external users reporting bugs to it, but we never took more steps improve this component.
1. [Automatic Readahead](#) is enabled for Iterators in Direct IO mode as well. With this, long-range and full-table scans benchmarks in Sysbench (via a MyRocks build) match that of the Buffered IO mode.
  2. It is advisable to turn on mid-point insertion strategy for the Block Cache if your workload is a mix of point and range queries, by setting`LRUCacheOptions.high_pri_pool_ratio = 0.5`. (Note that this depends on`BlockBasedTableOptions.cache_index_and_filter_blocks` and`cache_index_and_filter_blocks_with_high_priority`as well).

# What is compressed?

In each SST file, data blocks and index blocks can be compressed individually. Users can specify what compression types to use. Filter blocks are not compressed.

## Configuration

Compression configuration is per column family.

Use `options.compression` to specify the compression to use. By default it is Snappy. We believe LZ4 is almost always better than Snappy. We leave Snappy as default to avoid unexpected compatibility problems to previous users. LZ4/Snappy is a lightweight compression algorithm so it usually strikes a good balance between space and CPU usage.

If you want to further reduce space and have some free CPU to use, you can try to set a heavy-weight compression by setting `options.bottommost_compression`. The bottommost level will be compressed using this compression style. Usually the bottommost level contains majority of the data, so users get an almost optimal space setting, without paying CPU for compressing all the data at any level. We recommend ZSTD. If it is not available, Zlib is the second choice.

If you have a lot of free CPU and want to reduce not just space but write amplification too, try to set `options.compression` to heavy weight compression type. We recommend ZSTD. Use Zlib if it is not available.

With a countermanded legacy setting `options.compression_per_level`, you can have an even finer control of compression style of each level. When this option is used, `options.compression` is ignored, while `options.bottommost_compression` still applies. But we believe there are very few use cases where this tuning will help.

Be aware that when you set different compression to different levels, compaction “trivial moves” that violate the compression styles will not be executed, and the file will be rewrite using the expected compression.

The specified compression type always applies to both of index and data blocks. You can disable compression for index blocks by setting

```
BlockBasedTableOptions.enable_index_compression = false .
```

## Compression level and window size setting

Some compression types support different compression level and window setting. You can set them through `options.compression_opts`. If the compression type doesn't support these setting, it will be a no-op.

# Dictionary Compression

Users can choose to compress each SST file of their bottommost level with a dictionary stored in the file. In some use cases, this can save some space. See [Dictionary Compression](#).

## Compression Library

If you pick a compression type but the library for it is not available, RocksDB will fall back to no compression. RocksDB will print out availability of compression types in the header of log files like this:

```
1. 2017/12/01-17:34:59.368239 7f768b5d0200 Compression algorithms supported:  
2. 2017/12/01-17:34:59.368240 7f768b5d0200      Snappy supported: 1  
3. 2017/12/01-17:34:59.368241 7f768b5d0200      Zlib supported: 1  
4. 2017/12/01-17:34:59.368242 7f768b5d0200      Bzip supported: 0  
5. 2017/12/01-17:34:59.368243 7f768b5d0200      LZ4 supported: 1  
6. 2017/12/01-17:34:59.368244 7f768b5d0200      ZSTDNotFinal supported: 1  
7. 2017/12/01-17:34:59.368282 7f768b5d0200      ZSTD supported: 1
```

Check the logging for potential compilation problems.

# Purpose

Dynamic dictionary compression algorithms have trouble compressing small data. With default usage, the compression dictionary starts off empty and is constructed during a single pass over the input data. Thus small input leads to a small dictionary, which cannot achieve good compression ratios.

In RocksDB, each block in a block-based table (SST file) is compressed individually. Block size defaults to 4KB, from which the compression algorithm cannot build a sizable dictionary. The dictionary compression feature presets the dictionary with data sampled from multiple blocks, which improves compression ratio when there are repetitions across blocks.

# Usage

Set `rocksdb::CompressionOptions::max_dict_bytes` (in `include/rocksdb/options.h`) to a nonzero value indicating the maximum per-file dictionary size.

Also `rocksdb::CompressionOptions::zstd_max_train_bytes` may be used to generate a training dictionary of max bytes for ZStd compression. Using ZStd's dictionary trainer can achieve even better compression ratio improvements than using `max_dict_bytes` alone. The training data will be used to generate a dictionary of `max_dict_bytes`.

# Implementation

Dictionary compression is only implemented for the bottom-most level. The dictionary will be constructed for each SST file in a subcompaction by sampling entire data blocks in the file. When dictionary compression is enabled, the uncompressed data blocks in the file being generated will be buffered in memory, upto `target_file_size` bytes. Once the limit is reached, or the file is finished, data blocks are taken uniformly/randomly from the buffered data blocks and used to train the ZStd dictionary trainer.

The dictionary is stored in the file's meta-block in order for it to be known when uncompressing. During reads, if `BlockBasedTableOptions::cache_index_and_filter_blocks` is `false`, the dictionary meta-block is read and pinned in memory but not charged to the block cache. If it is `true`, the dictionary meta-block is cached in the block cache, with high priority if `cache_index_and_filter_blocks_with_high_priority` is `true`, and with low priority otherwise. As for prefetching and pinning the dictionary meta-block in the cache, the behavior depends on the RocksDB version. In RocksDB versions lower than 6.4, the dictionary meta-block is not prefetched or pinned in the cache, unlike index and filter blocks. Starting RocksDB version 6.4, the dictionary meta-block is prefetched and pinned in the block cache the same way as index/filter blocks (for instance, setting `pin_index_and_filter_blocks_in_cache` to `true` results in the dictionary

meta-block getting pinned as well for L0 files).

The in-memory uncompression dictionary is a digested form of the raw dictionary stored on disk, and is larger in size. The digested form makes uncompression faster, but does consume more memory.

## Limitations

---

- Applies only to bottommost level

# Background and motivations

**SST integrity:** In current RocksDB, we calculate the checksum for block (e.g., data block) before they are flushed to file system and store the checksum in block trailer. When reading the blocks, the checksum is verified. It ensures the correctness of data block. However, to better protect the data in RocksDB, checksum for each SST file is needed, especially when the SST files are stored remotely or the SST file are moved or copied. File might be corrupted during the transmission or when it is stored in the storage.

**SST identity:** If a wrong SST file is transferred to a RocksDB SST file directory, all block checksum will match, but it doesn't contain the data we want. This usually can be caught by file name and file size mismatch because the chance that two different SST files share the same size is very small, but it may not be a good assumption to make. A full file checksum

SST file checksum can be used when: 1) SST files are copied to other places (e.g., backup, move, or replicate); 2) SST files are stored remotely, 3) ingesting external SST files to RocksDB, 4) verify the SST file when the whole file is read in DB (e.g., compaction).

# Design

1. where to generate: SST file checksum is generated when a SST file is generated in RocksDB (1. flush Memtable 2. compaction) via `writeable_file_writer`.
2. Flexibility
  - i. `options.file_checksum_gen_factory` is for upper-layer applications to plugin a specific file checksum generator factory implementation.  
`FileChecksumGenFactory` creates a `FileChecksumGenerator` object for each SST file and it generates the file checksum for a certain file. The object IS NOT shared, so `FileChecksumGenerator` can store the intermediate data during checksum generating in the object and the implementation does not need to be thread safe.
  - ii. Provide a default checksum generator (`FileChecksumGenCrc32c`) and factory (`FileChecksumGenCrc32cFactory`) for SST files (based on Crc32c) such that user can easily use it if they do not have their own requirement.
  - iii. The checksum value is `std::string`, any other checksum value type such as `uint32`, `int`, `uint64` can be easily converted to a string type. `checksum` function name is also a string.
3. what should be stored
  - i. the checksum value if self.
  - ii. the name of the checksum function: there are many different checksum functions. Therefore, the checksum value should be pair with its function name. Otherwise, either RocksDB or the application is not able to make meaningful checksum check.

4. where to store the checksums
  - i. we store the checksum function name and checksum value in vstorage as part of FileMetadata.
  - ii. we store the checksum function name and checksum value in MANIFEST for persistency
5. Tools: Dump the checksum of all SST file from MANIFEST in a map (in ldb)

## How to use

In order to enable the full file checksum, user needs to initialize the Options.file\_checksum\_gen\_factory. For example:

```

1. Options options;
2. FileChecksumGenCrc32cFactory* file_checksum_gen_factory = new FileChecksumGenCrc32cFactory();
3. options.file_checksum_gen_factory.reset(file_checksum_gen_factory);
4. ImmutableCFOptions ioptions(options);
5. .....

```

To implement a customized checksum generator factory, the application needs to implement a checksum generator. For example:

```

1. class FileChecksumGenCrc32c : public FileChecksumGenerator {
2. public:
3.   FileChecksumGenCrc32c(const FileChecksumGenContext& /*context*/) {
4.     checksum_ = 0;
5.   }
6.   void Update(const char* data, size_t n) override {
7.     checksum_ = crc32c::Extend(checksum_, data, n);
8.   }
9.   void Finalize() override { checksum_str_ = Uint32ToString(checksum_); }
10.  std::string GetChecksum() const override { return checksum_str_; }
11.  const char* Name() const override { return "FileChecksumCrc32c"; }
12. private:
13.   uint32_t checksum_;
14.   std::string checksum_str_;
15. };

```

And also the checksum generator factory, for example:

```

1. class FileChecksumGenCrc32cFactory : public FileChecksumGenFactory {
2. public:
3.   std::unique_ptr<FileChecksumGenerator> CreateFileChecksumGenerator(
4.     const FileChecksumGenContext& context) override {
5.     return std::unique_ptr<FileChecksumGenerator>(
6.       new FileChecksumGenCrc32c(context));
7.   }
8.   const char* Name() const override { return "FileChecksumGenCrc32cFactory"; }
9. };

```

When `sst_file_checksum_func` is initialized (`!=nullptr`), RocksDB generate the checksum value when creating the SST file.

In the current stage, we do not provide a public db interface to list or get the checksum value and checksum function name. However, there are two ways that user can get the checksum.

1. by calling `db->GetLiveFileMetadata(std::vector<LiveFileMetaData>)`, checksum value and checksum function name are included in the `LiveFileMetadata`. The checksum information is from vstorage in memory.
2. If the db is not running, or if user only has the Manifest file, we can use ldb tool to print a list of checksum with the file name. It will print a list of SST file wit checksum information as the following format:[file\_number, checksum\_function\_name, checksum value]

```
1. ./ldb --db=<db path> file_checksum_dump.
```

## The Next Step

We plan to work on following:

1. Take advantage of SST file checksum with backup engine.
2. Work with some use cases to apply the full file checksum.
3. Implement WAL file checksum and store them in manifest too.

Currently in RocksDB, an error during a write operation (write to WAL, Memtable Flush, background compaction etc) may cause the database instance to go into read-only mode and further user writes may not be accepted. The variable `ErrorHandler::bg_error_` is set to the failure `Status`. The `DBOptions::paranoid_checks` option controls how aggressively RocksDB will check and flag errors. The default value of this option is true. The table below shows the various scenarios where errors are considered and potentially affect database operations.

| Error Reason                                                                                                                                                       | When bgerror is set                             |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------|
| Sync WAL ( <code>BackgroundErrorReason::kWriteCallback</code> )                                                                                                    | Always                                          |
| Memtable insert failure ( <code>BackgroundErrorReason::kMemTable</code> )                                                                                          | Always                                          |
| Memtable flush ( <code>BackgroundErrorReason::kFlush</code> )                                                                                                      | <code>DBOptions::paranoid_checks</code> is true |
| <code>SstFileManager::IsMaxAllowedSpaceReached()</code> reports max allowed space reached during memtable flush<br>( <code>BackgroundErrorReason::kFlush</code> )  | Always                                          |
| <code>SstFileManager::IsMaxAllowedSpaceReached()</code> reports max allowed space reached during compaction<br>( <code>BackgroundErrorReason::kCompaction</code> ) | Always                                          |
| <code>DB::CompactFiles</code> ( <code>BackgroundErrorReason::kCompaction</code> )                                                                                  | <code>DBOptions::paranoid_checks</code> is true |
| Background compaction ( <code>BackgroundErrorReason::kCompaction</code> )                                                                                          | <code>DBOptions::paranoid_checks</code> is true |
| Write ( <code>BackgroundErrorReason::kWriteCallback</code> )                                                                                                       | <code>DBOptions::paranoid_checks</code> is true |

## Detection

If the database instance goes into read-only mode, the following foreground operations will return the error status on all subsequent calls -

- `DB::Write` , `DB::Put` , `DB::Delete` , `DB::SingleDelete` , `DB::DeleteRange` , `DB::Merge`
- `DB::IngestExternalFile`
- `DB::CompactFiles`
- `DB::CompactRange`
- `DB::Flush`

The returned `Status` will indicate the error code, sub-code as well as severity. The severity of the error can be determined by calling `Status::severity()`. There are 4 severity levels and they are defined as follows -

1. `Status::Severity::kSoftError` - Errors of this severity do not prevent writes to the DB, but it does mean that the DB is in a degraded mode. Background compactions and flush may not be able to run in a timely manner.
2. `Status::Severity::kHardError` - The DB is in read-only mode, but it can be transitioned back to read-write mode once the cause of the error has been addressed.
3. `Status::Severity::kFatalError` - The DB is in read-only mode. The only way to recover is

to close the DB, remedy the underlying cause of the error, and then re-open the DB.

4. `Status::Severity::kUnrecoverableError` - This is the highest severity and indicates a corruption in the database. It may be possible to close and re-open the DB, but the contents of the database may no longer be correct.

In addition to the above, a notification callback `EventListener::OnBackgroundError` will be called as soon as the background error is encountered.

## Recovery

---

There are 3 possible ways to recover from a background error without shutting down the database -

1. The `EventListener::OnBackgroundError` callback can override the error status if it determines that its not serious enough to stop further writes to the DB. It can do so by setting the `bg_error` parameter. Doing so can be risky, as RocksDB may not be able to guarantee the consistency of the DB. Check the `BackgroundErrorReason` and severity of the error before overriding it.
2. Call `DB::Resume()` to manually resume the DB and put it in read-write mode. This function will flush memtables for all the column families, clear the error, purge any obsolete files, and restart background flush and compaction operations.
3. Automatic recovery from background errors. This is done by polling the system to ensure the underlying error condition is resolved, and then following the same steps as `DB::Resume()` to restore write capability. Notification callbacks `EventListener::OnErrorRecoveryBegin` and `EventListener::OnErrorRecoveryCompleted` are called at the start and end of the recovery process respectively, to inform the user of the status. The retry behavior can be controlled by setting `max_bgerror_resume_count` and `bgerror_resume_retry_interval` in `DBOptions`.

## Auto Recovery Situations

---

At present, the automatic recovery is supported in the following scenarios -

1. ENOSPC error from the filesystem
2. IO errors reported by the `FileSystem` as retryable (typically transient errors such as network outages). When WAL is not in use, the database will continue to buffer writes in the memtable (i.e the database remains in read-write mode). Writes may eventually stall once `max_write_buffer_number` memtables are accumulated.
3. Errors during WAL sync, recovery is done only if 2PC is not in use.

We may add more cases in the future.

## When to need it

When the DB uses tons of gigabytes of memory, there is a higher chance that when accessing data in memory, the program will get data TLB miss, as well as cache misses to retrieve the mapping. When using hash table based indexes and bloom filters, provided by some mem tables and table readers, users are more prone to it, because the data locality is worse. Those indexes and blooms are perfect candidates to be allocated in huge page TLB. When you see a high data TLB overheads in data structures where the feature is supported, consider to give it a try.

Currently, it is only supported in Linux.

## How to use it

### Requisition

- You need to pre-reserve huge pages in linux
- Find out the huge page size to allocate

See Linux Documentation/vm/hugetlbpage.txt for details.

### Configure

Here is where the feature is available to use and how:

1. mem table's bloom filter: set Options.memtable\_prefix\_bloom\_huge\_page\_tlb\_size to be the huge page size.
2. hash linked list mem table's indexes and bloom filters: when calling NewHashLinkListRepFactory() to create a factory object for the mem table, pass huge page size into the parameter huge\_page\_tlb\_size.
3. PlainTableReader's indexes and bloom filters. When creating table factory for plain tables using NewPlainTableFactory() or NewTotalOrderPlainTableFactory(), set the parameter huge\_page\_tlb\_size to be the huge page size.

- [Logger](#)
- [Statistics](#)
- [Perf Context and IO Stats Context](#)
- [EventListener](#)

# Introduction

RocksDB supports a generalized message logging infrastructure. RocksDB caters to a variety of use cases – from low power mobile systems to high end servers running distributed applications. The framework helps extent the message logging infrastructure as per the use case requirements. The mobile app might need a relatively simpler logging mechanism, compared to a server running mission critical application. It also provides a means to integrate RocksDB log messages with the embedded application logging infrastructure.

## Existing Logger Implementations

The [Logger](#) class provides the interface definition for logging messages from RocksDB.

The various implementations of Logger available are:

| Implementation | Use                                                                                |
|----------------|------------------------------------------------------------------------------------|
| NullLogger     | /dev/null equivalent for logger                                                    |
| StderrLogger   | Pipes the messages to std::err equivalent                                          |
| HdfsLogger     | Logs messages to HDFS                                                              |
| PosixLogger    | Logs messages to POSIX file                                                        |
| AutoRollLogger | Automatically rolls files as they reach a certain size. Typically used for servers |
| EnvLogger      | Log using an arbitrary Env                                                         |
| WinLogger      | Specialized logger for Windows OS                                                  |

## Writing Your Custom Logger

Users are encouraged to write their own logging infrastructure as per the use case by extending any one of the existing logger implementations.

## Auto Roll Logger

With the default logger, the log files can grow to very large. It makes it harder for users to budget the disk space for it, especially for databases that are relatively small. Auto roll logger can help cap the disk usage by information logs. An auto roll logger will be automatically created on top of the logger defined in Env, if a user sets `options.max_log_file_size`, each file will be capped with that size, and combine the option with `options.keep_log_file_num` will have the total size of info log files under control.

`options.log_file_time_to_roll` can make log file rolling triggered by time.

DB Statistics provides cumulative stats over time. It serves different function from DB properties and [perf and IO Stats context](#): statistics accumulate stats for history, while DB properties report current state of the database; DB statistics give an aggregated view across all operations, whereas perf and IO stats context allow us to look inside of individual operations.

## Usage

Function `CreateDBStatistics()` creates a statistics object.

Here is an example to pass it to one DB:

```
1. Options options;
2. options.statistics = rocksdb::CreateDBStatistics();
```

Technically, you can create a statistics object and pass to multiple DBs. Then the statistics object will contain aggregated values for all those DBs. Note that some stats are undefined and have no meaningful information across multiple DBs. One such statistic is “rocksdb.sequence.number”.

Advanced users can implement their own statistics class. See the last section for details.

## Stats Level And Performance Costs

The overhead of statistics is usually small but non-negligible. We usually observe an overhead of 5%-10%.

Stats are implemented using atomic integers (atomic increments). Furthermore, stats measuring time duration require to calls the get the current time. Both of the atomic increment and timing functions introduce overhead, which varies across different platforms.

We have five levels of statistics, `kExceptHistogramOrTimers` , `kExceptTimers` , `kExceptDetailedTimers` , `kExceptTimeForMutex` and `kAll` . ( `kExceptHistogramOrTimers` and `kExceptTimers` will only be available since 6.1 Release)

- `kAll` : Collects all stats, including measuring duration of mutex operations. If getting time is expensive on the platform to run, it can reduce scalability to more threads, especially for writes.
- `kExceptTimeForMutex` : Collects all stats except the counters requiring to get time inside the mutex lock. `rocksdb.db.mutex.wait.micros` counter is not measured. By measuring the counter, we call the timing function inside DB mutex. If the timing function is slow, it can reduce write throughput significantly.
- `kExceptDetailedTimers` : Collects all stats except time inside mutex lock AND time spent on compression.

- `kExceptTimers` : Excluding all timing stats.
  - `kExceptHistogramOrTimers` : Excluding all timing stats, as well as histograms.
- Histograms are more expensive than pure counter stats, because it needs to seek to specific bucket and maintain min/max/count/std, etc. This is the most lightweight level.

## Access The Stats

---

### Stats Types

There are two types of stats, ticker and histogram.

The ticker type is represented by 64-bit unsigned integer. The value never decreases or resets. Ticker stats are used to measure counters (e.g. “rocksdb.block.cache.hit”), cumulative bytes (e.g. “rocksdb.bytes.written”) or time (e.g. “rocksdb.l0.slowdown.micros”).

The histogram type measures distribution of a stat across all operations. Most of the histograms are for distribution of duration of a DB operation. Taking “rocksdb.db.get.micros” as an example, we measure time spent on each Get() operation and calculate the distribution for all of them.

### Print Human Readable String

We can get a human readable string of all the counters by calling `ToString()`.

## Dump Statistics Periodically in information logs

Statistics are automatically dumped to information logs, for periodic interval of `options.stats_dump_period_sec`. Before 5.18 release, it is only dumped after a compaction, so if the database doesn't serve any write for a long time, statistics may not be dumped, despite of `options.stats_dump_period_sec`.

### Access Stats Programmatically

We can also access specific stat directly from the statistics object. The list of ticker types can be found in enum `Tickers`. By calling `statistics.getTickerCount()` for a ticker type, we can retrieve the value. Similarly, single histogram stat can be queried by calling `statistics.histogramData()` with enum `Histograms`, or `statistics.getHistogramString()`.

### Stats For Time-Interval

All the statistics are cumulative since the opening of the DB. If you need to monitor or report it on time-interval basis, you can check the value periodically and compute the time interval value by taking the difference between the current value and the previous value.

## User-Defined Statistics

---

Statistics is an abstract class and users can implement their own class and pass it to options.statistics. This is useful when you want to integrate RocksDB's stats to your own stats system. When you implement a user-defined statistic, be aware of the volume of calls to recordTick() and measureTime() by RocksDB. The user-defined stats can easily be the performance bottleneck if not implemented carefully.

Perf Context and IO Stats Context can help us understand the performance bottleneck of individual DB operations. Options.statistics stores cumulative statistics for all operations from all threads since the opening of the DB. Perf Context and IO Stat Context look inside individual operations.

This is the header file for perf context:

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/perf\\_context.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/perf_context.h)

This is the header file for IO Stats Context:

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/iostats\\_context.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/iostats_context.h)

The level of profiling for the two is controlled by the same function in this header file: [https://github.com/facebook/rocksdb/blob/master/include/rocksdb/perf\\_level.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/perf_level.h)

Perf Context and IO Stats Context use the same mechanism. The only difference is that Perf Context measures functions of RocksDB, whereas IO Stats Context measures I/O related calls. These features need to be enabled in the thread where the query to profile is executed. If the profile level is higher than disable, RocksDB will update the counters in a thread-local data structure. After the query, we can read the counters from the structure.

## How to Use Them

Here is a typical example of using using Perf Context and IO Stats Context:

```
1. #include "rocksdb/iostats_context.h"
2. #include "rocksdb/perf_context.h"
3.
4. rocksdb::SetPerfLevel(rocksdb::PerfLevel::kEnableTimeExceptForMutex);
5.
6. rocksdb::get_perf_context()->Reset();
7. rocksdb::get_iostats_context()->Reset();
8.
9. ... // run your query
10.
11. rocksdb::SetPerfLevel(rocksdb::PerfLevel::kDisable);
12.
13. ... // evaluate or report variables of rocksdb::get_perf_context() and/or rocksdb::get_iostats_context()
```

Note that the same perf level is applied to both of Perf Context and IO Stats Context.

You can also call rocksdb::get\_perf\_context()->ToString() and rocksdb::get\_iostats\_context->ToString() for a human-readable report.

## Profile Levels And Costs

As always, there are trade-offs between statistics quantity and overhead, so we have designed several profile levels to choose from:

- `kEnableCount` will only enable counters.
- `kEnableTimeAndCPUTimeExceptForMutex` is introduced since CPU Time counters are introduced. With this level, non-mutex-timing related counters will be enabled, as well as CPU counters.
- `kEnableTimeExceptForMutex` enables counter stats and most stats of time duration, except when the timing function needs to be called inside a shared mutex.
- `kEnableTime` further adds stats for mutex acquisition and waiting time.

`kEnableCount` avoids the more expensive calls to get system times, which results in lower overhead. We often measure all of the operations using this level, and report them when specific counters are abnormal.

With `kEnableTimeExceptForMutex`, RocksDB may call the timing function dozens of times for an operation. Our common practice is to turn it on with sampled operations, or when the user requests it. Users need to be careful when choosing sample rates, since the cost of the timing functions varies across different platforms.

`kEnableTime` further allows timing within shared mutex, but profiling an operation may slow down other operations. When we suspect that mutex contention is the performance bottleneck, we use this level to verify the problem.

How do we deal with the counters disabled in a level? If a counter is disabled, it won't be updated.

## Stats

---

We are giving some typical examples of how to use those stats to solve your problems. We are not introducing all the stats here. A full description of all stats can be found in the header files.

## Perf Context

### Binary Searchable Costs

`user_key_comparison_count` helps us figure out whether too many comparisons in binary search can be a problem, especially when a more expensive comparator is used. Moreover, since number of comparisons is usually uniform based on the memtable size, the SST file size for Level 0 and size of other levels, an significant increase of the counter can indicate unexpected LSM-tree shape. You may want to check whether flush/compaction can keep up with the write speed.

### Block Cache and OS Page Cache Efficiency

`block_cache_hit_count` tells us how many times we read data blocks from block cache, and `block_read_count` tells us how many times we have to read blocks from the file system

(either block cache is disabled or it is a cache miss). We can evaluate the block cache efficiency by looking at the two counters over time.

`block_read_byte` tells us how many total bytes we read from the file system. It can tell us whether a slow query can be caused by reading large blocks from the file system. Index and bloom filter blocks are usually large blocks. A large block can also be the result of a very large key or value.

In many setting of RocksDB, we rely on OS page cache to reduce I/O from the device. In fact, in the most common hardware setting, we suggest users tune the OS page cache size to be large enough to hold all the data except the last level so that we can limit a ready query to issue no more than one I/O. With this setting, we will issue multiple file system calls but at most one of them end up with reading from the device. To verify whether, it is the case, we can use counter `block_read_time` to see whether total time spent on reading the blocks from file system is expected.

## Tombstones

When deleting a key, RocksDB simply puts a marker, called tombstone to memtable. The original value of the key will not be removed until we compact the files containing the keys with the tombstone. The tombstone may even live longer even after the original value is removed. So if we have lots of consecutive keys deleted, a user may experience slowness when iterating across these tombstones. Counters

`internal_delete_skipped_count` tells us how many tombstones we skipped.  
`internal_key_skipped_count` covers some other keys we skip.

## Get() Break-Down

We can use “get\_\*” stats to break down time inside one Get() query. The most important two are `get_from_memtable_time` and `get_from_output_files_time`. The counters tell us whether the slowness is caused by memtable, SST tables, or both. `seek_on_memtable_time` can tell us how much of the time is spent on seeking memtables.

## Write Break-Down

“write\_\*” stats break down write operations. `write_wal_time`, `write_memtable_time` and `write_delay_time` tell us the time is spent on writing WAL, memtable, or active slowdown. `write_pre_and_post_process_time` mostly means time spent on waiting in the writer queue. If the write is assigned to a commit group but it is not the group leader, `write_pre_and_post_process_time` will also include the time waiting for the group commit leader to finish.

## Iterator Operations Break-Down

“seek\_\*” and “find\_next\_user\_entry\_time” break down iterator operations. The most interesting one is `seek_child_seek_count`. It tells us how many sub-iterators we have, which mostly means number of sorted runs in the LSM tree.

## Per-level PerfContext

In order to provide deeper insights into the performance implication of LSM tree structures, per-level PerfContext ( `PerfContextByLevel` ) was introduced to break down counters by levels. `PerfContext::level_to_perf_context` was added to maintain the mapping from level number to `PerfContextByLevel` object. User can access it directly by calling `(rocksdb::get_perf_context()->level_to_perf_context))[level_number]`.

By default per-level PerfContext is disabled, `EnablePerLevelPerfContext()`, `DisablePerLevelPerfContext()`, and, `ClearPerLevelPerfContext` can be called to enable/disable/clear it. When per-level PerfContext is enabled, `rocksdb::get_perf_context->ToString()` will also include non-zero per-level perf context counters, in the form of

```
1. bloom_filter_useful = 1@level5, 2@level7
```

which means `bloom_filter_useful` was incremented once at level 5 and twice at level 7.

## IO Stats Context

We have counters for time spent on major file system calls. Write related counters are more interesting to look if you see stalls in write paths. It tells us we are slow because of which file system operations, or it's not caused by file system calls.

# EventListener

EventListener class contains a set of call-back functions that will be called when a specific RocksDB event happens such as completing a flush or compaction job. Such callback APIs can be used as a building block for developing custom features such as stats-collector or external compaction algorithm. Available EventListener callbacks can be found in [include/rocksdb/listener.h](#).

## How to use it?

In DBOptions, there's a variable called `listeners`, which allows developers to add custom EventListener to listen to the events of a specific rocksdb instance.

```
1. // A vector of EventListeners which call-back functions will be called
2. // when specific RocksDB event happens.
3. std::vector<std::shared_ptr<EventListener>> listeners;
```

To listen to a rocksdb instance, it can be done by simply adding a custom EventListener to `DBOptions::listeners` and use that options to open a DB:

```
1. // listen to a rocksdb instance
2. DBOptions db_options;
3. ...
4. db_options.listeners.emplace_back(new MyListener());
```

Note that in either case, unless specially specified in the documentation, all EventListener call-backs must be implemented in a thread-safe way even when an EventListener only listens to a single column family (For example, imagine the case where `OnCompactionCompleted()` could be called by multiple threads at the same time as a single column family might complete more than one compaction jobs at the same time).

## Listen to a specific event

The default behavior of all EventListener callbacks is no-op. This allows developers to only focus the event they're interested in. To listen to a specific event, it is as simple as implementing its related call-back. For example, the following EventListener counts the number of flush job completed since DB open by implementing

```
OnFlushCompleted() :
```

```
1. class FlushCountListener : public EventListener {
2. public:
3.     FlushCountListener() : flush_count_(0) {}
4.     void OnFlushCompleted(
5.         DB* db, const std::string& name,
```

```
6.     const std::string& file_path,
7.     bool triggered_writes_slowdown,
8.     bool triggered_writes_stop) override {
9.     flush_count_++;
10. }
11. private:
12.     std::atomic_int flush_count_;
13. };
```

## Threading

All EventListener callbacks will be called using the actual thread that involves in that specific event. For example, it is the RocksDB background flush thread that does the actual flush to call `EventListener::OnFlushCompleted()`. This allows developers to collect thread-dependent stats from the EventListener callback such as via thread local variable.

## Locking

All EventListener callbacks are designed to be called without the current thread holding any DB mutex. This is to prevent potential deadlock and performance issue when using EventListener callback in a complex way. However, all EventListener call-back functions should not run for an extended period of time before the function returns, otherwise RocksDB may be blocked. For example, it is not suggested to do

`DB::CompactFiles()` (as it may run for a long while) or issue many of `DB::Put()` (as Put may be blocked in certain cases) in the same thread in the EventListener callback. However, doing `DB::CompactFiles()` and `DB::Put()` in a thread other than the EventListener callback thread is considered safe.

In this page, we summarize some known issues and limitations that RocksDB users need to know:

- `rocksdb::DB` instances need to be destroyed before your main function exits.  
RocksDB instances usually depend on some internal static variables. Users need to make sure `rocksdb::DB` instances are destroyed before those static variables.
- Universal Compaction Style has a limitation on total data size. See [Universal Compaction](#).
- Some features are not supported in RocksJava, See [RocksJava Basics](#). If there is a feature in the C++ API which is missing from the Java API that you need, please open an [issue](#) with a feature request.
- If you use prefix iterating and iterate out of the prefix range, by running `Prev()` will not recover from the previous key and the results are undefined.
- If you use prefix iterating and you are changing iterating order, `Seek()>Prev()` or `Next()>Prev()`, you may not get correct results.
- Atomicity is by default not guaranteed after DB recovery for more than one multiple column families and WAL is disabled. In the case of no WAL and multiple column families, setting `atomic_flush` to true in `DBOptions` provides guarantee for atomicity after recovery.

# Information Logs

Usually, looking at information logs to see whether there are something abnormal is the first step of troubleshooting.

By default, RocksDB's information logs reside in the same directory as your database. Look for `LOG` for the most recent log file, and `LOG.*` files for older ones. If you've explicitly set `option.db_log_dir`, find the logs there. The log file names will contain information of the paths of your database.

Some users have overridden the default logger or use non-default ones and they'll need to find the logs accordingly. If options. Be aware that the default log level is `INFO`. With `DEBUG` level you'll see more information, and with `WARN` level less.

With the default logger, a log line might look like this:

```
2019/09/17-13:42:20.597910 7fef48069300 [__impl/db_impl_compaction_flush.cc:1473] [default] Manual compaction
1. starting
```

After the timestamp, `7fef48069300` is the thread ID. Since usually a flush or compaction happens in one same thread, the thread ID might help you correlate which lines belong to the same job. `__impl/db_impl_compaction_flush.cc:1473` shows the source file and line number where the line is logged. It might be cut short to avoid long log lines. `default` is the column family name.

## Examine data on disk

If you see that RocksDB has returned unexpected results. You may consider to examine the database files and see whether the data on disk is wrong and if it is wrong in what way. `ldb` is the best tool to do that (ldb stands for LevelDB and we never renamed it).

Start with examining the data with `get` and `scan` commands. These commands will only examine keys visible to users. If you want to examine invisible keys, e.g. tombstones, older versions, you can use `idump` command. Another useful command is `manifest_dump`, which shows the LSM-tree structure of the database. This can help narrow down the problem to LSM-tree metadata or certain SST files.

If you use a customized comparator, merge operator or Env, you may need to build a customized `ldb` for it to work with your database. You can do it by building a binary which calls `LDBTool::Run()` with customized options, or register your plug in using object registry before calling the function. See `ldb_tools.h` for details.

To examine a specific SST file, use `sst_dump` tool. Starting with `scan` to examine the logical data. If needed, command `raw` can help examine data in more details.

See `--help` and [Administration and Data Access Tool](#) for more details about the two tools.

## Debugging Performance Problems

---

It's a good idea to start with statistics. [Statistics](#) and the information returned by `DB::GetProperty()` are two good places to look. Information Logs contain some performance information about each flush or compaction. For slow reads, [Perf Context and IO Stats Context](#) can help break down the latency. [RocksDB Tuning Guide](#) has some information about RocksDB performance.

## Asking For Help

---

We use [github issues](#) only for bug reports, and use [RocksDB's Google Group](#) or [Facebook Group](#) for other issues. It's not always clear to users whether it is RocksDB bug or not. Pick one using your best judgement.

To help the community to help more efficiently, provide as much information as possible. Try to include:

- Your environment.
- The language binding you are using: C++, C, Java, or third-party bindings.
- RocksDB release number.
- Options used (e.g. paste the option file under the DB directory).
- Findings when examining the database with `lrb` or `sst_dump`, related statistics, etc.
- It may be helpful to paste the compaction summary.
- You can even consider to attach the information files. Data itself is not logged there.

When reporting bugs, [MyRocks's bug reporting guidelines](#) might be helpful too.

For performance related questions, it may be helpful to check [How to ask a performance related question](#).

- [Administration and Data Access Tool](#)
- [Benchmarking Tools](#)
- [How to Backup RocksDB?](#)
- [Replication Helpers](#)
- [Checkpoints](#)
- [How to persist in-memory RocksDB database](#)
- [Stress Test](#)
- [Third-party language bindings](#)
- [RocksDB Trace, Replay, Analyzer, and Workload Generation](#)
- [Block cache analysis and simulation tools](#)

# Ldb Tool

The ldb command line tool offers multiple data access and database admin commands. Some examples are listed below. For more information, please consult the help message displayed when running ldb without any arguments and the unit tests in tools/ldb\_test.py.

Example data access sequence:

```
1.      $ ./ldb --db=/tmp/test_db --create_if_missing put a1 b1
2.      OK
3.
4.
5.      $ ./ldb --db=/tmp/test_db get a1
6.      b1
7.
8.      $ ./ldb --db=/tmp/test_db get a2
9.      Failed: NotFound:
10.
11.     $ ./ldb --db=/tmp/test_db scan
12.     a1 : b1
13.
14.     $ ./ldb --db=/tmp/test_db scan --hex
15.     0x6131 : 0x6231
16.
17.     $ ./ldb --db=/tmp/test_db put --key_hex 0x6132 b2
18.     OK
19.
20.     $ ./ldb --db=/tmp/test_db scan
21.     a1 : b1
22.     a2 : b2
23.
24.     $ ./ldb --db=/tmp/test_db get --value_hex a2
25.     0x6232
26.
27.     $ ./ldb --db=/tmp/test_db get --hex 0x6131
28.     0x6231
29.
30.     $ ./ldb --db=/tmp/test_db batchput a3 b3 a4 b4
31.     OK
32.
33.     $ ./ldb --db=/tmp/test_db scan
34.     a1 : b1
35.     a2 : b2
36.     a3 : b3
37.     a4 : b4
38.
39.     $ ./ldb --db=/tmp/test_db batchput "multiple words key" "multiple words value"
40.     OK
41.
```

```

42. $ ./ldb --db=/tmp/test_db scan
43. Created bg thread 0x7f4a1dbff700
44. a1 : b1
45. a2 : b2
46. a3 : b3
47. a4 : b4
48. multiple words key : multiple words value

```

To dump an existing leveldb database in HEX:

```
1. $ ./ldb --db=/tmp/test_db dump --hex > /tmp/dbdump
```

To load the dumped HEX format data to a new leveldb database:

```
$ cat /tmp/dbdump | ./ldb --db=/tmp/test_db_new load --hex --compression_type=bzip2 --block_size=65536 --
1. create_if_missing --disable_wal
```

To compact an existing leveldb database:

```
1. $ ./ldb --db=/tmp/test_db_new compact --compression_type=bzip2 --block_size=65536
```

You can specify command line `--column_family=<string>` for which column family your query will be against.

`--try_load_options` will try to load the options file in the DB to open the DB. It is a good idea to always try to have this option on when you operate the DB. If you open the DB with default options, it may mess up LSM-tree structure which can't be recovered automatically.

## SST dump tool

sst\_dump tool can be used to gain insights about a specific SST file. There are multiple operations that sst\_dump can execute on a SST file.

```

1. $ ./sst_dump
2. file or directory must be specified.
3.
4. sst_dump --file=<data_dir_OR_sst_file> [--command=check|scan|raw]
5. --file=<data_dir_OR_sst_file>
6. Path to SST file or directory containing SST files
7.
8. --command=check|scan|raw|verify
    check: Iterate over entries in files but dont print anything except if an error is encountered (default
9. command)
10.     scan: Iterate over entries in files and print them to screen
11.     raw: Dump all the table contents to <file_name>.dump.txt
    verify: Iterate all the blocks in files verifying checksum to detect possible corruption but dont print
12. anything except if a corruption is encountered
13. recompress: reports the SST file size if recompressed with different

```

```

14.           compression types
15.
16.   --output_hex
17.     Can be combined with scan command to print the keys and values in Hex
18.
19.   --from=<user_key>
20.     Key to start reading from when executing check|scan
21.
22.   --to=<user_key>
23.     Key to stop reading at when executing check|scan
24.
25.   --prefix=<user_key>
26.     Returns all keys with this prefix when executing check|scan
27.     Cannot be used in conjunction with --from
28.
29.   --read_num=<num>
30.     Maximum number of entries to read when executing check|scan
31.
32.   --verify_checksum
33.     Verify file checksum when executing check|scan
34.
35.   --input_key_hex
36.     Can be combined with --from and --to to indicate that these values are encoded in Hex
37.
38.   --show_properties
39.     Print table properties after iterating over the file when executing
40.     check|scan|raw
41.
42.   --set_block_size=<block_size>
43.     Can be combined with --command=recompress to set the block size that will
44.     be used when trying different compression algorithms
45.
46.   --compression_types=<comma-separated list of CompressionType members, e.g.,
47.     kSnappyCompression>
48.     Can be combined with --command=recompress to run recompression for this
49.     list of compression types
50.
51.   --parse_internal_key=<0xKEY>
52.     Convenience option to parse an internal key on the command line. Dumps the
53.     internal key in hex format {'key' @ SN: type}

```

## Dumping SST file blocks

```
1. ./sst_dump --file=/path/to/sst/000829.sst --command=raw
```

This command will generate a txt file named /path/to/sst/000829\_dump.txt. This file will contain all index blocks and data blocks encoded in Hex. It will also contain information like table properties, footer details and meta index details.

## Printing entries in SST file

```
1. ./sst_dump --file=/path/to/sst/000829.sst --command=scan --read_num=5
```

This command will print the first 5 keys in the SST file to the screen. the output may look like this

```
1. 'Key1' @ 5: 1 => Value1
2. 'Key2' @ 2: 1 => Value2
3. 'Key3' @ 4: 1 => Value3
4. 'Key4' @ 3: 1 => Value4
5. 'Key5' @ 1: 1 => Value5
```

The output can be interpreted like this

```
1. '<key>' @ <sequence number>: <type> => <value>
```

Please notice that if your key has non-ascii characters, it will be hard to print it on screen, in this case it's a good idea to use --output\_hex like this

```
1. ./sst_dump --file=/path/to/sst/000829.sst --command=scan --read_num=5 --output_hex
```

You can also specify where do you want to start reading from and where do you want to stop by using --from and --to like this

```
1. ./sst_dump --file=/path/to/sst/000829.sst --command=scan --from="key2" --to="key4"
```

You can pass --from and --to using hexadecimal as well by using --input\_key\_hex

```
1. ./sst_dump --file=/path/to/sst/000829.sst --command=scan --from="0x6B657932" --to="0x6B657934" --input_key_hex
```

Checking SST file

```
1. ./sst_dump --file=/path/to/sst/000829.sst --command=check --verify_checksum
```

This command will iterate over all entries in the SST file but wont print any thing except if it encountered a problem in the SST file. It will also verify the checksum.

Printing SST file properties

```
1. ./sst_dump --file=/path/to/sst/000829.sst --show_properties
```

This command will read the SST file properties and print them, output may look like this

```
1. from [] to []
2. Process /path/to/sst/000829.sst
3. Sst file format: block-based
4. Table Properties:
```

```

5. -----
6. # data blocks: 26541
7. # entries: 2283572
8. raw key size: 264639191
9. raw average key size: 115.888262
10. raw value size: 26378342
11. raw average value size: 11.551351
12. data block size: 67110160
13. index block size: 3620969
14. filter block size: 0
15. (estimated) table size: 70731129
16. filter policy name: N/A
17. # deleted keys: 571272

```

## Trying different compression algorithms

sst\_dump can be used to check the size of the file under different compression algorithms.

```
1. ./sst_dump --file=/path/to/sst/000829.sst --show_compression_sizes
```

By using --show\_compression\_sizes sst\_dump will recreate the SST file in memory using different compression algorithms and report the size, output may look like this

```

1. from [] to []
2. Process /path/to/sst/000829.sst
3. Sst file format: block-based
4. Block Size: 16384
5. Compression: kNoCompression Size: 103974700
6. Compression: kSnappyCompression Size: 103906223
7. Compression: kZlibCompression Size: 80602892
8. Compression: kBZip2Compression Size: 76250777
9. Compression: kLZ4Compression Size: 103905572
10. Compression: kLZ4HCCompression Size: 97234828
11. Compression: kZSTDNotFinalCompression Size: 79821573

```

These files are created in memory and they are generated with block size of 16KB, the block size can be change by using --set\_block\_size.

# db\_bench

`db_bench` is the main tool that is used to benchmark RocksDB's performance. RocksDB inherited db\_bench from LevelDB, and enhanced it to support many additional options. db\_bench supports many benchmarks to generate different types of workloads, and its various options can be used to control the tests.

If you are just getting started with db\_bench, here are a few things you can try:

1. Start with a simple benchmark like `fillseq` (or `fillrandom`) to create a database and fill it with some data

```
1. ./db_bench --benchmarks="fillseq"
```

If you want more stats, add the meta operator "stats" and `--statistics` flag.

```
1. ./db_bench --benchmarks="fillseq,stats" --statistics
```

1. Read the data back

```
1. ./db_bench --benchmarks="readrandom" --use_existing_db
```

You can also combine multiple benchmarks to the string that is passed to `--benchmarks` so that they run sequentially. Example:

```
1. ./db_bench --benchmarks="fillseq,readrandom,readseq"
```

More in-depth example of db\_bench usage can be found [here](#) and [here](#).

Benchmarks List:

```
1.      fillseq      -- write N values in sequential key order in async mode
        fillseqdeterministic      -- write N values in the specified key order and keep the shape of the
2.  LSM tree
3.      fillrandom     -- write N values in random key order in async mode
        filluniquerandomdeterministic      -- write N values in a random key order and keep the shape of
4.  the LSM tree
5.      overwrite     -- overwrite N values in random key order in async mode
6.      fillsync       -- write N/100 values in random key order in sync mode
7.      fill100K       -- write N/1000 100K values in random order in async mode
8.      deleteseq      -- delete N keys in sequential order
9.      deleterandom   -- delete N keys in random order
10.     readseq        -- read N times sequentially
11.     readtocache    -- 1 thread reading database sequentially
12.     readreverse    -- read N times in reverse order
13.     readrandom     -- read N times in random order
14.     readmissing    -- read N missing keys in random order
```

```

15.      readwhilewriting    -- 1 writer, N threads doing random reads
16.      readwhilemerging   -- 1 merger, N threads doing random reads
17.      readrandomwriterandom -- N threads doing random-read, random-write
18.      prefixscanrandom   -- prefix scan N times in random order
19.      updaterrandom       -- N threads doing read-modify-write for random keys
20.      appendrrandom       -- N threads doing read-modify-write with growing values
21.      mergerandom        -- same as updaterrandom/appendrrandom using merge operator. Must be used with
22.      merge_operator
23.      readrandommergerandom -- perform N random read-or-merge operations. Must be used with merge_operator
24.      newiterator         -- repeated iterator creation
25.      seekrandom          -- N random seeks, call Next seek_nexsts times per seek
26.      seekrandomwhilewriting -- seekrandom and 1 thread doing overwrite
27.      seekrandomwhilemerging -- seekrandom and 1 thread doing merge
28.      crc32c              -- repeated crc32c of 4K of data
29.      xxhash               -- repeated xxHash of 4K of data
30.      acquireload          -- load N*1000 times
31.      fillseekseq          -- write N values in sequential key, then read them by seeking to each key
32.      randomtransaction     -- execute N random transactions and verify correctness
33.      randomreplacekeys    -- randomly replaces N keys by deleting the old version and putting the new
34.      version
35.      timeseries           -- 1 writer generates time series data and multiple readers doing random reads
36.      on id

```

For a list of all options:

```

1. $ ./db_bench -help
2. db_bench:
3. USAGE:
4. ./db_bench [OPTIONS]...
5.
6. Flags from tools/db_bench_tool.cc:
7. -advise_random_on_open (Advise random access on table file open) type: bool
8.   default: true
9. -allow_concurrent_memtable_write (Allow multi-writers to update mem tables
10.   in parallel.) type: bool default: true
11. -base_background_compactions (The base number of concurrent background
12.   compactions to occur in parallel.) type: int32 default: 1
13. -batch_size (Batch size) type: int64 default: 1
14. -benchmark_read_rate_limit (If non-zero, db_bench will rate-limit the reads
15.   from RocksDB. This is the global rate in ops/second.) type: uint64
16.   default: 0
17. -benchmark_write_rate_limit (If non-zero, db_bench will rate-limit the
18.   writes going into RocksDB. This is the global rate in bytes/second.)
19.   type: uint64 default: 0
20. -benchmarks (Comma-separated list of operations to run in the specified
21.   order. Available benchmarks:
22.     fillseq      -- write N values in sequential key order in async mode
23.     fillseqdeterministic -- write N values in the specified key order
24.     and keep the shape of the LSM tree
25.     fillrandom   -- write N values in random key order in async mode
26.     filluniquerandomdeterministic -- write N values in a random key
27.     order and keep the shape of the LSM tree
28.     overwrite    -- overwrite N values in random key order in async mode

```

```

29.      fillsync    -- write N/100 values in random key order in sync mode
30.      fill100K   -- write N/1000 100K values in random order in async mode
31.      deleteseq  -- delete N keys in sequential order
32.      deleterandom -- delete N keys in random order
33.      readseq    -- read N times sequentially
34.      readtocache -- 1 thread reading database sequentially
35.      readreverse -- read N times in reverse order
36.      readrandom  -- read N times in random order
37.      readmissing -- read N missing keys in random order
38.      readwhilewriting -- 1 writer, N threads doing random reads
39.      readwhilemerging -- 1 merger, N threads doing random reads
40.      readrandomwriterandom -- N threads doing random-read, random-write
41.      prefixscanrandom -- prefix scan N times in random order
42.      updaterrandom -- N threads doing read-modify-write for random keys
43.      appendrandom -- N threads doing read-modify-write with growing values
44.      mergerandom -- same as updaterrandom/appendrandom using merge operator.

45. Must be used with merge_operator
46.     readrandommergerandom -- perform N random read-or-merge operations. Must
47. be used with merge_operator
48.     newiterator -- repeated iterator creation
49.     seekrandom -- N random seeks, call Next seek_nexts times per seek
50.     seekrandomwhilewriting -- seekrandom and 1 thread doing overwrite
51.     seekrandomwhilemerging -- seekrandom and 1 thread doing merge
52.     crc32c -- repeated crc32c of 4K of data
53.     xxhash -- repeated xxHash of 4K of data
54.     acquireload -- load N*1000 times
55.     fillseekseq -- write N values in sequential key, then read them by
56. seeking to each key
57.     randomtransaction -- execute N random transactions and verify
58. correctness
59.     randomreplacekeys -- randomly replaces N keys by deleting the old
60. version and putting the new version

61.
62.     timeseries -- 1 writer generates time series data and
63. multiple readers doing random reads on id
64.

65. Meta operations:
66.     compact -- Compact the entire DB
67.     stats -- Print DB stats
68.     resetstats -- Reset DB stats
69.     levelstats -- Print the number of files and bytes per level
70.     sstables -- Print sstable info
71.     heaprofile -- Dump a heap profile (if supported by this port)
72. ) type: string
73. default:
74. "fillseq,fillseqdeterministic,fillsync,fillrandom,filluniquerandomdeterministic,overwrite,readrandom,newiterator,
75. -block_restart_interval (Number of keys between restart points for delta
76. encoding of keys in data block.) type: int32 default: 16
77. -block_size (Number of bytes in a block.) type: int32 default: 4096
78. -bloom_bits (Bloom filter bits per key. Negative means use default
79. settings.) type: int32 default: -1
80. -bloom_locality (Control bloom filter probes locality) type: int32
81. default: 0
82. -bytes_per_sync (Allows OS to incrementally sync SST files to disk while

```

```

82.      they are being written, in the background. Issue one request for every
83.      bytes_per_sync written. 0 turns it off.) type: uint64 default: 0
84.      -cache_high_pri_pool_ratio (Ratio of block cache reserve for high pri
85.          blocks. If > 0.0, we also enable
86.          cache_index_and_filter_blocks_with_high_priority.) type: double
87.          default: 0
88.          -cache_index_and_filter_blocks (Cache index/filter blocks in block cache.)
89.              type: bool default: false
90.              -cache_numshardbits (Number of shards for the block cache is 2 **
91.                  cache_numshardbits. Negative means use default settings. This is applied
92.                  only if FLAGS_cache_size is non-negative.) type: int32 default: 6
93.                  -cache_size (Number of bytes to use as a cache of uncompressed data)
94.                      type: int64 default: 8388608
95.                      -compaction_fadvice (Access pattern advice when a file is compacted)
96.                          type: string default: "NORMAL"
97.                          -compaction_pri (priority of files to compaction: by size or by data age)
98.                              type: int32 default: 0
99.                              -compaction_readahead_size (Compaction readahead size) type: int32
100.                                 default: 0
101.                                 -compaction_style (style of compaction: level-based, universal and fifo)
102.                                     type: int32 default: 0
103.                                     -compressed_cache_size (Number of bytes to use as a cache of compressed
104.   data.) type: int64 default: -1
105.   -compression_level (Compression level. For zlib this should be -1 for the
106.   default level, or between 0 and 9.) type: int32 default: -1
107.   -compression_max_dict_bytes (Maximum size of dictionary used to prime the
108.   compression library.) type: int32 default: 0
109.   -compression_ratio (Arrange to generate values that shrink to this fraction
110.   of their original size after compression) type: double default: 0.5
111.   -compression_type (Algorithm to use to compress the database) type: string
112.   default: "snappy"
113.   -cuckoo_hash_ratio (Hash ratio for Cuckoo SST table.) type: double
114.   default: 0.9000000000000002
115.   -db (Use the db with the following name.) type: string default: ""
116.   -db_write_buffer_size (Number of bytes to buffer in all memtables before
117.   compacting) type: int64 default: 0
118.   -delayed_write_rate (Limited bytes allowed to DB when soft_rate_limit or
119.   level0_slowdown_writes_trigger triggers) type: uint64 default: 8388608
120.   -delete_obsolete_files_period_micros (Ignored. Left here for backward
121.   compatibility) type: uint64 default: 0
122.   -deletepercent (Percentage of deletes out of reads/writes/deletes (used in
123.   RandomWithVerify only). RandomWithVerify calculates writepercent as (100
124. - FLAGS_readwritepercent - deletepercent), so deletepercent must be
125. smaller than (100 - FLAGS_readwritepercent)) type: int32 default: 2
126. - deletes (Number of delete operations to do. If negative, do FLAGS_num
127.     deletions.) type: int64 default: -1
128. - disable_auto_compactions (Do not auto trigger compactions) type: bool
129.     default: false
130. - disable_seek_compaction (Not used, left here for backwards compatibility)
131.     type: int32 default: 0
132. - disable_wal (If true, do not write WAL for write.) type: bool
133.     default: false
134. - dump_malloc_stats (Dump malloc stats in LOG ) type: bool default: true

```

```

135. -duration (Time in seconds for the random-ops tests to run. When 0 then num
136.   & reads determine the test duration) type: int32 default: 0
137. -enable_io_prio (Lower the background flush/compaction threads' IO
138.   priority) type: bool default: false
139. -enable_numa (Make operations aware of NUMA architecture and bind memory
140.   and cpus corresponding to nodes together. In NUMA, memory in same node as
141.   CPUs are closer when compared to memory in other nodes. Reads can be
142.   faster when the process is bound to CPU and memory of same node. Use
143.   "$numactl --hardware" command to see NUMA memory architecture.)
144.   type: bool default: false
145. -enable_pipelined_write (Allow WAL and memtable writes to be pipelined)
146.   type: bool default: true
147. -enable_write_thread_adaptive_yield (Use a yielding spin loop for brief
148.   writer thread waits.) type: bool default: true
149. -env_uri (URI for registry Env lookup. Mutually exclusive with --hdfs.)
150.   type: string default: ""
151. -expand_range_tombstones (Expand range tombstone into sequential regular
152.   tombstones.) type: bool default: false
153. -expire_style (Style to remove expired time entries. Can be one of the
154.   options below: none (do not expire data), compaction_filter (use a
155.   compaction filter to remove expired data), delete (seek IDs and remove
156.   expired data) (used in TimeSeries only).) type: string default: "none"
157. -fifo_compaction_allow_compaction (Allow compaction in FIFO compaction.)
158.   type: bool default: true
159. -fifo_compaction_max_table_files_size_mb (The limit of total table file
160.   sizes to trigger FIFO compaction) type: uint64 default: 0
161. -file_opening_threads (If open_files is set to -1, this option set the
162.   number of threads that will be used to open files during DB::Open())
163.   type: int32 default: 16
164. -finish_after_writes (Write thread terminates after all writes are
165.   finished) type: bool default: false
166. -hard_pending_compaction_bytes_limit (Stop writes if pending compaction
167.   bytes exceed this number) type: uint64 default: 137438953472
168. -hard_rate_limit (DEPRECATED) type: double default: 0
169. -hash_bucket_count (hash bucket count) type: int64 default: 1048576
170. -hdfs (Name of hdfs environment. Mutually exclusive with --env_uri.)
171.   type: string default: ""
172. -histogram (Print histogram of operation timings) type: bool default: false
173. -identity_as_first_hash (the first hash function of cuckoo table becomes an
174.   identity function. This is only valid when key is 8 bytes) type: bool
175.   default: false
176. -index_block_restart_interval (Number of keys between restart points for
177.   delta encoding of keys in index block.) type: int32 default: 1
178. -key_id_range (Range of possible value of key id (used in TimeSeries
179.   only).) type: int32 default: 100000
180. -key_size (size of each key) type: int32 default: 16
181. -keys_per_prefix (control average number of keys generated per prefix, 0
182.   means no special handling of the prefix, i.e. use the prefix comes with
183.   the generated random number.) type: int64 default: 0
184. -level0_file_num_compaction_trigger (Number of files in level-0 when
185.   compactions start) type: int32 default: 4
186. -level0_slowdown_writes_trigger (Number of files in level-0 that will slow
187.   down writes.) type: int32 default: 20

```

```

188. -level0_stop_writes_trigger (Number of files in level-0 that will trigger
189.     put stop.) type: int32 default: 36
190. -level_compaction_dynamic_level_bytes (Whether level size base is dynamic)
191.     type: bool default: false
192. -max_background_compactions (The maximum number of concurrent background
193.     compactions that can occur in parallel.) type: int32 default: 1
194. -max_background_flushes (The maximum number of concurrent background
195.     flushes that can occur in parallel.) type: int32 default: 1
196. -max_bytes_for_level_base (Max bytes for level-1) type: uint64
197.     default: 268435456
198. -max_bytes_for_level_multiplier (A multiplier to compute max bytes for
199.     level-N (N >= 2)) type: double default: 10
200. -max_bytes_for_level_multiplier_additional (A vector that specifies
201.     additional fanout per level) type: string default: ""
202. -max_compaction_bytes (Max bytes allowed in one compaction) type: uint64
203.     default: 0
204. -max_num_range_tombstones (Maximum number of range tombstones to insert.)
205.     type: int64 default: 0
206. -max_successive_merges (Maximum number of successive merge operations on a
207.     key in the memtable) type: int32 default: 0
208. -max_total_wal_size (Set total max WAL size) type: uint64 default: 0
209. -max_write_buffer_number (The number of in-memory memtables. Each memtable
210.     is of sizewrite_buffer_size.) type: int32 default: 2
211. -max_write_buffer_number_to_maintain (The total maximum number of write
212.     buffers to maintain in memory including copies of buffers that have
213.     already been flushed. Unlike max_write_buffer_number, this parameter does
214.     not affect flushing. This controls the minimum amount of write history
215.     that will be available in memory for conflict checking when Transactions
216.     are used. If this value is too low, some transactions may fail at commit
217.     time due to not being able to determine whether there were any write
218.     conflicts. Setting this value to 0 will cause write buffers to be freed
219.     immediately after they are flushed. If this value is set to -1,
220.     'max_write_buffer_number' will be used.) type: int32 default: 0
221. -memtable_bloom_size_ratio (Ratio of memtable size used for bloom filter. 0
222.     means no bloom filter.) type: double default: 0
223. -memtable_insert_with_hint_prefix_size (If non-zero, enable memtable insert
224.     with hint with the given prefix size.) type: int32 default: 0
225. -memtable_use_huge_page (Try to use huge page in memtables.) type: bool
226.     default: false
227. -memtablerep () type: string default: "skip_list"
228. -merge_keys (Number of distinct keys to use for MergeRandom and
229.     ReadRandomMergeRandom. If negative, there will be FLAGS_num keys.)
230.     type: int64 default: -1
231. -merge_operator (The merge operator to use with the database. If a new merge
232.     operator is specified, be sure to use fresh database. The possible merge
233.     operators are defined in utilities/merge_operators.h) type: string
234.     default: ""
235. -mergereadpercent (Ratio of merges to merges&reads (expressed as
236.     percentage) for the ReadRandomMergeRandom workload. The default value 70
237.     means 70% out of all read and merge operations are merges. In other
238.     words, 7 merges for every 3 gets.) type: int32 default: 70
239. -min_level_to_compress (If non-negative, compression starts from this
240.     level. Levels with number < min_level_to_compress are not compressed.

```

```

241.     Otherwise, apply compression_type to all levels.) type: int32 default: -1
242.     -min_write_buffer_number_to_merge (The minimum number of write buffers that
243.         will be merged togetherbefore writing to storage. This is cheap because
244.         it is an in-memory merge. If this feature is not enabled, then all
245.         these write buffers are flushed to L0 as separate files and this increases
246.         read amplification because a get request has to check in all of these
247.         files. Also, an in-memory merge may result in writing less data to
248.         storage if there are duplicate records in each of these individual write
249.         buffers.) type: int32 default: 1
250.     -mmap_read (Allow reads to occur via mmap-ing files) type: bool
251.         default: false
252.     -mmap_write (Allow writes to occur via mmap-ing files) type: bool
253.         default: false
254.     -new_table_reader_for_compaction_inputs (If true, uses a separate file
255.         handle for compaction inputs) type: int32 default: 1
256.     -num (Number of key/values to place in database) type: int64
257.         default: 1000000
258.     -num_column_families (Number of Column Families to use.) type: int32
259.         default: 1
260.     -num_deletion_threads (Number of threads to do deletion (used in TimeSeries
261.         and delete expire_style only).) type: int32 default: 1
262.     -num_hot_column_families (Number of Hot Column Families. If more than 0,
263.         only write to this number of column families. After finishing all the
264.         writes to them, create new set of column families and insert to them.
265.         Only used when num_column_families > 1.) type: int32 default: 0
266.     -num_levels (The total number of levels) type: int32 default: 7
267.     -num_multi_db (Number of DBs used in the benchmark. 0 means single DB.)
268.         type: int32 default: 0
269.     -numdistinct (Number of distinct keys to use. Used in RandomWithVerify to
270.         read/write on fewer keys so that gets are more likely to find the key and
271.         puts are more likely to update the same key) type: int64 default: 1000
272.     -open_files (Maximum number of files to keep open at the same time (use
273.         default if == 0)) type: int32 default: -1
274.     -optimistic_transaction_db (Open a OptimisticTransactionDB instance.
275.         Required for randomtransaction benchmark.) type: bool default: false
276.     -optimize_filters_for_hits (Optimizes bloom filters for workloads for most
277.         lookups return a value. For now this doesn't create bloom filters for the
278.         max level of the LSM to reduce metadata that should fit in RAM. )
279.         type: bool default: false
280.     -options_file (The path to a RocksDB options file. If specified, then
281.         db_bench will run with the RocksDB options in the default column family
282.         of the specified options file. Note that with this setting, db_bench will
283.         ONLY accept the following RocksDB options related command-line arguments,
284.         all other arguments that are related to RocksDB options will be ignored:
285.             --use_existing_db
286.             --statistics
287.             --row_cache_size
288.             --row_cache_numshardbits
289.             --enable_io_prio
290.             --dump_malloc_stats
291.             --num_multi_db
292.         ) type: string default: ""
293.     -perf_level (Level of perf collection) type: int32 default: 1

```

```

294. -pin_l0_filter_and_index_blocks_in_cache (Pin index/filter blocks of L0
295.   files in block cache.) type: bool default: false
296. -pin_slice (use pinnable slice for point lookup) type: bool default: true
297. -prefix_size (control the prefix size for HashSkipList and plain table)
298.   type: int32 default: 0
299. -random_access_max_buffer_size (Maximum windows randomaccess buffer size)
300.   type: int32 default: 1048576
301. -range_tombstone_width (Number of keys in tombstone's range) type: int64
302.   default: 100
303. -rate_limit_delay_max_milliseconds (When hard_rate_limit is set then this
304.   is the max time a put will be stalled.) type: int32 default: 1000
305. -rate_limiter_bytes_per_sec (Set options.rate_limiter value.) type: uint64
306.   default: 0
307. -read_amp_bytes_per_bit (Number of bytes per bit to be used in block
308.   read-amp bitmap) type: int32 default: 0
309. -read_cache_direct_read (Whether to use Direct IO for reading from read
310.   cache) type: bool default: true
311. -read_cache_direct_write (Whether to use Direct IO for writing to the read
312.   cache) type: bool default: true
313. -read_cache_path (If not empty string, a read cache will be used in this
314.   path) type: string default: ""
315. -read_cache_size (Maximum size of the read cache) type: int64
316.   default: 4294967296
317. -read_random_exp_range (Read random's key will be generated using
318.   distribution of num * exp(-r) where r is uniform number from 0 to this
319.   value. The larger the number is, the more skewed the reads are. Only used
320.   in readrandom and multireadrandom benchmarks.) type: double default: 0
321. -readonly (Run read only benchmarks.) type: bool default: false
322. -reads (Number of read operations to do. If negative, do FLAGS_num reads.)
323.   type: int64 default: -1
324. -readwritepercent (Ratio of reads to reads/writes (expressed as percentage)
325.   for the ReadRandomWriteRandom workload. The default value 90 means 90%
326.   operations out of all reads and writes operations are reads. In other
327.   words, 9 gets for every 1 put.) type: int32 default: 90
328. -report_bg_io_stats (Measure times spents on I/Os while in compactions. )
329.   type: bool default: false
330. -report_file (Filename where some simple stats are reported to (if
331.   --report_interval_seconds is bigger than 0)) type: string
332.   default: "report.csv"
333. -report_file_operations (if report number of file operations) type: bool
334.   default: false
335. -report_interval_seconds (If greater than zero, it will write simple stats
336.   in CVS format to --report_file every N seconds) type: int64 default: 0
337. -reverse_iterator (When true use Prev rather than Next for iterators that
338.   do Seek and then Next) type: bool default: false
339. -row_cache_size (Number of bytes to use as a cache of individual rows (0 =
340.   disabled).) type: int64 default: 0
341. -seed (Seed base for random number generators. When 0 it is deterministic.)
342.   type: int64 default: 0
343. -seek_nxts (How many times to call Next() after Seek() in fillseekseq,
344.   seekrandom, seekrandomwhilewriting and seekrandomwhilemerging)
345.   type: int32 default: 0
346. -show_table_properties (If true, then per-level table properties will be

```

```

347.     printed on every stats-interval when stats_interval is set and
348.     stats_per_interval is on.) type: bool default: false
349.     -simcache_size (Number of bytes to use as a simcache of uncompressed data.
350.         Negative value disables simcache.) type: int64 default: -1
351.     -skip_list_lookahead (Used with skip_list memtablerep; try linear search
352.         first for this many steps from the previous position) type: int32
353.         default: 0
354.     -soft_pending_compaction_bytes_limit (Slowdown writes if pending compaction
355.         bytes exceed this number) type: uint64 default: 68719476736
356.     -soft_rate_limit (DEPRECATED) type: double default: 0
357.     -statistics (Database statistics) type: bool default: false
358.     -statistics_string (Serialized statistics string) type: string default: ""
359.     -stats_interval (Stats are reported every N operations when this is greater
360.         than zero. When 0 the interval grows over time.) type: int64 default: 0
361.     -stats_interval_seconds (Report stats every N seconds. This overrides
362.         stats_interval when both are > 0.) type: int64 default: 0
363.     -stats_per_interval (Reports additional stats per interval when this is
364.         greater than 0.) type: int32 default: 0
365.     -stddev (Standard deviation of normal distribution used for picking keys
366.         (used in RandomReplaceKeys only).) type: double default: 2000
367.     -subcompactions (Maximum number of subcompactions to divide L0-L1
368.         compactions into.) type: uint64 default: 1
369.     -sync (Sync all writes to disk) type: bool default: false
370.     -table_cache_numshardbits () type: int32 default: 4
371.     -target_file_size_base (Target file size at level-1) type: int64
372.         default: 67108864
373.     -target_file_size_multiplier (A multiplier to compute target level-N file
374.         size (N >= 2)) type: int32 default: 1
375.     -thread_status_per_interval (Takes and report a snapshot of the current
376.         status of each thread when this is greater than 0.) type: int32
377.         default: 0
378.     -threads (Number of concurrent threads to run.) type: int32 default: 1
379.     -time_range (Range of timestamp that store in the database (used in
380.         TimeSeries only).) type: uint64 default: 100000
381.     -transaction_db (Open a TransactionDB instance. Required for
382.         randomtransaction benchmark.) type: bool default: false
383.     -transaction_lock_timeout (If using a transaction_db, specifies the lock
384.         wait timeout in milliseconds before failing a transaction waiting on a
385.         lock) type: uint64 default: 100
386.     -transaction_set_snapshot (Setting to true will have each transaction call
387.         SetSnapshot() upon creation.) type: bool default: false
388.     -transaction_sets (Number of keys each transaction will modify (use in
389.         RandomTransaction only). Max: 9999) type: uint64 default: 2
390.     -transaction_sleep (Max microseconds to sleep in between reading and
391.         writing a value (used in RandomTransaction only).) type: int32
392.         default: 0
393.     -truth_db (Truth key/values used when using verify) type: string
394.         default: "/dev/shm/truth_db/dbbench"
395.     -universal_allow_trivial_move (Allow trivial move in universal compaction.)
396.         type: bool default: false
397.     -universal_compression_size_percent (The percentage of the database to
398.         compress for universal compaction. -1 means compress everything.)
399.         type: int32 default: -1

```

```

400. -universal_max_merge_width (The max number of files to compact in universal
401. style compaction) type: int32 default: 0
402. -universal_max_size_amplification_percent (The max size amplification for
403. universal style compaction) type: int32 default: 0
404. -universal_min_merge_width (The minimum number of files in a single
405. compaction run (for universal compaction only).) type: int32 default: 0
406. -universal_size_ratio (Percentage flexibility while comparing file size
407. (for universal compaction only).) type: int32 default: 0
408. -use_adaptive_mutex (Use adaptive mutex) type: bool default: false
409. -use_blob_db (Open a BlobDB instance. Required for largevalue benchmark.)
410. type: bool default: false
411. -use_block_based_filter (if use kBlockBasedFilter instead of kFullFilter
412. for filter block. This is valid if only we use BlockTable) type: bool
413. default: false
414. -use_clock_cache (Replace default LRU block cache with clock cache.)
415. type: bool default: false
416. -use_cuckoo_table (if use cuckoo table format) type: bool default: false
417. -use_direct_io_for_flush_and_compaction (Use O_DIRECT for background flush
418. and compaction I/O) type: bool default: false
419. -use_direct_reads (Use O_DIRECT for reading data) type: bool default: false
420. -use_existing_db (If true, do not destroy the existing database. If you
421. set this flag and also specify a benchmark that wants a fresh database,
422. that benchmark will fail.) type: bool default: false
423. -use_fsync (If true, issue fsync instead of fdatasync) type: bool
424. default: false
425. -use_hash_search (if use kHashSearch instead of kBinarySearch. This is
426. valid if only we use BlockTable) type: bool default: false
427. -use_plain_table (if use plain table instead of block-based table format)
428. type: bool default: false
429. -use_single_deletes (Use single deletes (used in RandomReplaceKeys only).)
430. type: bool default: true
431. -use_stderr_info_logger (Write info logs to stderr instead of to LOG file.
432. ) type: bool default: false
433. -use_tailing_iterator (Use tailing iterator to access a series of keys
434. instead of get) type: bool default: false
435. -use_uint64_comparator (use Uint64 user comparator) type: bool
436. default: false
437. -value_size (Size of each value) type: int32 default: 100
438. -verify_checksum (Verify checksum for every block read from storage)
439. type: bool default: false
440. -wal_bytes_per_sync (Allows OS to incrementally sync WAL files to disk
441. while they are being written, in the background. Issue one request for
442. every wal_bytes_per_sync written. 0 turns it off.) type: uint64
443. default: 0
444. -wal_dir (If not empty, use the given dir for WAL) type: string default: ""
445. -wal_size_limit_MB (Set the size limit for the WAL Files in MB.)
446. type: uint64 default: 0
447. -wal_ttl_seconds (Set the TTL for the WAL Files in seconds.) type: uint64
448. default: 0
449. -writable_file_max_buffer_size (Maximum write buffer for Writable File)
450. type: int32 default: 1048576
451. -write_buffer_size (Number of bytes to buffer in memtable before
452. compacting) type: int64 default: 67108864

```

```

453. -write_thread_max_yield_usec (Maximum microseconds for
454.   enable_write_thread_adaptive_yield operation.) type: uint64 default: 100
455. -write_thread_slow_yield_usec (The threshold at which a slow yield is
456.   considered a signal that other processes or threads want the core.)
457.   type: uint64 default: 3
458. -writes (Number of write operations to do. If negative, do --num reads.)
459.   type: int64 default: -1
460. -writes_per_range_tombstone (Number of writes between range tombstones)
461.   type: int64 default: 0

```

## cache\_bench

```

1. ./cache_bench --help
2. cache_bench: Warning: SetUsageMessage() never called
3. ...
4. Flags from cache/cache_bench.cc:
5. -cache_size (Number of bytes to use as a cache of uncompressed data.)
6.   type: int64 default: 8388608
7. -erase_percent (Ratio of erase to total workload (expressed as a
8.   percentage)) type: int32 default: 10
9. -insert_percent (Ratio of insert to total workload (expressed as a
10.   percentage)) type: int32 default: 40
11. -lookup_percent (Ratio of lookup to total workload (expressed as a
12.   percentage)) type: int32 default: 50
13. -max_key (Max number of key to place in cache) type: int64
14.   default: 1073741824
15. -num_shard_bits (shard_bits.) type: int32 default: 4
16. -ops_per_thread (Number of operations per thread.) type: uint64
17.   default: 1200000
18. -populate_cache (Populate cache before operations) type: bool
19.   default: false
20. -threads (Number of concurrent threads to run.) type: int32 default: 16
21. -use_clock_cache () type: bool default: false

```

## persistent\_cache\_bench

```

1. $ ./persistent_cache_bench -help
2. persistent_cache_bench:
3. USAGE:
4. ./persistent_cache_bench [OPTIONS]...
5. ...
6. Flags from utilities/persistent_cache/persistent_cache_bench.cc:
7. -benchmark (Benchmark mode) type: bool default: false
8. -cache_size (Cache size) type: uint64 default: 18446744073709551615
9. -cache_type (Cache type. (block_cache, volatile, tiered)) type: string
10.   default: "block_cache"
11. -enable_pipelined_writes (Enable async writes) type: bool default: false
12. -iosize (Read IO size) type: int32 default: 4096

```

```
13. -log_path (Path for the log file) type: string default: "/tmp/log"
14. -nsec (nsec) type: int32 default: 10
15. -nthread_read (Lookup threads) type: int32 default: 1
16. -nthread_write (Insert threads) type: int32 default: 1
17. -path (Path for cachefile) type: string default: "/tmp/microbench/blkcache"
18. -volatile_cache_pct (Percentage of cache in memory tier.) type: int32
   default: 10
20. -writer_iosize (File writer IO size) type: int32 default: 4096
21. -writer_qdepth (File writer qdepth) type: int32 default: 1
```

## Backup API

For the C++ API, see `include/rocksdb/utilities/backupable_db.h`. The key abstraction is the backup engine, which exposes simple interfaces to create backups, get info about backups, and restore from backup. There are two distinct representations of backup engines: (1) `BackupEngine` for creating new backups, and (2) `BackupEngineReadOnly` for restoring from backup. Either one can be used to get info about backups.

## Creating and verifying a backup

In RocksDB, we have implemented an easy way to backup your DB and verify correctness. Here is a simple example:

```

1.  #include "rocksdb/db.h"
2.  #include "rocksdb/utilities/backupable_db.h"
3.
4.  #include <vector>
5.
6.  using namespace rocksdb;
7.
8.  int main() {
9.     Options options;
10.    options.create_if_missing = true;
11.    DB* db;
12.    Status s = DB::Open(options, "/tmp/rocksdb", &db);
13.    assert(s.ok());
14.    db->Put(...); // do your thing
15.
16.    BackupEngine* backup_engine;
17.    s = BackupEngine::Open(Env::Default(), BackupableDBOptions("/tmp/rocksdb_backup"), &backup_engine);
18.    assert(s.ok());
19.    s = backup_engine->CreateNewBackup(db);
20.    assert(s.ok());
21.    db->Put(...); // make some more changes
22.    s = backup_engine->CreateNewBackup(db);
23.    assert(s.ok());
24.
25.    std::vector<BackupInfo> backup_info;
26.    backup_engine->GetBackupInfo(&backup_info);
27.
28.    // you can get IDs from backup_info if there are more than two
29.    s = backup_engine->VerifyBackup(1 /* ID */);
30.    assert(s.ok());
31.    s = backup_engine->VerifyBackup(2 /* ID */);
32.    assert(s.ok());
33.    delete db;
34.    delete backup_engine;
35. }
```

This simple example will create a couple backups in “/tmp/rocksdb\_backup”. Note that you can create and verify multiple backups using the same engine.

Backups are normally incremental (see `BackupableDBOptions::share_table_files`). You can create a new backup with `BackupEngine::CreateNewBackup()` and only the new data will be copied to backup directory (for more details on what gets copied, see [Under the hood](#)).

Once you have some backups saved, you can issue `BackupEngine::GetBackupInfo()` call to get a list of all backups together with information on timestamp of the backup and the size (please note that sum of all backups’ sizes is bigger than the actual size of the backup directory because some data is shared by multiple backups). Backups are identified by their always-increasing IDs.

When `BackupEngine::VerifyBackups()` is called, it checks the file sizes in the backup directory against the original sizes of the corresponding files in the db directory. However, we do not verify checksums since it would require reading all the data. Note that the only valid use case for `BackupEngine::VerifyBackups()` is invoking it on a backup engine after that same engine was used for creating backup(s) because it uses state captured during backup time.

## Restoring a backup

Restoring is also easy:

```

1. #include "rocksdb/db.h"
2. #include "rocksdb/utilities/backupable_db.h"
3.
4. using namespace rocksdb;
5.
6. int main() {
7.     BackupEngineReadOnly* backup_engine;
8.     Status s = BackupEngineReadOnly::Open(Env::Default(), BackupableDBOptions("/tmp/rocksdb_backup"),
9. &backup_engine);
10.    assert(s.ok());
11.    s = backup_engine->RestoreDBFromBackup(1, "/tmp/rocksdb", "/tmp/rocksdb");
12.    assert(s.ok());
13.    delete backup_engine;
14. }
```

This code will restore the first backup back to “/tmp/rocksdb”. The first parameter of `BackupEngineReadOnly::RestoreDBFromBackup()` is the backup ID, second is target DB directory, and third is the target location of log files (in some DBs they are different from DB directory, but usually they are the same. See `Options::wal_dir` for more info).

`BackupEngineReadOnly::RestoreDBFromLatestBackup()` will restore the DB from the latest backup, i.e., the one with the highest ID.

Checksum is calculated for any restored file and compared against the one stored during the backup time. If a checksum mismatch is detected, the restore process is aborted and `Status::Corruption` is returned.

You must reopen any live databases to see the restored data.

## Backup directory structure

```

1. /tmp/rocksdb_backup/
2. └── LATEST_BACKUP
3.   └── meta
4.     └── 1
5.   └── private
6.     └── 1
7.       ├── CURRENT
8.       ├── MANIFEST-000008
9.       └── OPTIONS-000009
10.      └── shared_checksum
11.        └── 000007_1498774076_590.sst

```

`LATEST_BACKUP` is a file containing the highest backup ID. In our example above, it contains “1”. It was used to for getting latest backup number, but no longer needed since there’re easier ways to get the number from META. The file will be removed in RocksDB 5.0.

`meta` directory contains a “meta-file” describing each backup, where its name is the backup ID. For example, a meta-file contains a listing of all files belonging to that backup. The format is described fully in the implementation file (`utilities/backupable/backupable_db.cc`).

`private` directory always contains non-SST files (options, current, manifest, and WALs). In case `Options::share_table_files` is unset, it also contains the SST files.

`shared` directory (not shown) contains SST files when `Options::share_table_files` is set and `Options::share_files_with_checksum` is unset. In this directory, files are named using only by their name in the original DB. So, it should only be used to backup a single RocksDB instance; otherwise, filenames can conflict.

`shared_checksum` directory contains SST files when both `Options::share_table_files` and `Options::share_files_with_checksum` are set. In this directory, files are named using their name in the original database, size, and checksum. These attributes uniquely identify files that can come from multiple RocksDB instances.

## Backup performance

Beware that backup engine’s `open()` takes time proportional to the number of existing backups since we initialize info about files in each existing backup. So if you target a remote file system (like HDFS), and you have a lot of backups, then initializing the backup engine can take some time due to all the network round-trips. We recommend to keep your backup engine alive and not to recreate it every time you need to do a backup or restore.

Another way to keep engine initialization fast is to remove unnecessary backups. To delete unnecessary backups, just call `PurgeOldBackups(N)`, where N is how many backups you'd like to keep. All backups except the N newest ones will be deleted. You can also choose to delete arbitrary backup with call `DeleteBackup(id)`.

Also beware that performance is decided by reading from local db and copying to backup. Since you may use different environments for reading and copying, the parallelism bottleneck can be on one of the two sides. For example, using more threads for backup (See Advanced usage) won't be helpful if local db is on HDD, because the bottleneck in this condition is disk reading capability, which is saturated. Also a poor small HDFS cluster cannot show good parallelism. It'll be beneficial if local db is on SSD and backup target is a high-capacity HDFS. In our benchmarks, using 16 threads will reduce the backup time to 1/3 of single-thread job.

## Under the hood

When you call `BackupEngine::CreateNewBackup()`, it does the following:

1. Disable file deletions
2. Get live files (this includes table files, current, options and manifest file).
3. Copy live files to the backup directory. Since table files are immutable and filenames unique, we don't copy a table file that is already present in the backup directory. For example, if there is a file `00050.sst` already backed up and `GetLiveFiles()` returns `00050.sst`, we will not copy that file to the backup directory. However, checksum is calculated for all files regardless if a file needs to be copied or not. If a file is already present, the calculated checksum is compared against previously calculated checksum to make sure nothing crazy happened between backups. If a mismatch is detected, backup is aborted and the system is restored back to the state before `BackupEngine::CreateNewBackup()` is called. One thing to note is that a backup abortion could mean a corruption from a file in backup directory or the corresponding live file in current DB. Options, manifest and current files are always copied to the private directory, since they are not immutable.
4. If `flush_before_backup` was set to false, we also need to copy log files to the backup directory. We call `GetSortedWalFiles()` and copy all live files to the backup directory.
5. Re-enable file deletions

## Advanced usage

We can store user-defined metadata in the backups. Pass your metadata to

`BackupEngine::CreateNewBackupWithMetadata()` and then read it back later using `BackupEngine::GetBackupInfo()`. For example, this can be used to identify backups using different identifiers from our auto-incrementing IDs.

We also backup and restore the options file now. After restore, you can load the options from db directory using `rocksdb::LoadLatestOptions()` or `rocksdb::LoadOptionsFromFile()`.

The limitation is that not everything in options object can be transformed to text in a file. You still need a few steps to manually set up [missing items](#) in options after restore and load. Good news is that you need much less than previously.

You need to instantiate some env and initialize `BackupableDBOptions::backup_env` for `backup_target`. Put your backup root directory in `BackupableDBOptions::backup_dir`. Under the directory the files will be organized in the structure mentioned above.

`BackupableDBOptions::share_table_files` controls whether backups are done incrementally. If true, SST files will go under a “shared/” subdirectory. Conflicts can arise when different SST files use the same name (e.g., when multiple databases have the same target backup directory).

`BackupableDBOptions::share_files_with_checksum` controls how shared files are identified. If true, shared SST files are identified using checksum, size, and seqnum. This prevents the conflicts mentioned above when multiple databases use a common target backup directory.

`BackupableDBOptions::max_background_operations` controls the number of threads used for copying files during backup and restore. For distributed file systems like HDFS, it can be very beneficial to increase the copy parallelism.

`BackupableDBOptions::info_log` is a Logger object that is used to print out LOG messages if not-nullptr. See [Logger wiki](#).

If `BackupableDBOptions::sync` is true, we will use `fsync(2)` to sync file data and metadata to disk after every file write, guaranteeing that backups will be consistent after a reboot or if machine crashes. Setting it to false will speed things up a bit, but some (newer) backups might be inconsistent. In most cases, everything should be fine, though.

If you set `BackupableDBOptions::destroy_old_data` to true, creating new `BackupEngine` will delete all the old backups in the backup directory.

`BackupEngine::CreateNewBackup()` method takes a parameter `flush_before_backup`, which is false by default. When `flush_before_backup` is true, `BackupEngine` will first issue a memtable flush and only then copy the DB files to the backup directory. Doing so will prevent log files from being copied to the backup directory (since flush will delete them). If `flush_before_backup` is false, backup will not issue flush before starting the backup. In that case, the backup will also include log files corresponding to live memtables. Backup will be consistent with current state of the database regardless of `flush_before_backup` parameter.

## Further reading

For the implementation, see [utilities/backupable/backupable\\_db.cc](#).

There are multiple functions available for users to build a replication system with RocksDB as the storage engine for single node.

## Functions for full DB physical copy

---

`Checkpoints` can build a physical snapshot of the database to another file system directory, with files are hard linked to the source as much as possible.

An alternative is to use `GetLiveFiles()`, combining with `DisableFileDeletion()` and `EnableFileDeletion()`. `GetLiveFiles()` returns a full list of files needed to be copied for the database. In order to prevent them from being compacted while copying them, users can first call `DisableFileDeletion()`, and then retrieve the file list from `GetLiveFiles()`, copy them, and finally call `EnableFileDeletion()`.

## Function for streaming updates, and catching up

---

Through most systems distribute updates to replicas as an independent component, RocksDB also provides some APIs for doing it.

Incremental replication needs to be able to find and tail all the recent changes to the database. The API `GetUpdatesSince` allows an application to *tail* the RocksDB transaction log. It can continuously fetch transactions from the RocksDB transaction log and apply them to a remote replica or a remote backup.

A replication system typically wants to annotate each Put with some arbitrary metadata. This metadata may be used to detect loops in the replication pipeline. It can also be used to timestamp and sequence transactions. For this purpose, RocksDB supports an API called `PutLogData` that an application may use to annotate each Put with metadata. This metadata is stored only in the transaction log and is not stored in the data files. The metadata inserted via `PutLogData` can be retrieved via the `GetUpdatesSince` API.

RocksDB transaction logs are created in the database directory. When a log file is no longer needed, it is moved to the archive directory. The reason for the existence of the archive directory is because a replication stream that is falling behind might need to retrieve transactions from a log file that is way in the past. The API `GetSortedWalFiles` returns a list of all transaction log files.

Checkpoint is a feature in RocksDB which provides the ability to take a snapshot of a running RocksDB database in a separate directory. Checkpoints can be used as a point in time snapshot, which can be opened Read-only to query rows as of the point in time or as a Writeable snapshot by opening it Read-Write. Checkpoints can be used for both full and incremental backups.

The Checkpoint feature enables RocksDB to create a consistent snapshot of a given RocksDB database in the specified directory. If the snapshot is on the same filesystem as the original database, the SST files will be hard-linked, otherwise SST files will be copied. The manifest and CURRENT files will be copied. In addition, if there are multiple column families, log files will be copied for the period covering the start and end of the checkpoint, in order to provide a consistent snapshot across column families.

A Checkpoint object needs to be created for a database before checkpoints are created. The API is as follows:

```
1. Status Create(DB* db, Checkpoint** checkpoint_ptr);
```

Given a checkpoint object and a directory, the CreateCheckpoint function creates a consistent snapshot of the database in the given directory.

```
1. Status CreateCheckpoint(const std::string& checkpoint_dir);
```

The directory should not already exist and will be created by this API. The directory will be an absolute path. The checkpoint can be used as a read-only copy of the DB or can be opened as a standalone DB. When opened read/write, the SST files continue to be hard links and these links are removed when the files are obsoleted. When the user is done with the snapshot, the user can delete the directory to remove the snapshot.

Checkpoints are used for online backup in MyRocks, which is MySQL using RocksDB as the storage engine. (MySQL on RocksDB)

In recent months, we have focused on optimizing RocksDB for in-memory workloads. With growing RAM sizes and strict low-latency requirements, lots of applications decide to keep their entire data in memory. Running in-memory database with RocksDB is easy – just mount your RocksDB directory on tmpfs or ramfs [1]. Even if the process crashes, RocksDB can recover all of your data from in-memory filesystem. However, what happens if the machine reboots?

In this article we will explain how you can recover your in-memory RocksDB database even after a machine reboot.

Every update to RocksDB is written to two places - one is an in-memory data structure called memtable and second is write-ahead log. Write-ahead log can be used to completely recover the data in memtable. By default, when we flush the memtable to table file, we also delete the current log, since we don't need it anymore for recovery (the data from the log is “persisted” in the table file – we say that the log file is obsolete). However, if your table file is stored in in-memory file system, you may need the obsolete write-ahead log to recover the data after the machine reboots. Here's how you can do that.

`Options::wal_dir` is the directory where RocksDB stores write-ahead log files. If you configure this directory to be on flash or disk, you will not lose current log file on machine reboot. `Options::WAL_ttl_seconds` is the timeout when we delete the archived log files. If the timeout is non-zero, obsolete log files will be moved to `archive/` directory under `Options::wal_dir`. Those archived log files will only be deleted after the specified timeout.

Let's assume `Options::wal_dir` is a directory on persistent storage and `Options::WAL_ttl_seconds` is set to one day. To fully recover the DB, we also need to backup the current snapshot of the database (containing table and metadata files) with a frequency of less than one day. RocksDB provides an utility that enables you to easily backup the snapshot of your database. You can learn more about it here: <https://github.com/facebook/rocksdb/wiki/How-to-backup-RocksDB%3F>

You should configure the backup process to avoid backing up log files, since they are already stored in persistent storage. To do that, set `BackupableDBOptions::backup_log_files` to false.

Restore process by default cleans up entire DB and WAL directory. Since we didn't include log files in the backup, we need to make sure that restoring the database doesn't delete log files in WAL directory. When restoring, configure `RestoreOptions::keep_log_file` to true. That option will also move any archived log files back to WAL directory, enabling RocksDB to replay all archived log files and rebuild the in-memory database state.

To reiterate, here's what you have to do:

- Set DB directory to tmpfs or ramfs mounted drive
- Set `Options::wal_log` to a directory on persistent storage
- Set `Options::WAL_ttl_seconds` to T seconds

- Backup RocksDB every T/2 seconds, with `BackupableDBOptions::backup_log_files = false`
- When you lose data, restore from backup with `RestoreOptions::keep_log_file = true`

[1] You might also want to consider using PlainTable format for table files –  
<https://github.com/facebook/rocksdb/wiki/PlainTable-Format>

The reliability of RocksDB is critical. To keep it reliable while so many new features and improvements keep being made, besides normal unit tests, an important defense is continuous runs of stress test, called “crash test”, which proves to be an efficient way of catching correctness bugs.

Contrary to unit tests which tend to be: (1) deterministic; (2) covers one single functionality, or interaction with only a few features; (3) total run time is limited, crash tests to cover a different set of scenarios:

1. Randomized.
2. Covers combination of a wide range of features.
3. Continuously running.
4. Simulate failures.

## How to Run It

---

Right now, there are two crash tests to run:

```
1. make crash_test -j  
2. make crash_test_with_atomic_flush -j
```

Arguments can be passed to the crash test which will overwrite the default one. For eg:

```
1. export CRASH_TEST_EXT_ARGS="--use_direct_reads=1 --max_write_buffer_number=4"  
2. make crash_test -j
```

If you can't use GNU make, you can manually build db\_stress binary, and run script:

```
1. python -u tools/db_crashtest.py whitebox  
2. python -u tools/db_crashtest.py blackbox  
3. python -u tools/db_crashtest.py --simple whitebox  
4. python -u tools/db_crashtest.py --simple blackbox  
5. python -u tools/db_crashtest.py --cf_consistency blackbox  
6. python -u tools/db_crashtest.py --cf_consistency whitebox
```

Arguments can be passed to the crash test which will overwrite the default one. For eg:

```
1. python -u tools/db_crashtest.py whitebox --use_direct_reads=1 --max_write_buffer_number=4
```

To have it run continuously, it is recommended that different flavors of builds are used. For example, we run normal build, ASAN, TSAN and UBSAN continuously.

If you want to run crash tests in release mode, you need to set the DEBUG\_LEVEL environment to be 0.

If you want to provide the test directory, you need to set the TEST\_TMPDIR environment with the test directory path.

## Basic Workflow

---

The crash test run keeps running for several hours, and operates on one single DB. It crashes the program once a while. The database is reopened after the crash. After each crash, the database is restarted with a new set of options. Each option is turned on or off randomly. During each run, multiple operations are issued to the database concurrently with randomized inputs. Data validation is continuously executing, and after each restart, all the data is verified. Debug mode is used, so asserts can be triggered too.

## Database Operations

---

A database with a large number of rows (1M) in a number of column families(10) is created. We also create an array of large vectors to store the values of these rows in memory.

The rows are partitioned between multiple threads(32). Each thread is assigned to operate on a fixed contiguous portion of the rows. Each thread does a number of operations on its piece of the database and then verifies it with the in memory copy of the database which is also being updated as changes are made to the database. As configured, there can be different number of operations, threads, rows, column families, operations per batch, iterations, and operations between database reopen. Different database options can also be modified which results in various configurations and features being tested. eg. checksums, time to live, universal/level compaction, compression types to use at different levels, maximum number of write buffers, minimum number of buffers to flush.

The goal is for the test to cover as many database operations as possible. Right now, it issues most write operations, put, write, delete, single delete, range delete. From the read part, it covers get, multiget and iterators, with or without snapshots. We also issue administrative operations, including compact range, compact files, creating checkpoints, create/drop column families, etc. Some transaction operations are covered too.

While it already covers a wide range of database operations, it doesn't cover anything. Everyone is welcome to contribute to it and narrow the gap.

## Data Validation

---

Data is validated using several ways:

1. In one test case, when we add a key to the database, the value is

deterministically determined by the key itself. When we read the key back, we assert the value is expected.

2. Before data crashes, RocksDB remembers whether a key should exist or not. We keep querying random keys and see whether it shows up or doesn't show up in the database as expected.
3. While writing anything to the database, we also write to an external log file. Between after restarts, the database state is compared with the log file.
4. In another test case, we write the same key/value to multiple column families in a single write batch. We verify whether each column family contains completely the same data.

## Periodically crash the database

---

We periodically crash the system in two approach: “black box crash” or “white box crash”. The “black box crash” simply call Linux “kill -9” to shut down the program. With “white box crash”, some crash points are defined in several places in the code. The places are before and after RocksDB calls various file system operations. When the test program executes to these points, there is a chance that the program shuts down itself. The crash points are before or after file system operations, because only data written in file systems is used when the database is recovered from the crash. Between two file system operations, the data in the file system stays the same, so extra crashing points are unlikely to expose more bugs.

Third-party language bindings

Check out the list at <https://github.com/facebook/rocksdb/blob/master/LANGUAGE-BINDINGS.md>

# Query Tracing and Replaying

The trace\_replay APIs allow the user to track down the query information to a trace file. In the current implementation, Get, WriteBatch (Put, Delete, Merge, SingleDelete, and DeleteRange), Iterator (Seek and SeekForPrev) are the queries that tracked by the trace\_replay API. Key, query time stamp, value (if applied), cf\_id forms one trace record. Since one lock is used to protect the tracing instance and there will be extra IOs for the trace file, the performance of DB will be influenced. According to the current test on the MyRocks and ZippyDB shadow server, the performance hasn't been a concern in those shadows. The trace records from the same DB instance are written to a binary trace file. User can specify the path of trace file (e.g., store in different storage devices to reduce the IO influence).

Currently, the trace file can be replayed by using the db\_bench. The queries records in the traces file are replayed to the target DB instance according to the time stamps. It can replay the workload nearly the same as the workload being collected, which will provide a more production-like testing case.

An simple example to use the tracing APIs:

```

1. Options opt;
2. Env* env = rocksdb::Env::Default();
3. EnvOptions env_options(opt);
4. std::string trace_path = "/tmp/trace_test_example";
5. std::unique_ptr<TraceWriter> trace_writer;
6. DB* db = nullptr;
7. std::string db_name = "/tmp/rocksdb";
8.
9. /*Create the trace file writer*/
10. NewFileTraceWriter(env, env_options, trace_path, &trace_writer);
11. DB::Open(opt, db_name, &db);
12.
13. /*Start tracing*/
14. TraceOptions trace_opt;
15. db->StartTrace(trace_opt, std::move(trace_writer));
16.
17. /* your call of RocksDB APIs */
18.
19. /*End tracing*/
20. db->EndTrace()

```

To replay the trace:

```
1. ./db_bench --benchmarks=replay --trace_file=/tmp/trace_test_example --num_column_families=5
```

# Trace Analyzing, Visualizing, and Modeling

After the user finishes the tracing steps by using the trace\_replay APIs, the user will get one binary trace file. In the trace file, Get, Seek, and SeekForPrev are tracked with separate trace record, while queries of Put, Merge, Delete, SingleDelete, and DeleteRange are packed into WriteBatches. One tool is needed to 1) interpret the trace into the human readable format for further analyzing, 2) provide rich and powerful in-memory processing options to analyze the trace and output the corresponding results, and 3) be easy to add new analyzing options and query types to the tool.

The RocksDB team developed the initial version of the tool: trace\_analyzer. It provides the following analyzing options and output results.

Note that most of the generated analyzing results output files will be separated in different column families and different query types, which means, one query type in one column family will have its own output files. Usually, one specified output option will generate one output file.

## Analyze The Trace

---

The trace analyzer options

```

1. -analyze_delete (Analyze the Delete query.) type: bool default: false
2. -analyze_get (Analyze the Get query.) type: bool default: false
3. -analyze_iterator ( Analyze the iterate query like seek() and
4.   seekForPrev()) type: bool default: false
5. -analyze_merge (Analyze the Merge query.) type: bool default: false
6. -analyze_put (Analyze the Put query.) type: bool default: false
7. -analyze_range_delete (Analyze the DeleteRange query.) type: bool
8.   default: false
9. -analyze_single_delete (Analyze the SingleDelete query.) type: bool
10.  default: false
11. -convert_to_human_readable_trace (Convert the binary trace file to a human
12.   readable txt file for further processing. This file will be extremely
13.   large (similar size as the original binary trace file). You can specify
14.   'no_key' to reduce the size, if key is not needed in the next step
15.   File name: <prefix>-human_readable_trace.txt
16.   Format:[type_id cf_id value_size time_in_micosec <key>].) type: bool
17.   default: false
18. -key_space_dir (<the directory stores full key space files>
19.   The key space files should be: <column family id>.txt) type: string
20.   default: ""
21. -no_key ( Does not output the key to the result files to make smaller.)
22.   type: bool default: false
23. -no_print (Do not print out any result) type: bool default: false
24. -output_access_count_stats (Output the access count distribution statistics
25.   to file.
26.   File name:  <prefix>-<query
27.   type>-<cf_id>-accessed_key_count_distribution.txt
28.   Format:[access_count number_of_access_count]) type: bool default: false
29. -output_dir (The directory to store the output files.) type: string

```

```

30.    default: ""
31. -output_ignore_count (<threshold>, ignores the access count <= this value,
32. it will shorter the output.) type: int32 default: 0
33. -output_key_distribution (Output the key size distribution.) type: bool
34.     default: false
35. -output_key_stats (Output the key access count statistics to file
36. for accessed keys:
37.   File name: <prefix>-<query type>-<cf_id>-accessed_key_stats.txt
38.   Format:[cf_id value_size access_keyid access_count]
39.   for the whole key space keys:
40.     File name: <prefix>-<query type>-<cf_id>-whole_key_stats.txt
41.     Format:[whole_key_space_keyid access_count]) type: bool default: false
42. -output_prefix (The prefix used for all the output files.) type: string
43.     default: "trace"
44. -output_prefix_cut (The number of bytes as prefix to cut the keys.
45. if it is enabled, it will generate the following:
46. for accessed keys:
47.   File name: <prefix>-<query type>-<cf_id>-accessed_key_prefix_cut.txt
48.   Format:[accessed_keyid access_count_of_prefix number_of_keys_in_prefix
49. average_key_access prefix_succ_ratio prefix]
50. for whole key space keys:
51.   File name: <prefix>-<query type>-<cf_id>-whole_key_prefix_cut.txt
52.   Format:[start_keyid_in_whole_keyspace prefix]
53.   if 'output_qps_stats' and 'top_k' are enabled, it will output:
54.     File name: <prefix>-<query
55.     type>-<cf_id>-accessed_top_k_qps_prefix_cut.txt
56.     Format:[the_top_kth_qps_time QPS], [prefix qps_of_this_second].)
57.     type: int32 default: 0
58. -output_qps_stats (Output the query per second(qps) statistics
59. For the overall qps, it will contain all qps of each query type. The time
60. is started from the first trace record
61. File name: <prefix>-qps_stats.txt
62. Format: [qps_type_1 qps_type_2 ..... overall_qps]
63. For each cf and query, it will have its own qps output
64. File name: <prefix>-<query type>-<cf_id>-qps_stats.txt
65. Format:[query_count_in_this_second].) type: bool default: false
66. -output_time_series (Output the access time in second of each key, such
67. that we can have the time series data of the queries
68. File name: <prefix>-<query type>-<cf_id>-time_series.txt
69. Format:[type_id time_in_sec access_keyid].) type: bool default: false
70. -output_value_distribution (Output the value size distribution, only
71. available for Put and Merge.
72. File name: <prefix>-<query
73. type>-<cf_id>-accessed_value_size_distribution.txt
74. Format:[Number_of_value_size_between x and x+value_interval is: <the
75. count>]) type: bool default: false
76. -print_correlation (intput format: [correlation pairs][.,.])
77. Output the query correlations between the pairs of query types listed in
78. the parameter, input should select the operations from:
79. get, put, delete, single_delete, range_delete, merge. No space between
80. the pairs separated by commar. Example: =[get,put]... It will print out
81. the number of pairs of 'A after B' and the average time interval between
82. the two query) type: string default: ""

```

```

83. -print_overall_stats ( Print the stats of the whole trace, like total
84.   requests, keys, and etc.) type: bool default: true
85. -print_top_k_access (<top K of the variables to be printed> Print the top k
86.   accessed keys, top k accessed prefix and etc.) type: int32 default: 1
87. -trace_path (The trace file path.) type: string default: ""
88. -value_interval (To output the value distribution, we need to set the value
89.   intervals and make the statistic of the value size distribution in
90.   different intervals. The default is 8.) type: int32 default: 8

```

## One Example

```

./trace_analyzer -analyze_get -output_access_count_stats -output_dir=/data/trace/result -output_key_stats -
output_qps_stats -convert_to_human_readable_trace -output_value_distribution -output_key_distribution -
1. print_overall_stats -print_top_k_access=3 -output_prefix=test -trace_path=/data/trace/trace

```

## Query Type Options

User can specify which type queries that should be analyzed and use “-analyze\_”.

## Output Human Readable Traces

The original binary trace stores the encoded data structures and content, to interpret the trace, the tool should use the RocksDB library. Thus, to simplify the further analyzing of the trace, user can specify

```

1. -convert_to_human_readable_trace

```

The original trace will be converted to a txt file, the content is “[type\_id cf\_id  
value\_size time\_in\_micosec ]”. If the key is not needed, user can specify “-no\_key” to reduce the file size. This option is independent to all other option, once it is specified, the converted trace will be generated. If the original key is included, the txt file size might be similar or even larger than the original trace file size.

## Input and Output Options

To analyze a trace file, user need to indicate the path to the trace file by

```

1. -trace_path=<path to the trace>

```

To store the output files, user can specify a directory (make sure the directory exist before running the analyzer) to store these files

```

1. -output_dir=<the path to the output directory>

```

If user wants to analyze the accessed keys together with the existing keyspace. User needs to specify a directory that stores the keyspace files. The file should be in the name of “.txt” and each line is one key. Usually, user can use the “./ldb scan” of the LDB tool to dump out all the existing keys. To specify the directory

```
1. -key_space_dir=<the path to the key space directory>
```

To collect the output files more easily, user can specify the “prefix” for all the output files

```
1. -output_prefix=<the prefix, like "trace1">
```

If user does not want to print out the general statistics to the screen, user can specify

```
1. -no_print
```

## The Analyzing Options

Currently, the trace\_analyzer tool provides several different analyzing options to characterize the workload. Some of the results are directly printed out (options with prefix “-print”) others will output to the files (options with prefix “-output”). User can specify the combination of the options to analyze the trace. Note that, some of the analyzing options are very memory intensive (e.g., -output\_time\_series, -print\_correlation, and -key\_space\_dir). If the memory is not enough, try to run them in different time.

The general information of the workloads like the total number of keys, the number of analyzed queries of each column family, the key and value size statistics (average and medium), the number of keys being accessed are printed to screen when this option is specified

```
1. -print_overall_stats
```

To get the total access count of each key and the size of the value, user can specify

```
1. -output_key_stats
```

It will output the access count of each key to a file and the keys are sorted in lexicographical order. Each key will be assigned with an integer as the ID for further processing

In some workloads, the composition of the keys has some common part. For example, in MyRocks, the first X bytes of the key is the table index\_num. We can use the first X bytes to cut the keys into different prefix range. By specifying the number of bytes to cut the key space, the trace\_analyzer will generate a file. One record in the file represents a cut of prefix, the corresponding KeyID, and the prefix content are stored. There will be two separate files if the -key\_space\_dir is specified. One file is for the accessed keys, the other one is for the whole key space. Usually, the prefix cut file is used together with the accessed\_key\_stats.txt and whole\_key\_stats.txt respectively.

```
1. -output_prefix_cut=<number of bytes as prefix>
```

If user wants to visualize the accesses of the keys in the tracing timeline, user can specify:

```
1. -output_time_series
```

Each access to one key will be stored as one record in the time series file.

If the user is interested to know about the detailed QPS changing during the tracing time, user can specify:

```
1. -output_qps_stats
```

For each query type of one column family, one file with the query number per second will be generated. Also, one file with the QPS of each query type on all column families as well as the overall QPS are output to a separate file. The average QPS and peak QPS will be printed out.

Sometimes, user might be interested to know about the TOP statistics. user can specify

```
1. -print_top_k_access
```

The top K accessed keys, the access number will be printed. Also, if the prefix\_cut option is specified, the top K accessed prefix with their total access count are printed. At the same time, the top K prefix with the highest average access is printed.

If the user is interested to know about the value size distribution (only applicable for Put and Merge ) user can specify

```
1. -output_value_distribution
2. -value_interval
```

Since the value size varies a lot, User might just want to know how many values are in each value size range. User can specify the value\_interval=x to generate the number of values between [0,x), [x,2x).....

The key size distribution is output to the file if user specify

```
1. -output_key_distribution
```

## Visualize the workload

After processing the trace with trace\_analyzer, user will get a couple of output

files. Some of the files can be used to visualize the workload such as the heatmap (the hotness or coldness of keys), time series graph (the overview of the key access in timeline), the QPS of the analyzed queries.

Here, we use the open source plot tool GNUPLOT as an example to generate the graphs. More details about the GNUPLOT can be found here (<http://gnuplot.info/>). User can directly write the GNUPLOT command to draw the graph, or to make it simple, user can use the following shell script to generate the GNUPLOT source file (before using the script, make sure the file name and some of the content are replaced with the effective ones).

To draw the heat map of accessed keys

```

1. #!/bin/bash
2.
3. # The query type
4. ops="iterator"
5.
6. # The column family ID
7. cf="9"
8.
9. # The column family name if known, if not, replace it with some prefix
10. cf_name="rev:cf-assoc-deleter-id1-type"
11.
12. form="accessed"
13.
14. # The column number that will be plotted
15. use="4"
16.
17. # The higher bound of Y-axis
18. y=2
19.
20. # The higher bound of X-axis
21. x=29233
22. echo "set output '${cf_name}-${ops}-${form}-key_heatmap.png'" > plot-${cf_name}-${ops}-${form}-heatmap.gp
23. echo "set term png size 2000,500" >>plot-${cf_name}-${ops}-${form}-heatmap.gp
24. echo "set title 'CF: ${cf_name} ${form} Key Space Heat Map'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
25. echo "set xlabel 'Key Sequence'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
26. echo "set ylabel 'Key access count'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
27. echo "set yrange [0:$y]">>plot-${cf_name}-${ops}-${form}-heatmap.gp
28. echo "set xrange [0:$x]">>plot-${cf_name}-${ops}-${form}-heatmap.gp
29.
30. # If the preifx cut is avialable, it will draw the prefix cut
31. while read f1 f2
32. do
33.     echo "set arrow from $f1,0 to $f1,$y nohead lc rgb 'red'" >> plot-${cf_name}-${ops}-${form}-heatmap.gp
34. done < "trace.1532381594728669-${ops}-${cf}-${form}_key_prefix_cut.txt"
    echo "plot 'trace.1532381594728669-${ops}-${cf}-${form}_key_stats.txt' using ${use} notitle w dots lt 2"
35. >>plot-${cf_name}-${ops}-${form}-heatmap.gp
36. gnuplot plot-${cf_name}-${ops}-${form}-heatmap.gp

```

## To draw the time series map of accessed keys

```

1. #!/bin/bash
2.
3. # The query type
4. ops="iterator"
5.
6. # The higher bound of X-axis
7. x=29233
8.
9. # The column family ID
10. cf="8"
11.
12. # The column family name if known, if not, replace it with some prefix
13. cf_name="rev:cf-assoc-deleter-id1-type"
14.
15. # The type of the output file
16. form="time_series"
17.
18. # The column number that will be plotted
19. use="3:2"
20.
21. # The total time of the tracing duration, in seconds
22. y=88000
23. echo "set output '${cf_name}-${ops}-${form}-key_heatmap.png'" > plot-${cf_name}-${ops}-${form}-heatmap.gp
24. echo "set term png size 3000,3000" >>plot-${cf_name}-${ops}-${form}-heatmap.gp
25. echo "set title 'CF: ${cf_name} time series'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
26. echo "set xlabel 'Key Sequence'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
27. echo "set ylabel 'Key access count'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
28. echo "set yrang [0:$y]">>plot-${cf_name}-${ops}-${form}-heatmap.gp
29. echo "set xrange [0:$x]">>plot-${cf_name}-${ops}-${form}-heatmap.gp
30.
31. # If the preifx cut is avialable, it will draw the prefix cut
32. while read f1 f2
33. do
34.     echo "set arrow from $f1,0 to $f1,$y nohead lc rgb 'red'" >> plot-${cf_name}-${ops}-${form}-heatmap.gp
35. done < "trace.1532381594728669-${ops}-${cf}-accessed_key_prefix_cut.txt"
    echo "plot 'trace.1532381594728669-${ops}-${cf}-${form}.txt' using ${use} notitle w dots lt 2" >>plot-
36. ${cf_name}-${ops}-${form}-heatmap.gp
37. gnuplot plot-${cf_name}-${ops}-${form}-heatmap.gp

```

## To plot out the QPS

```

1. #!/bin/bash
2.
3. # The query type
4. ops="iterator"
5.
6. # The higher bound of the QPS
7. y=5
8.
9. # The column family ID

```

```

10. cf="9"
11.
12. # The column family name if known, if not, replace it with some prefix
13. cf_name="rev:cf-assoc-deleter-id1-type"
14.
15. # The type of the output file
16. form="qps_stats"
17.
18. # The column number that will be plotted
19. use="1"
20.
21. # The total time of the tracing duration, in seconds
22. x=88000
23. echo "set output '${cf_name}-${ops}-${form}-IO_per_second.png'" > plot-${cf_name}-${ops}-${form}-heatmap.gp
24. echo "set term png size 2000,1200" >>plot-${cf_name}-${ops}-${form}-heatmap.gp
25. echo "set title 'CF: ${cf_name} QPS Over Time'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
26. echo "set xlabel 'Time in second'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
27. echo "set ylabel 'QPS'">>plot-${cf_name}-${ops}-${form}-heatmap.gp
28. echo "set yrange [0:$y]">>plot-${cf_name}-${ops}-${form}-heatmap.gp
29. echo "set xrange [0:$x]">>plot-${cf_name}-${ops}-${form}-heatmap.gp
    echo "plot 'trace.1532381594728669-${ops}-${cf}-${form}.txt' using ${use} notitle with linespoints" >>plot-
30. ${cf_name}-${ops}-${form}-heatmap.gp
31. gnuplot plot-${cf_name}-${ops}-${form}-heatmap.gp

```

## Model the Workloads

We can use different tools, script, and models to fit the workload statistics. Typically, user can use the distributions of key access count and key-range access count to fit in the model. Also, QPS can be modeled. Here, we use Matlab as an example to fit the key access count, key-range access count, and QPS. Note that, before we model the key-range hotness, we need to decide how to cut the whole key-space into key-ranges. Our suggestion is that the key-range size can be similar to the number of KV-pairs in a SST file.

User can try different distributions to fit the model. In this example, we use two-term exponential model to fit the access distribution, and two-term sin() to fit the QPS

To fit the key access statistics to the model and get the statistics, user can run the following script:

```

1. % This script is used to fit the key access count distribution
2. % to the two-term exponential distribution and get the parameters
3.
4. % The input file with suffix: accessed_key_stats.txt
5. fileID = fopen('trace.1531329742187378-get-4-accessed_key_stats.txt');
6. txt = textscan(fileID,'%f %f %f %f');
7. fclose(fileID);
8.
9. % Get the number of keys that has access count x

```

```

10. t2=sort(txt{4}, 'descend');
11.
12. % The number of access count that is used to fit the data
13. % The value depends on the accuracy demand of your model fitting
14. % and the value of count should be always not greater than
15. % the size of t2
16. count=30000;
17.
18. % Generate the access count x
19. x=1:1:count;
20. x=x';
21.
22. % Adjust the matrix and uniformed
23. y=t2(1:count);
24. y=y/(sum(y));
25. figure;
26.
27. % fitting the data to the exp2 model
28. f=fit(x,y,'exp2')
29.
30. %plot out the original data and fitted line to compare
31. plot(f,x,y);

```

To fit the key access count distribution to the model, user can run the following script:

```

1. % This script is used to fit the key access count distribution
2. % to the two-term exponential distribution and get the parameters
3.
4. % The input file with suffix: key_count_distribution.txt
5. fileID = fopen('trace-get-9-accessed_key_count_distribution.txt');
6. input = textscan(fileID, '%s %f %s %f');
7. fclose(fileID);
8.
9. % Get the number of keys that has access count x
10. t2=sort(input{4}, 'descend');
11.
12. % The number of access count that is used to fit the data
13. % The value depends on the accuracy demand of your model fitting
14. % and the value of count should be always not greater than
15. % the size of t2
16. count=100;
17.
18. % Generate the access count x
19. x=1:1:count;
20. x=x';
21. y=t2(1:count);
22.
23. % Adjust the matrix and uniformed
24. y=y/(sum(y));
25. y=y(1:count);
26. x=x(1:count);

```

```

27.
28. figure;
29. % fitting the data to the exp2 model
30. f=fit(x,y,'exp2')
31.
32. %plot out the original data and fitted line to compare
33. plot(f,x,y);

```

To fit the key-range average access count to the model, user can run the following script:

```

1. % This script is used to fit the prefix average access count distribution
2. % to the two-term exponential distribution and get the parameters
3.
4. % The input file with suffix: accessed_key_prefix_cut.txt
5. fileID = fopen('trace-get-4-accessed_key_prefix_cut.txt');
6. txt = textscan(fileID, '%f %f %f %f %s');
7. fclose(fileID);
8.
9. % The per key access (average) of each prefix, sorted
10. t2=sort(txt{4}, 'descend');
11.
12. % The number of access count that is used to fit the data
13. % The value depends on the accuracy demand of your model fitting
14. % and the value of count should be always not greater than
15. % the size of t2
16. count=1000;
17.
18. % Generate the access count x
19. x=1:1:count;
20. x=x';
21.
22. % Adjust the matrix and uniformed
23. y=t2(0:count);
24. y=y/(sum(y));
25. x=x(1:count);
26.
27. % fitting the data to the exp2 model
28. figure;
29. f=fit(x,y,'exp2')
30.
31. %plot out the original data and fitted line to compare
32. plot(f,x,y);

```

To fit the QPS to the model, user can run the following script:

```

1. % This script is used to fit the qps of the one query in one of the column
2. % family to the sin'x' model. 'x' can be 1 to 10. With the higher value
3. % of the 'x', you can get more accurate fitting of the qps. However,
4. % the model will be more complex and some times will be overfitted.
5. % The suggestion is to use sin1 or sin2

```

```

6.
7. % The input file shoud with surfix: qps_stats.txt
8. fileID = fopen('trace-get-4-io_stats.txt');
9. txt = textscan(fileID, '%f');
10. fclose(fileID);
11. t1=txt{1};
12.
13. % The input is the queries per second. If you directly use the qps
14. % you may got a high value of noise. Here, 'n' is the number of qps
15. % that you want to combined to one average value, such that you can
16. % reduce it to queries per n*seconds.
17. n=10;
18. s1 = size(t1, 1);
19. M = s1 - mod(s1, n);
20. t2 = reshape(t1(1:M), n, []);
21. y = transpose(sum(t2, 1) / n);
22.
23. % Up to this point, you need to move the data down to the x-axis,
24. % the offset is the ave. So the model will be
25. % s(x) = a1*sin(b1*x+c1) + a2*sin(b2*x+c2) + ave
26. ave = mean(y);
27. y=y-ave;
28.
29. % Adjust the matrix
30. count = size(y,1);
31. x=1:1:count;
32. x=x';
33.
34. % Fit the model to 'sin2' in this example and draw the point and
35. % fitted line to compare
36. figure;
37. s = fit(x,y,'sin2')
38. plot(s,x,y);

```

Users can use the model for further analyzing or use it to generate the synthetic workload.

## Synthetic Workload Generation based on Models

In the previous section, users can use the fitting functions of Matlab to fit the traced workload to different models, such that we can use a set of parameters and functions to profile the workload. We focus on the 4 variables to profile the workload: 1) value size; 2) KV-pair access hotness; 3) QPS; 4) Iterator scan length. According to our current research, the value size and Iterator scan length follows the Generalized Pareto Distribution. The probability density function is:  $f(x) = (1/\sigma)^{(1+k(x-\theta)/\sigma)^{-1-1/k}}$ . The KV-pair access follows power-law, in which about 80% of the KV-pair has an access count less than 4. We sort the keys based on the access count in descending order and fit them to the models. The two-term power model

fit the KV-pair access distribution best. The probability density function is:  $f(x) = ax^b+c$ . The Sine function fits the QPS best.  $F(x) = A\sin(Bx + C) + D$ .

Here is one example of the parameters we get from the workload collected in Facebook social graph:

1. Value Size:  $\sigma = 226.409$ ,  $k = 0.923$ ,  $\theta = 0$
2. KV-pair Access:  $a = 0.001636$ ,  $b = -0.7094$ , and  $c = 3.217 \times 10^{-9}$
3. QPS:  $A = 147.9$ ,  $B = 8.3 \times 10^{-5}$ ,  $C = -1.734$ ,  $D = 1064.2$
4. Iterator scan length:  $\sigma = 1.747$ ,  $k = 0.0819$ ,  $\theta = 0$

We developed a benchmark called “mixgraph” in db\_bench, which can use the four set of parameters to generate the synthetic workload. The workload is statistically similar to the original one. Note that, only the workload that can be fit to the models used for the four variables can be used in the mixgraph. For example, if the value size follows the power distribution instead of Generalized Pareto Distribution, we cannot use the mixgraph to generate the workloads.

More details about the workloads, benchmarks, and models, please refer to the published paper at FAST2020 ([here](#))

To enable the “mixgraph” benchmark, user needs to specify:

```
1. ./db_bench --benchmarks="mixgraph"
```

To set the parameters of the value size distribution (Generalized Pareto Distribution only), user needs to specify:

```
1. -value_k=<> -value_sigma=<> -value_theta=<>
```

To set the parameters of the KV-pair access distribution (power distribution only and  $C=0$ ), user needs to specify:

```
1. -key_dist_a=<> -key_dist_b=<>
```

To set the parameters of key-range distributions which follows the two-term exponential distribution ( $f(x)=a\exp(bx)+c\exp(dx)$ ), user needs to specify both the key-range hotness distribution parameters and the number of key-ranges (keyrange\_num):

```
1. -keyrange_dist_a=<> -keyrange_dist_b=<> -keyrange_dist_c=<> -keyrange_dist_d=<> -keyrange_num=<>
```

To set the parameters of the QPS (Sine), user needs to specify:

```
1. -sine_a=<> -sine_b=<> -sine_c=<> -sine_d=<> -sine_mix_rate_interval_milliseconds=<>
```

The mix rate is used to set the time interval that how long we should correct the rate according to the distribution, the smaller it is, the better it will fit.

To set the parameters of the iterator scan length distribution (Generalized Pareto Distribution only), user needs to specify:

```
1. -iter_k=<> -iter_sigma=<> -iter_theta=<>
```

User need to specify the query ratio between the Get, Put, and Seek. Such that we can generate the mixed workload that can be similar to the social graph workload (so called mix graph). Make sure that the sum of the ratio is 1.

```
1. -mix_get_ratio=<> -mix_put_ratio=<> -mix_seek_ratio=<>
```

Finally, user need to specify how many queries they want to execute:

```
1. -reads=<>
```

and what's the total KV-pairs are in the current testing DB

```
1. -num=<>
```

The num together with the aforementioned distributions decided the queries.

Here is one example that can be directly used to generate the workload which simulate the queries of ZippyDB described in the published paper [link](#). The workloads has 0.42 billion queries and 50 million KV-pairs in total. We use 30 key-ranges. Note that, if user runs the benchmark following the 24 hours Sine period, it will take about 22-24 hours. In order to speedup the benchmarking, user can increase the sine\_d to a larger value such as 45000 to increase the workload intensiveness and also reduce the sine\_b accordingly.

```
./db_bench --benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -
cache_size=268435456 -keyrange_dist_a=14.18 -keyrange_dist_b=-2.917 -keyrange_dist_c=0.0164 -
keyrange_dist_d=-0.08082 -keyrange_num=30 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236
-mix_get_ratio=0.85 -mix_put_ratio=0.14 -mix_seek_ratio=0.01 -sine_mix_rate_interval_milliseconds=5000 -
1. sine_a=1000 -sine_b=0.000073 -sine_d=4500 --perf_level=2 -reads=420000000 -num=50000000 -key_size=48
```

RocksDB may configure a certain amount of main memory as a block cache to accelerate data access. Understanding the efficiency of block cache is very important. The block cache analysis and simulation tools help a user to collect block cache access traces, analyze its access pattern, and evaluate alternative caching policies.

## Table of Contents

- [Quick Start](#)
- [Tracing Block Cache Accesses](#)
- [Trace Format](#)
- [Cache Simulations](#)
  - [RocksDB Cache Simulations](#)
  - [Python Cache Simulations](#)
    - [Supported Cache Simulators](#)
- [Analyzing Block Cache Traces](#)

## Quick Start

---

db\_bench supports tracing block cache accesses. This section demonstrates how to trace accesses when running db\_bench. It also shows how to analyze and evaluate caching policies using the generated trace file.

Create a database:

```
1. ./db_bench --benchmarks="fillseq" \
2. --key_size=20 --prefix_size=20 --keys_per_prefix=0 --value_size=100 \
3. --cache_index_and_filter_blocks --cache_size=1048576 \
4. --disable_auto_compactions=1 --disable_wal=1 --compression_type=none \
5. --min_level_to_compress=-1 --compression_ratio=1 --num=10000000
```

To trace block cache accesses when running `readrandom` benchmark:

```
1. ./db_bench --benchmarks="readrandom" --use_existing_db --duration=60 \
2. --key_size=20 --prefix_size=20 --keys_per_prefix=0 --value_size=100 \
3. --cache_index_and_filter_blocks --cache_size=1048576 \
4. --disable_auto_compactions=1 --disable_wal=1 --compression_type=none \
5. --min_level_to_compress=-1 --compression_ratio=1 --num=10000000 \
6. --threads=16 \
7. -block_cache_trace_file="/tmp/binary_trace_test_example" \
8. -block_cache_trace_max_trace_file_size_in_bytes=1073741824 \
9. -block_cache_trace_sampling_frequency=1
```

Convert the trace file to human readable format:

```
1. ./block_cache_trace_analyzer \
2. -block_cache_trace_path=/tmp/binary_trace_test_example \
3. -human_readable_trace_file_path=/tmp/human_readable_block_trace_test_example
```

Evaluate alternative caching policies:

```
1. bash block_cache_pysim.sh /tmp/human_readable_block_trace_test_example /tmp/sim_results/bench 1 0 30
```

Plot graphs:

```
1. python block_cache_trace_analyzer_plot.py /tmp/sim_results /tmp/sim_results_graphs
```

## Tracing Block Cache Accesses

RocksDB supports block cache tracing APIs `StartBlockCacheTrace` and `EndBlockCacheTrace`. When tracing starts, RocksDB logs detailed information of block cache accesses into a trace file. A user must specify a trace option and trace file path when start tracing block cache accesses.

A trace option contains `max_trace_file_size` and `sampling_frequency`.

- `max_trace_file_size` specifies the maximum size of the trace file. The tracing stops when the trace file size exceeds the specified `max_trace_file_size`.
- `sampling_frequency` determines how frequent should RocksDB trace an access. RocksDB uses spatial downsampling such that it traces all accesses to sampled blocks. A `sampling_frequency` of 1 means tracing all block cache accesses. A `sampling_frequency` of 100 means tracing all accesses on ~1% blocks.

An example to start tracing block cache accesses:

```
1. Env* env = rocksdb::Env::Default();
2. EnvOptions env_options;
3. std::string trace_path = "/tmp/binary_trace_test_example";
4. std::unique_ptr<TraceWriter> trace_writer;
5. DB* db = nullptr;
6. std::string dbname = "/tmp/rocksdb"
7.
8. /*Create the trace file writer*/
9. NewFileTraceWriter(env, env_options, trace_path, &trace_writer);
10. DB::Open(options, dbname, &db);
11.
12. /*Start tracing*/
13. db->StartBlockCacheTrace(trace_opt, std::move(trace_writer));
14.
15. /* Your call of RocksDB APIs */
16.
```

```

17. /*End tracing*/
18. db->EndBlockCacheTrace()

```

# Trace Format

We can convert the generated binary trace file into human readable trace file in csv format. It contains the following columns:

| Column Name                      | Values                                                                                                                 | Comment                                                                                                                                                                                                                                    |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Access timestamp in microseconds | unsigned long                                                                                                          |                                                                                                                                                                                                                                            |
| Block ID                         | unsigned long                                                                                                          | A unique block ID.                                                                                                                                                                                                                         |
| Block type                       | 7: Index block<br>8: Filter block<br>9: Data block<br>10:<br>Uncompressed dictionary block<br>11: Range deletion block |                                                                                                                                                                                                                                            |
| Block size                       | unsigned long                                                                                                          | Block size may be 0 when<br>1) compaction observes cache misses and does not insert the missing blocks into the cache.<br>2) IO error when fetching a block.<br>3) prefetching filter blocks but the SST file does not have filter blocks. |
| Column family ID                 | unsigned long                                                                                                          | A unique column family ID.                                                                                                                                                                                                                 |
| Column family name               | string                                                                                                                 |                                                                                                                                                                                                                                            |
| Level                            | unsigned long                                                                                                          | The LSM tree level of this block.                                                                                                                                                                                                          |
| SST file number                  | unsigned long                                                                                                          | The SST file this block belongs to.                                                                                                                                                                                                        |
| Caller                           | See <a href="#">Caller</a>                                                                                             | The caller that accesses this block, e.g., Get, Iterator, Compaction, etc.                                                                                                                                                                 |
| No insert                        | 0: do not insert the block upon a miss<br>1: insert the block upon a cache miss                                        |                                                                                                                                                                                                                                            |
| Get ID                           | unsigned long                                                                                                          | A unique ID associated with the Get request.                                                                                                                                                                                               |
| Get key ID                       | unsigned long                                                                                                          | The referenced key of the Get request.                                                                                                                                                                                                     |
| Get referenced data size         | unsigned long                                                                                                          | The referenced data (key+value) size of the Get request.                                                                                                                                                                                   |
| Is a cache hit                   | 0: A cache hit<br>1: A cache miss                                                                                      | The running RocksDB instance observes a cache hit/miss on this block.                                                                                                                                                                      |
| Get Does get                     | 0: Does not                                                                                                            | Data block only: Whether the referenced                                                                                                                                                                                                    |

|                                              |                   |                                                                                                |
|----------------------------------------------|-------------------|------------------------------------------------------------------------------------------------|
| referenced key exist in this block           | exist<br>1: Exist | key is found in this block.                                                                    |
| Get Approximate number of keys in this block | unsigned long     | Data block only.                                                                               |
| Get table ID                                 | unsigned long     | The table ID of the Get request. We treat the first four bytes of the Get request as table ID. |
| Get sequence number                          | unsigned long     | The sequence number associated with the Get request.                                           |
| Block key size                               | unsigned long     |                                                                                                |
| Get referenced key size                      | unsigned long     |                                                                                                |
| Block offset in the SST file                 | unsigned long     |                                                                                                |

## Cache Simulations

We support running cache simulators using both RocksDB built-in caches and caching policies written in python. The cache simulator replays the trace and reports the miss ratio given a cache capacity and a caching policy.

## RocksDB Cache Simulations

To replay the trace and evaluate alternative policies, we first need to provide a cache configuration file. An example file contains the following content:

```
1. lru, 0, 0, 16M, 256M, 1G, 2G, 4G, 8G, 12G, 16G, 1T
```

Cache configuration file format:

| Column Name          | Values                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Cache name           | lru: LRU<br>lrupriority: LRU with midpoint insertion<br>lru_hybrid: LRU that also caches row keys<br>ghost*: A ghost cache for admission control. It admits an entry on its second access. <ul style="list-style-type: none"> <li>Specifically, the ghost cache only maintains keys and is managed by LRU.</li> <li>Upon an access, the cache inserts the key into the ghost cache.</li> <li>Upon a cache miss, the cache inserts the missing entry only if it observes a hit in the ghost cache.</li> </ul> |
| Number of shard bits | unsigned long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Ghost cache capacity | unsigned long                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Cache sizes          | A list of comma separated cache sizes                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

Next, we can start simulating caches.

```
1. ./block_cache_trace_analyzer -mrc_only=true \
2. -block_cache_trace_downsample_ratio=100 \
3. -block_cache_trace_path=/tmp/binary_trace_test_example \
4. -block_cache_sim_config_path=/tmp/cache_config \
5. -block_cache_analysis_result_dir=/tmp/binary_trace_test_example_results \
6. -cache_sim_warmup_seconds=3600
```

It contains two important parameters:

`block_cache_trace_downsample_ratio` : The sampling frequency used to collect the trace. The simulator scales down the given cache size by this factor. For example, with `downsample_ratio` of 100, the cache simulator creates a 1 GB cache to simulate a 100 GB cache.

`cache_sim_warmup_seconds` : The number of seconds used for warmup. The reported miss ratio does NOT include the number of misses/accesses during the warmup.

The analyzer outputs a few files:

- A miss ratio curve file: `{trace_duration_in_seconds}_{total_accesses}_mrc`.
- Three miss ratio timeline files per second (1), per minute (60), and per hour (3600).
- Three number of misses timeline files per second (1), per minute (60), and per hour (3600).

## Python Cache Simulations

We also support a more diverse set of caching policies written in python. In addition to LRU, it provides replacement policies using reinforcement learning, cost class, and more. To use the python cache simulator, we need to first convert the binary trace file into human readable trace file.

```
1. ./block_cache_trace_analyzer \
2. -block_cache_trace_path=/tmp/binary_trace_test_example \
3. -human_readable_trace_file_path=/tmp/human_readable_block_trace_test_example
```

`block_cache_pysim.py` options:

- 1) `Cache` type (`ts`, `linucb`, `arc`, `lru`, `opt`, `pylru`, `pymru`, `pylfu`, `pyhb`, `gdszie`, `trace`).
- 2) One may evaluate the hybrid row\_block cache by appending '`_hybrid`' to a `cache_type`, e.g., `ts_hybrid`.
- 3) Note that hybrid is not supported with `opt` and `trace`.
- 4) `Cache` size (`xM`, `xG`, `XT`).
- 5) `3)` The sampling frequency used to collect the trace.
- 6) (The simulation scales down the cache size by the sampling frequency).
- 7) `4)` Warmup seconds (The number of seconds used for warmup).

9. 6) Result directory (A directory that saves generated results)
10. 7) Max number of accesses to process. (Replay the entire trace if set to -1.)
11. 8) The target column family. (The simulation will only run accesses on the target column family.)
12. If it is set to all, it will run against all accesses.)

One example:

```
python block_cache_pysim.py lru 16M 100 3600 /tmp/human_readable_block_trace_test_example /tmp/results
1. 10000000 0 all
```

We also provide a bash script to simulate a batch of cache configurations:

```
1. Usage: ./block_cache_pysim.sh trace_file_path result_dir downsample_size warmup_seconds max_jobs
2.
3. -max_jobs: The maximum number of simulators to run at a time.
```

One example:

```
1. bash block_cache_pysim.sh /tmp/human_readable_block_trace_test_example /tmp/sim_results/bench 1 0 30
```

block\_cache\_pysim.py output the following files:

- A miss ratio curve file: `data-ml-mrc-{cache_type}-{cache_size}-{target_cf_name}`.
- Two files on the timeline of miss ratios per minute (60), and per hour (3600).
- Two files on the timeline of number of misses per second (1), per minute (60), and per hour (3600). For `ts` and `linucb`, it also outputs the following files:
- Two files on the timeline of percentage of times a policy is selected: per minute (60) and per hour (3600).
- Two files on the timeline of number of times a policy is selected: per minute (60) and per hour (3600).

block\_cache\_pysim.sh combines the outputs of block\_cache\_pysim.py into following files:

- One miss ratio curve file per target column family: `ml_{target_cf_name}_mrc`
- One files on the timeline of number of misses per `{target_cf_name}{capacity}{time_unit}` : `ml_{target_cf_name}{capacity}{time_unit}miss_timeline`
- One files on the timeline of miss ratios per `{target_cf_name}{capacity}{time_unit}` : `ml_{target_cf_name}{capacity}{time_unit}miss_ratio_timeline`
- One files on the timeline of number of times a policy is selected per `{target_cf_name}{capacity}{time_unit}` : `ml_{target_cf_name}{capacity}{time_unit}policy_timeline`
- One files on the timeline of percentage of times a policy is selected per `{target_cf_name}{capacity}{time_unit}` : `ml_{target_cf_name}{capacity}{time_unit}policy_ratio_timeline`

## Supported Cache Simulators

| Cache Name | Comment |
|------------|---------|
|------------|---------|

|          |                                                                                                                                                                                                                                                                                                                                                                                      |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| lru      | Strict (Least recently used) LRU cache. The cache maintains an LRU queue.                                                                                                                                                                                                                                                                                                            |
| gdszie   | GreedyDual Size.<br>N. Young. The k-server dual and loose competitiveness for paging. <i>Algorithmica</i> , June 1994, vol. 11,(no.6):525-41. Rewritten version of ''On-line caching as cache size varies'', in The 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 241-250, 1991.                                                                                             |
| opt      | The Belady MIN algorithm.<br>L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. <i>IBM Syst. J.</i> 5, 2 (June 1966), 78-101.<br>DOI= <a href="http://dx.doi.org/10.1147/sj.52.0078">http://dx.doi.org/10.1147/sj.52.0078</a>                                                                                                                     |
| arc      | Adaptive replacement cache.<br>Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03). USENIX Association, Berkeley, CA, USA, 115-130.                                                                                                            |
| pylru    | LRU cache with random sampling.                                                                                                                                                                                                                                                                                                                                                      |
| pymru    | (Most recently used) MRU cache with random sampling.                                                                                                                                                                                                                                                                                                                                 |
| pylfu    | (Least frequently used) LFU cache with random sampling.                                                                                                                                                                                                                                                                                                                              |
| pyhb     | Hyperbolic Caching.<br>Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic caching: flexible caching for web applications. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17). USENIX Association, Berkeley, CA, USA, 499-511.                                                                              |
| pyccbt   | Cost class: block type                                                                                                                                                                                                                                                                                                                                                               |
| pyccfbt  | Cost class: column family + block type                                                                                                                                                                                                                                                                                                                                               |
| ts       | Thompson sampling<br>Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. A Tutorial on Thompson Sampling. <i>Found. Trends Mach. Learn.</i> 11, 1 (July 2018), 1-96. DOI: <a href="https://doi.org/10.1561/2200000070">https://doi.org/10.1561/2200000070</a>                                                                                       |
| linucb   | Linear UCB<br>Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A contextual-bandit approach to personalized news article recommendation. In Proceedings of the 19th international conference on World wide web (WWW '10). ACM, New York, NY, USA, 661-670.<br>DOI= <a href="http://dx.doi.org/10.1145/1772690.1772758">http://dx.doi.org/10.1145/1772690.1772758</a> |
| trace    | Trace                                                                                                                                                                                                                                                                                                                                                                                |
| *_hybrid | A hybrid cache that also caches row keys.                                                                                                                                                                                                                                                                                                                                            |

`py*` caches use random sampling at eviction time. It samples 64 random entries in the cache, sorts these entries based on a priority function, e.g., LRU, and evicts from the lowest priority entry until the cache has enough capacity to insert the new entry.

`pycc*` caches group cached entries by a cost class. The cache maintains aggregated statistics for each cost class such as number of hits, total size. A cached entry is also tagged with one cost class. At eviction time, the cache samples 64 random entries and group them by their cost class. It then evicts entries based on their cost class's statistics.

`ts` and `linucb` are two caches using reinforcement learning. The cache is configured

`ts` and `linucb` are two caches using reinforcement learning. The cache is configured with N policies, e.g., LRU, MRU, LFU, etc. The cache learns which policy is the best overtime and selects the best policy for eviction. The cache rewards the selected policy if the policy has not evicted the missing key before. `ts` does not use any feature of a block while `linucb` uses three features: a block's level, column family, and block type.

`trace` reports the misses observed in the collected trace.

## Analyzing Block Cache Traces

The `block_cache_trace_analyzer` analyzes a trace file and outputs useful statistics of the access pattern. It provides insights into how to tune and improve a caching policy.

The `block_cache_trace_analyzer` may output statistics into multiple csv files saved in a result directory. We can plot graphs on these statistics using

`block_cache_trace_analyzer_plot.py`.

Analyzer options:

```

1. -access_count_buckets (Group number of blocks by their access count given
2. these buckets. If specified, the analyzer will output a detailed analysis
3. on the number of blocks grouped by their access count break down by block
4. type and column family.) type: string default: ""
5. -analyze_blocks_reuse_k_reuse_window (Analyze the percentage of blocks that
6. are accessed in the [k, 2*k] seconds are accessed again in the next [2*k,
7. 3*k], [3*k, 4*k], ..., [k*(n-1), k*n] seconds.) type: int32 default: 0
8. -analyze_bottom_k_access_count_blocks (Print out detailed access
9. information for blocks with their number of accesses are the bottom k
10. among all blocks.) type: int32 default: 0
11. -analyze_callers (The list of callers to perform a detailed analysis on. If
12. speicfied, the analyzer will output a detailed percentage of accesses for
13. each caller break down by column family, level, and block type. A list of
14. available callers are: Get, MultiGet, Iterator, ApproximateSize,
15. VerifyChecksum, SSTDumpTool, ExternalSSTIngestion, Repair, Prefetch,
16. Compaction, CompactionRefill, Flush, SSTFileReader, Uncategorized.)
17. type: string default: ""
18. -analyze_correlation_coefficients_labels (Analyze the correlation
19. coefficients of features such as number of past accesses with regard to
20. the number of accesses till the next access.) type: string default: ""
21. -analyze_correlation_coefficients_max_number_of_values (The maximum number
22. of values for a feature. If the number of values for a feature is larger
23. than this max, it randomly selects 'max' number of values.) type: int32
24. default: 1000000
25. -analyze_get_spatial_locality_buckets (Group data blocks by their
26. statistics using these buckets.) type: string default: ""
27. -analyze_get_spatial_locality_labels (Group data blocks using these
28. labels.) type: string default: ""
29. -analyze_top_k_access_count_blocks (Print out detailed access information
30. for blocks with their number of accesses are the top k among all blocks.)
```

```

32. -block_cache_analysis_result_dir (The directory that saves block cache
33. analysis results.) type: string default: ""
34. -block_cache_sim_config_path (The config file path. One cache configuration
35. per line. The format of a cache configuration is
36. cache_name,num_shard_bits,ghost_capacity,cache_capacity_1,...,cache_capacity_N.
37. Supported cache names are lru, lru_priority, lru_hybrid. User may also add
38. a prefix 'ghost_' to a cache_name to add a ghost cache in front of the real
39. cache. ghost_capacity and cache_capacity can be xK, xM or xG where
40. x is a positive number.)
41. type: string default: ""
42. -block_cache_trace_downsample_ratio (The trace collected accesses on one in
43. every block_cache_trace_downsample_ratio blocks. We scale down the
44. simulated cache size by this ratio.) type: int32 default: 1
45. -block_cache_trace_path (The trace file path.) type: string default: ""
46. -cache_sim_warmup_seconds (The number of seconds to warmup simulated
47. caches. The hit/miss counters are reset after the warmup completes.)
48. type: int32 default: 0
49. -human_readable_trace_file_path (The file path that saves human readable
50. access records.) type: string default: ""
51. -mrc_only (Evaluate alternative cache policies only. When this flag is
52. true, the analyzer does NOT maintain states of each block in memory for
53. analysis. It only feeds the accesses into the cache simulators.)
54. type: bool default: false
55. -print_access_count_stats (Print access count distribution and the
56. distribution break down by block type and column family.) type: bool
57. default: false
58. -print_block_size_stats (Print block size distribution and the distribution
59. break down by block type and column family.) type: bool default: false
60. -print_data_block_access_count_stats (Print data block accesses by user Get
61. and Multi-Get.) type: bool default: false
62. -reuse_distance_buckets (Group blocks by their reuse distances given these
63. buckets. For example, if 'reuse_distance_buckets' is '1K,1M,1G', we will
64. create four buckets. The first three buckets contain the number of blocks
65. with reuse distance less than 1KB, between 1K and 1M, between 1M and 1G,
66. respectively. The last bucket contains the number of blocks with reuse
67. distance larger than 1G. ) type: string default: ""
68. -reuse_distance_labels (Group the reuse distance of a block using these
69. labels. Reuse distance is defined as the cumulated size of unique blocks
70. read between two consecutive accesses on the same block.) type: string
71. default: ""
72. -reuse_interval_buckets (Group blocks by their reuse interval given these
73. buckets. For example, if 'reuse_interval_buckets' is '1,10,100', we will
74. create four buckets. The first three buckets contain the number of blocks
75. with reuse interval less than 1 second, between 1 second and 10 seconds,
76. between 10 seconds and 100 seconds, respectively. The last bucket
77. contains the number of blocks with reuse interval longer than 100
78. seconds.) type: string default: ""
79. -reuse_interval_labels (Group the reuse interval of a block using these
80. labels. Reuse interval is defined as the time between two consecutive
81. accesses on the same block.) type: string default: ""
82. -reuse_lifetime_buckets (Group blocks by their reuse lifetime given these
83. buckets. For example, if 'reuse_lifetime_buckets' is '1,10,100', we will
84. create four buckets. The first three buckets contain the number of blocks

```

```

85.   with reuse lifetime less than 1 second, between 1 second and 10 seconds,
86.   between 10 seconds and 100 seconds, respectively. The last bucket
87.   contains the number of blocks with reuse lifetime longer than 100
88.   seconds.) type: string default: ""
89. -reuse_lifetime_labels (Group the reuse lifetime of a block using these
90.   labels. Reuse lifetime is defined as the time interval between the first
91.   access on a block and the last access on the same block. For blocks that
92.   are only accessed once, its lifetime is set to kMaxUint64.) type: string
93.   default: ""
94. -skew_buckets (Group the skew labels using these buckets.) type: string
95.   default: ""
96. -skew_labels (Group the access count of a block using these labels.)
97.   type: string default: ""
98. -timeline_labels (Group the number of accesses per block per second using
99.   these labels. Possible labels are a combination of the following: cf
100.  (column family), sst, level, bt (block type), caller, block. For example,
101.  label "cf_bt" means the number of access per second is grouped by unique
102.  pairs of "cf_bt". A label "all" contains the aggregated number of
103.  accesses per second across all possible labels.) type: string default: ""

```

An example that outputs a statistics summary of the access pattern:

```
1. ./block_cache_trace_analyzer -block_cache_trace_path=/tmp/test_trace_file_path
```

Another example:

```

1. ./block_cache_trace_analyzer \
2. -block_cache_trace_path=/tmp/test_trace_file_path \
3. -block_cache_analysis_result_dir=/tmp/sim_results/test_trace_results \
4. -print_block_size_stats \
5. -print_access_count_stats \
6. -print_data_block_access_count_stats \
7. -timeline_labels=cf,level,bt,caller \
8. -analyze_callers=Get,Iterator,Compaction \
9. -access_count_buckets=1,2,3,4,5,6,7,8,9,10,100,1000,10000,100000 \
10. -analyze_bottom_k_access_count_blocks=10 \
11. -analyze_top_k_access_count_blocks=10 \
12. -reuse_lifetime_labels=cf,level,bt \
13. -reuse_lifetime_buckets=1,10,100,1000,10000,100000,1000000 \
14. -reuse_interval_labels=cf,level,bt,caller \
15. -reuse_interval_buckets=1,10,100,1000,10000,100000,1000000 \
16. -analyze_blocks_reuse_k_reuse_window=3600 \
17. -analyze_get_spatial_locality_labels=cf,level,all \
18. -analyze_get_spatial_locality_buckets=10,20,30,40,50,60,70,80,90,100,101 \
19. -analyze_correlation_coefficients_labels=all,cf,level,bt,caller \
20. -skew_labels=block,bt,table,sst,cf,level \
21. -skew_buckets=10,20,30,40,50,60,70,80,90,100

```

Next, we can plot graphs using the following command:

```
1. python block_cache_trace_analyzer_plot.py /tmp/sim_results /tmp/sim_results_graphs
```

```
1. python block_cache_trace_analyzer_plot.py /tmp/sim_results /tmp/sim_results_graphs
```

- Delete Stale Files
- Partitioned Index/Filters
- WritePrepared-Transactions
- WriteUnprepared-Transactions
- How we keep track of live SST files
- How we index SST
- Merge Operator Implementation
- RocksDB Repairer
- Two Phase Commit
- Iterator's Implementation
- Simulation Cache
- Persistent Read Cache
- DeleteRange Implementation
- `unordered_write`

In this wiki we explain how files are deleted if they are not needed.

## SST Files

When compaction finishes, the input SST files are replaced by the output ones in the LSM-tree. However, they may not qualify for being deleted immediately. Ongoing operations depending the older version of the LSM-tree will prevent those files from being qualified to be dropped, until those operations finish. See [How we keep track of live SST files](#) for how the versioning of LSM-tree works.

The operations which can hold an older version of LSM-tree include:

1. Live iterators. Iterators pin the version of LSM-tree while they are created. All SST files from this version are prevented from being deleted. This is because an iterator reads data from a virtual snapshot, all SST files at the moment the iterator are preserved for data from the virtual snapshot to be available.
2. Ongoing compaction. Even if other compactions aren't compacting those SST files, the whole version of the LSM-tree is pinned.
3. Short window during Get(). In the short time a Get() is executed, the LSM-tree version is pinned to make sure it can read all the immutable SST files. When no operation pins an old LSM-tree version containing an SST file anymore, the file is qualified to be deleted.

Those qualified files are deleted by two mechanisms:

## Reference counting

RocksDB keeps a reference count for each SST file in memory. Each version of the LSM-tree holds one reference count for all the SST files in this version. The operations depending on this version (explained above) in turn hold reference count to the version of the LSM-tree directly or indirectly through "super version". Once the reference count of a version drops to 0, it drops the reference counts for all SST files in it. If a file's SST reference count drops to 0, it can be deleted. Usually they are deleted immediately, with following exceptions:

1. The file is found to be not needed when closing an iterator. If users set `ReadOptions.background_purge_on_iterator_cleanup=true`, rather than deleting the file immediately, we schedule a background job to the high-pri thread pool (the same pool where **flush jobs** are deleted) to delete the file.
2. In Get() or some other operations, if it dereference one version of LSM-tree and cause some SST files to be stale. Rather than having those files to be deleted, they are saved. The next flush job will clean it up, or if some other SST files are being deleted by another thread, these files will be deleted together. In this way, we make sure in Get() no file deletion I/O is made. Be aware that, if no flush happens, the stale files may remain there to be deleted.
3. If users have called `DB::DisableFileDeletions()`. All files to be deleted will be

hold. Once a `DB::EnableFileDeletions()` clears the file deletion restriction, it will delete all the SST files pending to be deleted.

## Listing all files to find stale files

---

Reference counting mechanism works for most of the case. However, reference count is not persistent so it is lost after DB restarts, so that we need another mechanism to garbage collect files. We do this full garbage collection when restarting the DB, and periodically based on `options.background_purge_on_iterator_cleanup`. The later case is just to be safe.

In this full garbage collection mode, we list all the files in the DB directory, and check whether each file against all the live versions of LSM-trees and see whether the file is in use. For files not needed, we delete them. However, not all the SST files in the DB directory not in live LSM-tree is stale. Files being created for an ongoing compaction or flush should not be removed. To prevent it from happening, we take use of a good feature that new files are created using the file name of an incremental number. Before each flush or compaction job runs, we remember the number of latest SST file created at that time. If a full garbage collection runs before the job finishes, all SST files with number larger than that will be kept. The condition is released after the job is finished. Since multiple flush and compaction jobs can run in parallel, all SST files with number larger than number the earliest ongoing job remembers will be kept. It possible that we have some false positive, but they will be cleared up eventually.

## Log Files

---

A Log file is qualified to be deleted if all the data in it has been flushed to SST files. Determining an log file can qualify to be deleted is very straight-forward for single column family DBs, slightly more complicated for multi-column family DBs, and even more complicated when two-phase-commit (2PC) is enabled.

## DB With Single column family

---

A log file has a 1:1 mapping with a memtable. Once a memtable is flushed, the respective log file will be deleted. This is done in the end of the flush job.

## DB With Multiple Column Families

---

When there are multiple column families, a new log file is created when memtable for any column family is flushed. One log file can only be deleted when the data in it for all column families has been flushed to SST files. The way RocksDB implements it is for each column family to keep track of the earliest log file that still contains unflushed data for this column family. A log file can only be deleted if it is earlier

than the earliest of the earliest log for all the column families.

## Two-Phase-Commit

In Two-Phase-Commit (2PC) case, there are two log entries for one write: one prepare and one commit. Only when committed data is flushed to memory, we can release logs containing prepare or commit to be deleted. There isn't a clear mapping between memtable and log files any more. For example, considering this sequence for one single column family DB:

```
1. -----
2. 001.log
3.   Prepare Tx1 Write (K1, V1)
4.   Prepare Tx2 Write (K2, V2)
5.   Commit Tx1
6.   Prepare Tx3 Write (K3, V3)
7. ----- <= Memtable Flush    <<<< Point A
8. 002.log
9.   Commit Tx2
10.  Prepare Tx4 Write (K4, V4)
11. ----- <= Memtable Flush   <<<< Point B
12. 003.log
13.   Commit Tx3
14.   Prepare Tx5 Write (K5, V5)
15. ----- <= Memtable Flush   <<<< Point C
```

In *Point A*, although the memtable is flushed, 001.log is not qualified to be deleted, because Tx2 and Tx3 are not committed yet. Similarly, in *Point B*, 0001.log still isn't qualified to be deleted, because Tx3 is not yet committed. Only in *Point C*, 001.log can be deleted. But 002.log still can't be deleted because of Tx4.

RocksDB use an intricate low-lock data structure to determine a log file is qualified to be deleted or not.

As DB/mem ratio gets larger, the memory footprint of filter/index blocks becomes non-trivial. Although `cache_index_and_filter_blocks` allows storing only a subset of them in block cache, their relatively large size negatively affects the performance by i) occupying the block cache space that could otherwise be used for caching data, ii) increasing the load on the disk storage by loading them into the cache after a miss. Here we illustrate these problems in more detail and explain how partitioning index/filters alleviates the overhead.

## How large are the index/filter blocks?

---

RocksDB has by default one index/filter block per SST file. The size of the index/filter block varies based on the configuration but for an SST of size 256MB the index/filter block of size 0.5/5MB is typical, which is much larger than the typical data block size of 4-32KB. That is fine when all index/filter blocks fit perfectly into memory and hence are read once per SST lifetime, not so much when they compete with data blocks for the block cache space and are also likely to be re-read many times from the disk.

## What is the big deal with large index/filter blocks?

---

When index/filter blocks are stored in block cache they are effectively competing with data blocks (as well as with each other) on this scarce resource. A filter of size 5MB is occupying the space that could otherwise be used to cache 1000s of data blocks (of size 4KB). This would result in more cache misses for data blocks. The large index/filter blocks also kick each other out of the block cache more often and exacerbate their own cache miss rate too. This is while only a small part of the index/filter block might have been actually used during its lifetime in the cache.

After the cache miss of an index/filter block, it has to be reloaded from the disk, and its large size is not helping in reducing the IO cost. While a simple point lookup might need at most a couple of data block reads (of size 4KB) one from each layer of LSM, it might end up also loading multiple megabytes of index/filter blocks. If that happens often then the disk is spending more time serving index/filter blocks rather than the actual data blocks.

## What is partitioned index/filters?

---

With partitioning, the index/filter block of an SST file is partitioned into smaller blocks with an additional top-level index on them. When reading an index/filter, only top-level index is loaded into memory. The partitioned index/filter then uses the top-level index to load on demand into the block cache the partitions that are required to perform the index/filter query. The top-level index, which has much smaller memory footprint, can be stored in heap or block cache depending on the

```
cache_index_and_filter_blocks      setting.
```

## Pros:

- Higher cache hit rate: Instead of polluting the cache space with large index/blocks, partitioning allows loading index/filters with much finer granularity and hence making effective use of the cache space.
- Less IO util: Upon a cache miss for an index/filter partition, only one partition requires to be loaded from the disk, which results into much lighter load on disk compared to reading the entire index/filter of the SST file.
- No compromise on index/filters: Without partitioning the alternative approach to reduce memory footprint of index/filters is to sacrifice their accuracy by for example larger data blocks or fewer bloom bits to have smaller index and filters respectively.

## Cons:

- Additional space for the top-level index: its quite small 0.1-1% of index/filter size.
- More disk IO: if the top-level index is not already in cache it would result to one additional IO. To avoid that they can be either stored in heap or stored in cache with hi priority (TODO work)
- Losing spatial locality: if a workload requires frequent, yet random reads from the same SST file, it would result into loading a separate index/filter partition upon each read, which is less efficient than reading the entire index/filter at once. Although we did not observe this pattern in our benchmarks, it is only likely to happen for L0/L1 layers of LSM, for which partitioning can be disabled (TODO work)

## Success stories

### HDD, 100TB DB

In this example, we have a DB of size 86G on HDD and emulate the small memory that is present to a node with 100TB of data by using direct IO (skipping OS file cache) and a very small block cache of size 60MB. Partitioning improves throughput by 11x from 5 op/s to 55 op/s.

```
./db_bench --benchmarks="readwhilewriting[X3],stats" --use_direct_reads=1 --compaction_readahead_size 1048576 --
use_existing_db --num=2000000000 --duration 600 --cache_size=62914560 --cache_index_and_filter_blocks=false --statistics
--histogram --bloom_bits=10 --target_file_size_base=268435456 --block_size=32768 --threads 32 --partition_filters
--partition_indexes --index_per_partition 100 --pin_l0_filter_and_index_blocks_in_cache --benchmark_write_rate_limit
204800 --max_bytes_for_level_base 134217728 --cache_high_pri_pool_ratio 0.9
```

### SSD, Linkbench

In this example, we have a DB of size 300G on SSD and emulate the small memory that would be available in presence of other DBs on the same node by using direct IO (skipping OS file cache) and block cache of size 6G and 2G. Without partitioning the linkbench throughput drops from 38k tps to 23k when reducing memory from 6G to 2G. With partitioning the throughput drops from 38k to only 30k.

## How to use it?

---

- `index_type = IndexType::kTwoLevelIndexSearch`
  - This is to enable partitioned indexes.
- `NewBloomFilterPolicy(BITS, false)`
  - Use full filters.
- `partition_filters = true`
  - This is to enable partitioned filters.
- `metadata_block_size = 4096`
  - This is the block size for index partitions.
- `cache_index_and_filter_blocks = false` [if you are on <= 5.14]
  - The partitions are stored in the block cache anyway. This is to control the location of top-level indexes (which easily fit into memory): pinned in heap or cached in the block cache. Having them stored in block cache is less experimented with.
- `cache_index_and_filter_blocks = true` and `pin_top_level_index_and_filter = true` [if you are on >= 5.15]
  - This would put everything in block cache but also pin the top-level indexes, which are quite small.
- `cache_index_and_filter_blocks_with_high_priority = true`
  - Recommended setting.
- `pin_l0_filter_and_index_blocks_in_cache = true`
  - Recommended setting as this property is extended to the index/filter partitions as well.
  - Use it only if the compaction style is level-based.
  - **Note:** with pinning blocks into the block cache, it could potentially go beyond the capacity if strict\_capacity\_limit is not set (which is the default case).
- block cache size: if you used to store the filter/index into heap, do not forget to increase the block cache size with the amount of memory that you are saving from the heap.

## Current limitations

---

- Partitioned filters cannot be enabled without having partitioned index enabled as well.
- We have the same number of filter and index partitions. In other words, whenever an index block is cut, the filter block is cut as well. We might change it in future if it shows to be causing deficiencies.

- The filter block size is determined by when the index block is cut. We will soon extend `metadata_block_size` to enforce the maximum size on both filter and index blocks, i.e., a filter block is cut either when an index block is cut or when its size is about to exceed `metadata_block_size` (TODO).

## Under the hood

Here we present the implementation details for the developers.

## BlockBasedTable Format

You can study the BlockBasedTable format [here](#). With partitioning the difference would be that the index block

```
1. [index block]
```

is stored as

```
1. [index block - partition 1]
2. [index block - partition 2]
3. ...
4. [index block - partition N]
5. [index block - top-level index]
```

and the footer of SST points to the top-level index block (which by itself is an index on index partition blocks). Each individual index block partition conforms the same format as kBinarySearch. The top-level index format also conforms with that of kBinarySearch and hence can be read using the normal data block readers.

Similar structure is used for partitioning the filter blocks. The format of each individual filter block conforms with that of kFullFilter. The top-level index format conforms with that of kBinarySearch, similar to top-level index of index blocks.

Note that with partitioning the SST inspection tools such `sst_dump` report the size of top-level index on index/filters rather than the collective size of index/filter blocks.

## Builder

Partitioned index and filters are built by `PartitionedIndexBuilder` and `PartitionedFilterBlockBuilder` respectively.

`PartitionedIndexBuilder` maintains `sub_index_builder_`, a pointer to `ShortenedIndexBuilder`, to build the current index partition. When determined by `flush_policy_`, the builder saves the pointer along with the last key in the index block, and creates a new active

`ShortenedIndexBuilder`. When `::Finish` is called on the builder, it calls `::Finish` on the earliest sub index builder and returns the resulting partition block. Next calls to `PartitionedIndexBuilder::Finish` will also include the offset of previously returned partition on the SST, which is used as the values of the top-level index. The last call to `PartitionedIndexBuilder::Finish` will finish the top-level index and return that instead. After storing the top-level index on SST, its offset will be used as the offset of the index block.

`PartitionedFilterBlockBuilder` inherits from `FullFilterBlockBuilder` which has a `FilterBitsBuilder` for building bloom filters. It also has a pointer to `PartitionedIndexBuilder` and invokes `ShouldCutFilterBlock` on it to determine when a filter block should be cut (right after when an index block is cut). To cut a filter block, it finishes the `FilterBitsBuilder` and stores the resulting block along with a partitioning key provided by `PartitionedIndexBuilder::GetPartitionKey()`, and reset the `FilterBitsBuilder` for the next partition. At the end each time `PartitionedFilterBlockBuilder::Finish` is invoked one of the partitions is returned, and also the offset of the previous partition is used to build the top-level index. The last call to `::Finish` will return the top-level index block.

The reasons for making `PartitionedFilterBlockBuilder` depend on `PartitionedIndexBuilder` was to enable an optimization for interleaving index/filter partitions on the SST file. That optimization not being pursued we are likely to cut this dependency in future.

## Reader

---

Partitioned indexes are read via `PartitionIndexReader` which operates on the top-level index block. When `NewIterator` is invoked a `TwoLevelIterator` on the the top-level index block. This simple implementation is feasible since each index partition has `kBinarySearch` format which is the same format as data blocks, and thus can be easily plugged as the lower level iterator. If `pin_l0_filter_and_index_blocks_in_cache` is set, the lower level iterators are pinned to `PartitionIndexReader`, so their corresponding index partitions will be pinned in block cache as long as `PartitionIndexReader` is alive. `BlockEntryIteratorState` uses a set of the pinned partition offsets to avoid unpinning an index partition twice.

`PartitionedFilterBlockReader` uses the top-level index to find the offset of the filter partition. It then invokes `GetFilter` on `BlockBasedTable` object to load the `FilterBlockReader` object on the filter partition from the block cache (or load it to the cache if it is not already there) and then releases the `FilterBlockReader` object. To extend `table_options.pin_l0_filter_and_index_blocks_in_cache` to filter partitions, `PartitionedFilterBlockReader` does not release the cache handles for such blocks (i.e., keep them pinned in block cache). It instead maintains `filter_cache_`, a map of pinned `FilterBlockReader`, which is also used to release the cache entries when `PartitionedFilterBlockReader` is destructed.

RocksDB supports both optimistic and pessimistic concurrency controls. The pessimistic transactions make use of locks to provide isolation between the transactions. The default write policy in pessimistic transactions is *WriteCommitted*, which means that the data is written to the DB, i.e., the memtable, only after the transaction is committed. This policy simplified the implementation but came with some limitations in throughput, transaction size, and variety in supported isolation levels. In the below, we explain these in detail and present the other write policies, *WritePrepared* and *WriteUnprepared*. We then dive into the design of *WritePrepared* transactions.

## *WriteCommitted*, Pros and Cons

---

With *WriteCommitted* write policy, the data is written to the memtable only after the transaction commits. This greatly simplifies the read path as any data that is read by other transactions can be assumed to be committed. This write policy, however, implies that the writes are buffered in memory in the meanwhile. This makes memory a bottleneck for large transactions. The delay of the commit phase in 2PC (two-phase commit) also becomes noticeable since most of the work, i.e., writing to memtable, is done at the commit phase. When the commit of multiple transactions are done in a serial fashion, such as in 2PC implementation of MySQL, the lengthy commit latency becomes a major contributor to lower throughput. Moreover this write policy cannot provide weaker isolation levels, such as READ UNCOMMITTED, that could potentially provide higher throughput for some applications.

## Alternatives: *WritePrepared* and *WriteUnprepared*

---

To tackle the lengthy commit issue, we should do memtable writes at earlier phases of 2PC so that the commit phase become lightweight and fast. 2PC is composed of Write stage, where the transaction `::Put` is invoked, the prepare phase, where `::Prepare` is invoked (upon which the DB promises to commit the transaction if later is requested), and commit phase, where `::Commit` is invoked and the transaction writes become visible to all readers. To make the commit phase lightweight, the memtable write could be done at either `::Prepare` or `::Put` stages, resulting into *WritePrepared* and *WriteUnprepared* write policies respectively. The downside is that when another transaction is reading data, it would need a way to tell apart which data is committed, and if they are, whether they are committed before the transaction's start, i.e., in the read snapshot of the transaction. *WritePrepared* would still have the issue of buffering the data, which makes the memory the bottleneck for large transactions. It however provides a good milestone for transitioning from *WriteCommitted* to *WriteUnprepared* write policy. Here we explain the design of *WritePrepared* policy. The changes that make the design to also support *WriteUnprepared* can be found [here](#).

# *WritePrepared* in a nutshell

These are the primary design questions that needs to be addressed:

1. How do we identify the key/values in the DB with transactions that wrote them?
2. How do we figure if a key/value written by transaction Txn\_w is in the read snapshot of the reading transaction Txn\_r?
3. How do we rollback the data written by aborted transactions?

With *WritePrepared*, a transaction still buffers the writes in a write batch object in memory. When 2PC `::Prepare` is called, it writes the in-memory write batch to the WAL (write-ahead log) as well as to the memtable(s) (one memtable per column family); We reuse the existing notion of sequence numbers in RocksDB to tag all the key/values in the same write batch with the same sequence number, `prepare_seq`, which is also used as the identifier for the transaction. At commit time, it writes a commit marker to the WAL, whose sequence number, `commit_seq`, will be used as the commit timestamp of the transaction. Before releasing the commit sequence number to the readers, it stores a mapping from `prepare_seq` to `commit_seq` in an in-memory data structure that we call *CommitCache*. When a transaction reading values from the DB (tagged with `prepare_seq`) it makes use of the *CommitCache* to figure if `commit_seq` of the value is in its read snapshot. To rollback an aborted transaction, we apply the status before the transaction by making another write that cancels out the writes of the aborted transaction.

The *CommitCache* is a lock-free data structure that caches the recent commit entries. Looking up the entries in the cache must be enough for almost all the transactions that commit in a timely manner. When evicting the older entries from the cache, it still maintains some other data structures to cover the corner cases for transactions that take abnormally too long to finish. We will cover them in the design details below.

## *WritePrepared* Design

Here we present the design details for *WritePrepared* transactions. We start by presenting the efficient design for *CommitCache*, and dive into other data structures as we see their importance to guarantee the correctness on top of *CommitCache*.

### *CommitCache*

The question of whether a data is committed or not is mainly about very recent transactions. In other words, given a proper rollback algorithm in place, we can assume any old data in the DB is committed and also is present in the snapshot of a reading transaction, which is mostly a recent snapshot. Leveraging this observation, maintaining a cache of recent commit entries must be sufficient for most of the cases. *CommitCache* is an efficient data structure that we designed for this purpose.

*CommitCache* is a fixed-size, in-memory array of commit entries. To update the cache with a commit entry, we first index the `prepare_seq` with the array size and then rewrite the corresponding entry in the array, i.e.: `CommitCache[ prepare_seq % array_size ] = < prepare_seq , commit_seq >`. Each insertion will result into eviction of the previous value, which results into updating `max_evicted_seq`, the maximum evicted sequence number from *CommitCache*. When looking up in the *CommitCache*, if a `prepare_seq > max_evicted_seq` and yet not in the cache, then it is considered as not committed. If the entry is otherwise found in the cache, then it is committed and will be read by the transaction with snapshot sequence number `snap_seq` if `commit_seq <= snap_seq`. If `prepare_seq < max_evicted_seq`, then we are reading an old data, which is most likely committed unless proven otherwise, which we explain below how.

Given 80K tps (transactions per second) of writes, 8M entries in the commit cache (hardcoded in `TransactionDBOptions::wp_commit_cache_bits`), and having sequence numbers increased by two per transaction, it would take roughly 50 seconds for an inserted entry to be evicted from the *CommitCache*. In practice however the delay between prepare and commit is a fraction of a millisecond and this limit is thus not likely to be met. For the sake of correctness however we need to cover the cases where a prepared transaction is not committed by the time `max_evicted_seq` advances its `prepare_seq`, as otherwise the reading transactions would assume it is committed. To do so, we maintain a heap of prepare sequence numbers called *PreparedHeap*: a `prepare_seq` is inserted upon `::Prepare` and removed upon `::Commit`. When `max_evicted_seq` advances, if it becomes larger than the minimum `prepare_seq` in the heap, we pop such entries and store them in a set called `delayed_prepared`. Verifying that `delayed_prepared` is empty is an efficient operation which needs to be done before calling an old `prepare_seq` as committed. Otherwise, the reading transactions should also look into `delayed_prepared` to see if the `prepare_seq` of the values that they read is found there. Let us emphasize that such cases is not expected to happen in a reasonable setup and hence not negatively affecting the performance.

Although for read-write transactions, they are expected to commit in fractions of a millisecond after the `::Prepare` phase, it is still possible for a few read-only transactions to hang on some very old snapshots. This is the case for example when a transaction takes a backup of the DB, which could take hours to finish. Such read-only transactions cannot assume an old data to be in their reading snapshot, since their snapshot could also be quite old. More precisely, we still need to keep an evicted entry `< prepare_seq , commit_seq >` around if there is a live snapshot with sequence number `snap_seq` where `prepare_seq <= snap_seq < commit_seq`. In such cases we add the `prepare_seq` to `old_commit_map`, a mapping from snapshot sequence number to a set of `prepare_seq`. The only transactions that would have to pay the cost of looking into this data structure are the ones that are reading from a very old snapshot. The size of this data structure is expected to be very small as i) there are only a few transactions doing backups and ii) there are limited number of concurrent transactions that overlap with their reading snapshot. `old_commit_map` is garbage collected lazily when the DB is informed of the snapshot release via its periodic fetch of snapshot list as part of the procedure for advancing `max_evicted_seq`.

## PreparedHeap

As its name suggests, *PreparedHeap* is a heap of prepare sequence numbers: a `prepare_seq` is inserted upon `::Prepare` and removed upon `::Commit`. The heap structure allows efficient check of the minimum `prepare_seq` against `max_evicted_seq` as was explained above. To allow efficient removal of entries from the heap, the actual removal is delayed until the entry reaches the top of the heap, i.e., becomes the minimum. To do so, the removed entry will be added to another heap if it is not already on top. Upon each change, the top of the two heaps are compared to see if the top of the main heap is tagged for removal.

## Rollback

To rollback an aborted transaction, for each written key/value we write another key/value to cancel out the previous write. Thanks to write-write conflict avoidance done via locks in pessimistic transactions, it is guaranteed that only one pending write will be on each key, meaning that we only need to look at the previous value to figure the state to which we should rollback. If the result of `::Get` from a snapshot with sequence number `kMaxSequenceNumber`, is a normal value (the to-be-rolled-back data will be skipped since they are not committed) then we do a `::Put` on that key with that value, and if it is a non-existent value, we insert a `::Delete` entry. All the values (written by aborted transaction as well as by the rollback step) then commit with the same `commit_seq`, and the `prepare_seq` of aborted transaction and of the rollback batch is removed from *PreparedHeap*.

## Atomic Commit

During a commit, a commit marker is written to the WAL and also a commit entry is added to the *CommitCache*. These two needs to be done atomically otherwise a reading transaction at one point might miss the update into the *CommitCache* but later sees that. We achieve that by updating the *CommitCache* before publishing the sequence number of the commit entry. In this way, if a reading snapshot can see the commit sequence number it is guaranteed that the *CommitCache* is already updated as well. This is done via a `PreReleaseCallback` that is added to `::WriteImpl` logic for this purpose.

`PreReleaseCallback` is also used to add `prepare_seq` to *PreparedHeap* so that its top always represents the smallest uncommitted transaction. (Refer to *Smallest Uncommitted Section* to see how this is used).

When we have two write queues (`two_write_queues \= true`) then the primary write queue can write to both WAL and memtable and the 2nd one can write only to the WAL, which will be used for writing the commit marker in `WritePrepared` transactions. In this case the primary queue (and its `PreReleaseCallback` callback) is always used for prepare entires and the 2nd queue (and its `PreReleaseCallback` callback) is always used only for commits. This i) avoids race condition between the two queues, ii) maintains the in-order addition to *PreparedHeap*, and iii) simplifies the code by avoiding concurrent

insertion to `CommitCache` (and the code that is called upon each eviction from it).

Since the last sequence number could advance by either queue while the other is not done with the reserved lower sequence number, this could raise an atomicity issue. To address that we introduce the notion of last published sequence number, which will be used when taking a snapshot. When we have one write queue, this is the same as the last sequence number and when we have two write queues this is the last committed entry (performed by the 2nd queue). This restriction penalizes non-2PC transactions by splitting them to two steps: i) write to memtable via the primary queue, ii) commit and publish the sequence number via the 2nd queue.

## IsInSnapshot

`IsInSnapshot(prepare_seq, snapshot_seq)` implements the core algorithm of `WritePrepared`, which puts all the data structures together to determine if a value tagged with `prepare_seq` is in the reading snapshot `snapshot_seq`.

Here is the simplified version of `IsInSnapshot` algorithm:

```

1. inline bool IsInSnapshot(uint64_t prep_seq, uint64_t snapshot_seq,
2.                         uint64_t min_uncommitted = 0,
3.                         bool *snap_released = nullptr) const {
4.     if (snapshot_seq < prep_seq) return false;
5.     if (prep_seq < min_uncommitted) return true;
6.     max_evicted_seq_ub = max_evicted_seq_.load();
7.     some_are_delayed = delayed_prepared_ not empty
8.     if (prep_seq in CommitCache) return CommitCache[prep_seq] <= snapshot_seq;
9.     if (max_evicted_seq_ub < prep_seq) return false; // still prepared
10.    if (some_are_delayed) {
11.        ...
12.    }
13.    if (max_evicted_seq_ub < snapshot_seq) return true; // old commit with no overlap with snapshot_seq
14.    // commit is old so is the snapshot, check if there was an overlap
15.    if (snaoshot_seq not in old_commit_map_) {
16.        *snap_released = true;
17.        return true;
18.    }
19.    bool overlapped = prepare_seq in old_commit_map_[snaoshot_seq];
20.    return !overlapped;
21. }
```

It returns true if it can determine that `commit_seq`  $\leq$  `snapshot_seq` and false otherwise.

- `snapshot_seq < prep_seq => commit_seq > snapshot_seq` because `prep_seq <= commit_seq`
- `prep_seq < min_uncommitted => commit_seq <= snapshot_seq`
- Checking emptiness of `delayed_prepared_` in `some_are_delayed` before `CommitCache` lookup is an optimization to skip acquiring lock on it if there is no delayed transaction in the system, as it is the norm.

- If not in `CommitCache` and none of the delayed prepared cases apply, then this is an old commit that is evicted from `CommitCache`.
  - `max_evicted_seq_ < snapshot_seq => commit_seq < snapshot_seq` since `commit_seq <= max_evicted_seq_`
  - Otherwise, `old_commit_map_` includes all such old snapshots as well as any commit that overlaps with them.

In the below we see the full implementation of `IsInSnapshot` that also covers the corner cases. `IsInSnapshot(prepare_seq, snapshot_seq)` implements the core algorithm of `WritePrepared`, which puts all the data structures together to determine if a value tagged with `prepare_seq` is in the reading snapshot `snapshot_seq`.

```

1. inline bool IsInSnapshot(uint64_t prep_seq, uint64_t snapshot_seq,
2.                           uint64_t min_uncommitted = 0,
3.                           bool *snap_released = nullptr) const {
4.     if (snapshot_seq < prep_seq) return false;
5.     if (prep_seq < min_uncommitted) return true;
6.     do {
7.         max_evicted_seq_lb = max_evicted_seq_.load();
8.         some_are_delayed = delayed_prepared_ not empty
9.         if (prep_seq in CommitCache) return CommitCache[prep_seq] <= snapshot_seq;
10.        max_evicted_seq_ub = max_evicted_seq_.load();
11.        if (max_evicted_seq_lb != max_evicted_seq_ub) continue;
12.        if (max_evicted_seq_ub < prep_seq) return false; // still prepared
13.        if (some_are_delayed) {
14.            if (prep_seq in delayed_prepared_) {
15.                // might be committed but not added to commit cache yet
16.                if (prep_seq not in delayed_prepared_commits_) return false;
17.                return delayed_prepared_commits_[prep_seq] < snapshot_seq;
18.            } else {
19.                // 2nd probe due to non-atomic commit cache and delayed_prepared_
20.                if (prep_seq in CommitCache) return CommitCache[prep_seq] <= snapshot_seq;
21.                max_evicted_seq_ub = max_evicted_seq_.load();
22.            }
23.        }
24.    } while (UNLIKELY(max_evicted_seq_lb != max_evicted_seq_ub));
25.    if (max_evicted_seq_ub < snapshot_seq) return true; // old commit with no overlap with snapshot_seq
26.    // commit is old so is the snapshot, check if there was an overlap
27.    if (snaoshot_seq not in old_commit_map_) {
28.        *snap_released = true;
29.        return true;
30.    }
31.    bool overlapped = prepare_seq in old_commit_map_[snaoshot_seq];
32.    return !overlapped;
33. }
```

- Since `max_evicted_seq_` and `CommitCache` are updated separately, the while loop simplifies the algorithm by ensuring that `max_evicted_seq_` is not changed during `CommitCache` lookup.
- The commit of a delayed prepared involves four non-atomic steps: i) update

- `CommitCache` ii) add to `delayed_prepared_commits_`, iii) publish sequence, and iv)  
remove from `delayed_prepared_`.
- If the reader simply follows `CommitCache` lookup + `delayed_prepared_` lookup order, it might find a delayed prepared in neither and miss checking against its `commit_seq`. So address that if the sequence was not found in `delayed_prepared_`, it still does a 2nd lookup in `CommitCache`. The reverse order ensures that it will see the commit if there was any.
  - There are odd scenarios where the commit of a delayed prepared could be evicted from commit cache before the entry is removed from `delayed_prepared_` list. `delayed_prepared_commits_` which is updated every time a delayed prepared is evicted from commit cache helps not to miss such commits.

## Flush/Compaction

---

Flush/Compaction threads, similarly to reading transaction, make use of `IsInSnapshot` API to figure which versions can be safely garbage collected without affecting the live snapshots. The difference is that a snapshot might be already released by the time the compaction is calling `IsInSnapshot`. To address that, if `IsInSnapshot` is extended with `snap_released` argument so that if it could not confidently give a true/false response, it will signal the caller that the `snapshot_seq` is no longer valid.

## Duplicate keys

---

WritePrepared writes all data of the same write batch with the same sequence number. This is assuming that there is no duplicate key in the write batch. To be able to handle duplicate keys, we divide a write batch to multiple sub-batches, one after each duplicate key. The memtable returns false if it receives a key with the same sequence number. The memtable inserter then advances the sequence number and tries again.

The limitation with this approach is that the write process needs to know the number of sub-batches beforehand so that it could allocate the sequence numbers for each write accordingly. When using transaction API this is done cheaply via the index of `WriteBatchWithIndex` as it already has mechanisms to detect duplicate insertions. When calling `::CommitBatch` to write a batch directly to the DB, however, the DB has to pay the cost of iterating over the write batch and count the number of sub-batches. This would result into a non-negligible overhead if there are many of such writes.

Detecting duplicate keys requires knowing the comparator of the column family (cf). If a cf is dropped, we need to first make sure that the WAL does not have an entry belonging to that cf. Otherwise if the DB crashes afterwards the recovery process will see the entry but does not have the comparator to tell whether it is a duplicate.

## Optimizations

Here we cover the additional details in the design that were critical in achieving good performance.

## Lock-free *CommitCache*

Most reads from recent data result into a lookup into *CommitCache*. It is therefore vital to make *CommitCache* efficient for reads. Using a fixed array was already a conscious design decision to serve this purpose. We however need to further avoid the overhead of synchronization for reading from this array. To achieve this, we make *CommitCache* an array of `std::atomic<uint64_t>` and encode `< prepare_seq , commit_seq >` into the available 64 bits. The reads and writes from the array are done with `std::memory_order_acquire` and `std::memory_order_release` respectively. In an `x86_64` architecture these operations are translated into simple reads and writes into memory thanks to the guarantees of the hardware cache coherency protocol. We have other designs for this data structure and will explore them in future.

To encode `< prepare_seq , commit_seq >` into 64 bits we use this algorithm: i) the higher bits of `prepare_seq` is already implied by the index of the entry in the *CommitCache*; ii) the lower bits of `prepare seq` are encoded in the higher bits of the 64-bit entry; iii) the difference between the `commit_seq` and `prepare_seq` is encoded into the lower bits of the 64-bit entry.

## Less-frequent updates to `max_evicted_seq`

Normally `max_evicted_seq` is expected to be updated upon each eviction from the *CommitCache*. Although updating `max_evicted_seq` is not necessarily expensive, the maintenance operations that come with it are. For example, it requires holding a mutex to verify the top in *PreparedHeap* (although this can be optimized to be done without a mutex). More importantly, it involves holding the db mutex for fetching the list of live snapshots from the DB since maintaining `old_commit_map` depends on the list of live snapshots up to `max_evicted_seq`. To reduce this overhead, upon each update to `max_evicted_seq` we increase its value further by 1% of the *CommitCache* size (if it does not exceed last publish sequence number), so that the maintenance is done 100 times before *CommitCache* array wraps around rather than once per eviction.

## Lock-free Snapshot List

In the above, we mentioned that a few read-only transactions doing backups are expected at each point of time. Each evicted entry, which is expected upon each insert, therefore needs to be checked against the list of live snapshots (which was taken when `max_evicted_seq` was last advanced). Since this is done frequently, it needs to be done efficiently, without holding any mutex. We therefore design a data structure that lets us perform this check in a lock-free manner. To do so, we store the first S snapshots (S hardcoded in `TransactionDBOptions::wp_snaoshot_cache_bits` to

be 128) in an array of `std::atomic<uint64_t>`, which we know is efficient for reads and write on x86\_64 architecture. The single writer updates the array with a list of snapshots sorted in ascending order by starting from index 0, and updates the size in an atomic variable.

We need to guarantee that the concurrent reader will be able to read all the snapshots that are still valid after the update. Both new and old lists are sorted and the new list is a subset of the previous list plus some new items. Thus if a snapshot repeats in both new and old lists, it will appear with a lower index in the new list. So if we simply insert the new snapshots in order, if an overwritten item is still valid in the new list, it is either written to the same place in the array or it is written in a place with a lower index before it gets overwritten by another item. This guarantees a reader that reads the array from the other side will eventually see a snapshot that repeats in the update, either before it gets overwritten by the writer or afterwards.

If the number of snapshots exceed the array size, the remaining updates will be stored in a vector protected by a mutex. This is just to ensure correctness in the corner cases and is not expected to happen in a normal run.

## Smallest Uncommitted

---

We keep track of smallest uncommitted data and store it in the snapshot. When reading the data if its sequence number lower than the smallest uncommitted data, then we skip lookup into CommitCache to reduce cpu cache misses. The smallest entry in

`delayed_prepared_` will represent the *Smallest UnCommitted* if it is not empty. Otherwise, which is almost always the case, the *PreparedHeap* top represents the smallest uncommitted thanks to the rule of adding entires to it in ascending order by doing so only via the primary write queue. *Smallest UnCommitted* will also be used to limit *ValidateSnapshot* search to memtables when it knows that *Smallest UnCommitted* is already larger than smallest sequence in the memtables.

## rocksdb\_commit\_time\_batch\_for\_recovery

---

As explained above, we have specified that commits are only done via the 2nd queue. When the commit is accompanied with `CommitTimeWriteBatch` however, then it would write to memtable too, which essentially would send the batch to the primary queue. In this case we do two separate writes to finish the commit: one writing `CommitTimeWriteBatch` via the main queue, and two committing that as well as the prepared batch via the 2nd queue. To avoid this overhead the users can set `rocksdb_commit_time_batch_for_recovery` configuration variable to true which tells RocksDB that such data will only be required during recovery. RocksDB benefits from that by writing the `CommitTimeWriteBatch` only to the WAL. It still keeps the last copy around in memory to write it to the SST file after each flush. When using this option `CommitTimeWriteBatch` cannot have duplicate entries since we do not want to pay the cost of counting sub-batches upon each commit request.

# Experimental Results

Here is a summary of improvements on some sysbench benchmarks as well as linkbench (done via MyRocks).

- benchmark.....tps.....p95 latency...cpu/query
- insert.....68%
- update-noindex...30%....38%
- update-index.....61%.....28%
- read-write.....6%.....3.5%
- read-only.....-1.2%....-1.8%
- linkbench.....1.9%.....+overall.....0.6%

Here are also the detailed results for [In-Memory Sysbench](#) and [SSD Sysbench](#) courtesy of [@mdcallag](#).

## Current Limitations

There is ~1% overhead for read workloads. This is due to the extra work that needs to be done to tell apart uncommitted data from committed ones.

The rollback of merge operands is currently disabled. This is to hack around a problem in MyRocks that does not lock the key before using the merge operand on it.

Currently `Iterator::Refresh` is not supported. There is no fundamental obstacle and it can be added upon request.

Although the DB generated by `WritePrepared` policy is backward/forward compatible with the classic `WriteCommitted` policy, the WAL format is not. Therefore to change the `WritePolicy` the WAL has to be emptied first by flushing the DB.

Non-2PC transactions will result into two writes if `two_write_queues` is enabled. If majority of transactions are non-2PC, the `two_write_queues` optimization should be disabled.

`TransactionDB::Write` incurs additional cost of detecting duplicate keys in the write batch.

If a column family is dropped, the WAL has to be cleaned up beforehand so that there would be no entry belonging to that CF in the WAL.

When using `rocksdb_commit_time_batch_for_recovery`, the data passed to `CommitTimeWriteBatch` cannot have duplicate keys and will be visible only after a memtable flush.

This document presents the initial design for moving memtable writes from the prepare phase to unprepared phase when the write batch is still being written to.

*WriteUnprepared* are to be announced as production-ready soon.

## Goals

These are the goals of the project:

1. Reduce memory footprint, which would also enable handling very large transactions.
2. Avoid write stalls caused by large transactions writing to DB at once.

## Synopsis

Transactions are currently buffered in memory until the prepare phase of 2PC. When the transaction is large, so is the buffered data, and writing such large buffered data to DB at once negatively affects the concurrent transactions. Moreover, buffering the entire data of very large transactions can run the server out of memory. Top contributors to memory footprint are i) the buffered key/values, i.e., the write batch, ii) the per-key locks. In this design we divide the project to three stages reducing the memory usage of i) values, ii) keys, iii) locks in them respectively.

To eliminate the memory usage of buffered values in large transaction, unprepared data will be gradually batched into unprepared batches and written into the database while the write batch is still being built. The keys are still buffered though to simplify the rollback algorithm. When a transaction commits, the commit cache is updated with an entry per unprepared batch. The main challenges are handling the rollback and read-your-own-writes.

Transactions need to be able to read their own uncommitted data. Previously this was done by looking into the buffered data before looking into the DB. We address this challenge by keeping the track of the sequence numbers of the written unprepared batches to the disk, and augment ReadCallback to return true if the sequence number of the read data matches them. This applies only to large transactions that do not have the write batch buffered in memory.

The current rollback algorithm in WritePrepared can work only after recovery when there is no live snapshots. In WriteUnPrepared however transactions can rollback in presence of live snapshots. We redesign the rollback algorithm to i) append to the transaction the prior values of the keys that are modified by the transaction, hence essentially cancelling the writes, ii) commit the rolled back transaction. Treating a rollback transaction as committed greatly simplified the implementation as the existing mechanisms of handling live snapshots with committing transaction can seamless apply. WAL will still include a rollback marker to enable the recovery procedure to reattempt the rollback should the DB crash in the middle. To determine

the set of prior values, the list of modified keys must be tracked in the Transaction object. In the absence of the buffered write batch, in the first stage of the project we still buffer the modified key set. In the 2nd stage, we retrieve the key set from the WAL when the transaction is large.

RocksDB tracks the list of keys locked in a transaction in TransactionLockMgr. For large transactions, the list of locked keys will not fit in memory. Automatic lock escalation into range locks can be used to approximate a set of point locks. When RocksDB detects that a transaction is locking a large number keys within a certain range, it will automatically upgrade it to a range lock. We will tackle this in the 3rd stage of the project. More details in Section “Key Locking” below.

## Stages

---

The plan is to execute this project in 3 stages:

1. Implement unprepared batches, but keys are still buffered in memory for rollback and for key-locking.
2. Use the WAL to obtain the key set during rollback instead of buffering in memory.
3. Implement range lock mechanism for large transactions.

## Implementation

---

### WritePrepared Overview

---

In the existing WritePrepared policy, the data structures are:

- PrepareHeap: A heap of in-progress prepare sequence numbers.
- CommitCache: a mapping between prepare sequence number to commit seq.
  - max\_evicted\_seq\_: the largest evicted sequence number from the commit cache.
- OldCommitMap: list of evicted entries from the commit cache if they are still invisible to some live snapshots.
- DelayedPrepared: list of prepared sequence numbers from PrepareHeap that are less than max\_evicted\_seq\_.

## Put

---

The writes into the database will need to be batched to avoid the overhead of going through write queues. To avoid confusion with “write batches”, these will be called “unprepared batches”. By batching, we also save on the number of unprepared sequence numbers that we have to generate and track in our data structures. Once a batch reaches a configurable threshold, it’ll be written as if it were a prepare operation in WritePreparedTxn, except that a new WAL type called BeginUnprepareXID will be used as opposed to BeginPersistedPrepareXID used by the WritePreparedTxn policy. All the

keys in the same unprepared batch will get the same sequence number (unless there is a duplicate key, which would divide the batch into multiple sub-batches).

Proposed WAL format: [BeginUnprepare(XID)]...[EndPrepare(XID)]

This implies that we'll need to know the XID of the transaction (where XID is a name unique during lifetime of the transaction) before final prepare (this is true for MyRocks, since we generate the XID of the transaction when it begins).

The Transaction object will need to track the list of unprepared batches that have written to the db. To do this, the Transaction object will contain a set of unprep\_seq numbers, and when an unprepared batch is written, the unprep\_seq is added to the set.

On unprepared batch write, the unprep\_seq number is also added to the unprepared heap (similar to prepare heap in WritePreparedTxn).

## Prepare

---

On prepare we will write out the remaining entries in the current write batch to the WAL but with using the BeginPersistedPrepare(XID) marker instead to denote that the transaction is now prepared. This is needed so that on crash, we can return to the application the list of prepared transactions so that the application can perform the correct action. Unprepared transactions will be implicitly rolled back on recovery.

Proposed WAL format: [BeginUnprepare(XID)]...[EndPrepare(XID)] ... [BeginUnprepare(XID)]...[EndPrepare(XID)] ... [BeginPersistedPrepare(XID)]...[EndPrepare(XID)] ... ...[Commit(XID)]

In this case, the final prepare gets BeginPersistedPrepare(XID) instead of BeginUnprepare(XID) to denote that the transaction has been truly prepared.

Note that although the DB (sst files) are backward- and forward-compatible between WritePreparedTxn and WriteUnpreparedTxn, the WAL of WriteUnpreparedTxn is not forward-compatible with that of WritePreparedTxn: WritePreparedTxn will successfully fail on recovering from WAL files generated by WriteUnpreparedTxn because of the new marker types. However, WriteUnpreparedTxn is still fully backward compatible with WritePreparedTxn and will be able to read WritePreparedTxn WAL files, since they will look the same.

## Commit

---

On commit, the commit map and unprepared heap need to be updated. For WriteUnprepared, a commit will potentially have multiple prepare sequence numbers associated with it. All (unprep\_seq, commit\_seq) pairs must be added to the commit map and all unprep\_seq must be removed from the unprepare\_heap.

If commit is performed without a prepare and the transaction has not previously written unprepared batches, then the current unprepared batch of writes will be

written out directly similar to CommitWithoutPrepareInternal in the WritePreparedTxn case. If the transaction has already written unprepared batches, then an implicit prepare phase is added.

## Rollback

In WritePreparedTxn, rollback implementation was limited to only rollbacks right after recovery. It was implemented like this:

1. prep\_seq = seq at which the prepared data was written
2. For each modified key, read the original values at prep\_seq - 1
3. Write the original values back but at a new sequence number, rollback\_seq
4. rollback\_seq is added to commit map
5. prep\_seq is then removed from PrepareHeap.

This implementation would not work if there are live snapshots to which prep\_seq is visible. This is because if max\_evicted\_seq advances beyond prep\_seq, we could have  $\text{prep\_seq} < \text{max\_evicted\_seq} < \text{snapshot\_seq} < \text{rollback\_seq}$ . Then, the transaction reading at snapshot\_seq would assume data at prep\_seq were committed since  $\text{prep\_seq} < \text{max\_evicted\_seq}$  and yet it is not recorded in old\_commit\_map.

This shortcoming in WritePreparedTxn was tolerated because MySQL would only rollback prepared transactions on recovery, where there would be no live snapshots to observe this inconsistency. In WriteUnpreparedTxn, however, this scenario occurs not just on recovery, but also on user-initiated rollbacks of unprepared transactions.

We fix the issue by writing a rollback marker, appending the rollback data to the aborted transaction, and then committing the transaction in the commit map. Since the transaction is appended with rolled back data, although committed, it does not change the state of db, and hence is effectively rolled back. If max\_evicted\_seq advances beyond prep\_seq, since the  $\langle \text{prep\_seq}, \text{commit\_seq} \rangle$  is added to the CommitCache, the existing procedure, i.e, adding evicted entry to old\_commit\_map, will take care of live snapshots with  $\text{prep\_seq} < \text{snapshot\_seq} < \text{commit\_seq}$ . If it crashed in the middle of rollback, on recovery it reads the rollback marker and finishes up the rollback process.

If the DB crashes in the middle of rollback, the recovery will see some partially written rolled back data in the WAL. Since the recovery process will eventually reattempt the rollback, such partial data will be simply overwritten with the new rolled back batch containing the same prior values.

The rollback batch could be written either at once or divided into multiple sub-patches in the case of a very large transaction. We will explore this option further during the implementation.

The other issue with rollbacks in WriteUnpreparedTxn is knowing which keys to rollback. Previously, since the whole prepare batch was buffered in memory, it was possible to just iterate over the write batch to find the set of modified keys that

need to be rolled back. In the first iteration of the project, we still keep the write key-set in the memory. In the next iteration, if the size of the key-set goes beyond a threshold, we purge the key set from memory and retrieve it from the WAL should the transaction aborts later. Every transaction already tracks a list of unprepared sequence numbers, and that can be used to seek into the correct position in the WAL.

## Get

---

The read path is largely the same as WritePreparedTxn. The only difference is for transactions being able to read its own writes. Currently, GetFromBatchAndDB handles this by first checking the write batch first before fetching from the DB where ReadCallback is called to determine visibility. In the absence of write batch we need another mechanism to handle this.

Recall that every transaction maintains a list of unprep\_seq. Before entering the main visibility logic as described in WritePreparedTxn, check if the key has a sequence number that exists in the set of unprep\_seq. If it is present, then the key is visible. This logic happens in ReadCallback which currently does not take a set of sequence numbers, but this can be extended so that the set of unprep\_seq can be passed down.

Currently, Get and Seek will seek directly to the sequence number specified by the snapshot when reading from DB, so uncommitted data written by the same transaction is potentially skipped before visibility logic can even be applied. To resolve this issue, this optimization would have to be removed if the current transaction is large enough to have its buffered write batch removed and instead written to the DB as unprepared batches.

## Recovery

---

Recovery will work similarly to WritePreparedTxn except with a few changes to how we determine transaction state (unprepared, prepared, aborted, committed).

During recovery, unprepared batches with the same XID must be tracked until the EndPrepare marker is observed. If recovery finishes without EndPrepare, then the transaction is unprepared and the equivalent of an application rollback is implicitly done.

If recovery ends with EndPrepare, but there is no commit marker, then the transaction is prepared, and is presented to the application.

If a rollback marker is found after EndPrepare but there is no commit marker, then the transaction is aborted, and the recovery process must overwrite the aborted data with their prior values.

If a commit marker is found, then the transaction is committed.

# Delayed Prepared

Large transactions will probably have long duration as well. If a transaction is not committed after a long time (1 min for a 100 kTPS workload), its sequence number is moved to DelayedPrepared, which is currently a simple set protected by a lock. If it turns out that the current implementation is a bottleneck, we will change DelayedPrepared from a set into a sharded hash table, similar to how transaction key locking is done. If it works well enough for key locking (which occurs more frequently), it should work fine for tracking prepared transactions.

## Key Locking

Currently, RocksDB supports point locks via a sharded hashtable in TransactionLockMgr. Every time a lock is requested, the key is hashed and a lookup is done to see if an existing lock with the same key exists. If so, the thread blocks, otherwise, the lock is granted and inserted into the hashtable. This implies that all locked keys will reside in memory, which may be problematic for large transactions.

To mitigate this problem, range locks can be used to approximate the large set of point locks when a transaction is detected to have locked many keys within a range. Here we present a preliminary approach to tackle this problem to show its feasibility. When reaching this stage of the project we will reconsider alternative designs, and/or whether the parallel Gap Locking has already eliminated the problem.

To support range locks, the key space will need to be partitioned in N logical partitions, where every partition represents a contiguous range in the key space. A partition key will represent every partition and can be calculated from the key itself through a callback provided by the application. A partition key is automatically write-locked if the number of locked keys in that partition goes beyond a threshold; in that case the individual keys are released.

```
A set will be used to hold the set of all partitions. A partition will have the following struct:
struct Partition {
    map<TransactionID, int> txn_locks;
    enum { UNLOCKED, LOCKED, LOCK_REQUEST } status = UNLOCKED;
    std::shared_ptr<std::mutex> part_mutex;
    std::shared_ptr<std::condition_variable> part_cv;
    int waiters = 0;
};
```

When a lock for a key is requested:

1. Lookup the corresponding partition structure. If it does not exist, then create and insert it. If status is UNLOCKED, then increment with `txn_locks[id]++`, otherwise increment waiters and block on `part_cv`. Decrement waiters and repeat this step when the thread is woken.
2. Request a point lock on the point lock hashtable.
  - i. If the point lock is granted, then check if the threshold is exceeded by looking at `txn_locks[id]`.
  - ii. If point lock times out, decrement with `txn_locks[id]-`.

To upgrade a lock:

1. Set partition status to LOCK\_REQUEST, increment waiters, and block on part\_cv until txn\_locks only contains the current transaction. Decrement waiters and recheck txn\_locks when woken.
2. Set status to LOCKED.
3. Remove all point locks in that partition from the point lock hashtable. The point locks to remove can be determined by tracked\_keys\_.
4. Update tracked\_keys\_ to remove all point locks in that partition and add the partition lock in tracked\_keys\_.

To unlock a point lock:

1. Remove the point lock from the point lock hashtable.
2. Decrement txn\_locks in the corresponding partition.
3. Signal on part\_cv. if waiters is nonzero. Otherwise remove the partition.

To unlock a partition lock (no user API to trigger this, but this happens when transactions end):

1. Signal on part\_cv if waiters is nonzero. Otherwise, remove the partition.

Note that any time data from partition is being read, its mutex part\_mutex must be held.

In RocksDB, the LSM tree consists of a list of SST files in the file system, besides WAL logs. After each compaction, compaction output files are added to the list while the input ones are removed from it. However, input files do not necessarily qualify to be deleted instantly, because some `get` or outstanding iterators might require the files to be kept around until their operations finished or iterators freed. In the rest of the page, we introduce how we keep this information.

The list of files in an LSM tree is kept in a data structure called `version`. In the end of a compaction or a mem table flush, a new `version` is created for the updated LSM tree. At one time, there is only one “current” `version` that represents the files in the up-to-date LSM tree. New `get` requests or new iterators will use the current `version` through the whole read process or life cycle of iterator. All `version`s used by `get` or iterators need to be kept. An out-of-date `version` that is not used by any `get` or iterator needs to be dropped. All files not used by any other version need to be deleted. For example,

If we start with a `version` with three files:

1. `v1={f1, f2, f3}` (current)
2. files on disk: `f1, f2, f3`

and now an iterator is created with it:

1. `v1={f1, f2, f3}` (current, used by iterator1)
2. files on disk: `f1, f2, f3`

Now a flush happens added `f4`, a new version is created:

1. `v2={f1, f2, f3, f4}` (current)
2. `v1={f1, f2, f3}` (used by iterator1)
3. files on disk: `f1, f2, f3, f4`

Now a compaction happened compact `f2`, `f3` and `f4` into a new file `f5` with a new `version` `v3` created:

1. `v3={f1, f5}` (current)
2. `v2={f1, f2, f3, f4}`
3. `v1={f1, f2, f3}` (used by iterator1)
4. files on disk: `f1, f2, f3, f4, f5`

Now `v2` is neither up-to-date, nor used by anyone, so it qualifies to be removed, together with `f4`. While `v1` still cannot be removed for it is still needed by `iterator1`:

1. `v3={f1, f5}` (current)
2. `v1={f1, f2, f3}` (used by iterator1)
3. files on disk: `f1, f2, f3, f5`

Assuming now `iterator1` is destroyed:

1. `v3={f1, f5}` (current)
2. `v1={f1, f2, f3}`
3. files on disk: `f1, f2, f3, f5`

Now `v1` is not used nor up-to-date, so it can be removed, with file `f2`, `f3`:

1. `v3={f1, f5}` (current)
2. files on disk: `f1, f5`

This logic is implemented using reference counts. Both of an SST file and a `version` have a reference count. While we create a `version`, we incremented the reference counts for all files. If a `version` is not needed, all files' of the version have their reference counts decremented. If a file's reference count drops to 0, the file can be deleted.

In a similar way, each `version` has a reference count. When a `version` is created, it is an up-to-date one, so it has reference count 1. If the `version` is not up-to-date anymore, its reference count is decremented. Anyone who needs to work on the `version` has its reference count incremented by 1, and decremented by 1 when finishing using it. When a `version`'s reference count is 0, it should be removed. Either a `version` is up-to-date or someone is using it, its reference count is not 0, so it will be kept.

Sometimes a reader holds reference of a `version` directly, like the source `version` for a compaction. More often, a reader holds it indirectly through a data structure called `super version`, which holds reference counts for list of mem tables and a `version` – a whole view of the DB. A reader only needs to increase and decrease one reference count, while it is the super version that holds the reference count of `version`. It also enables further optimization to avoid locking for the reference counting in most of the time. See the [blog post](#).

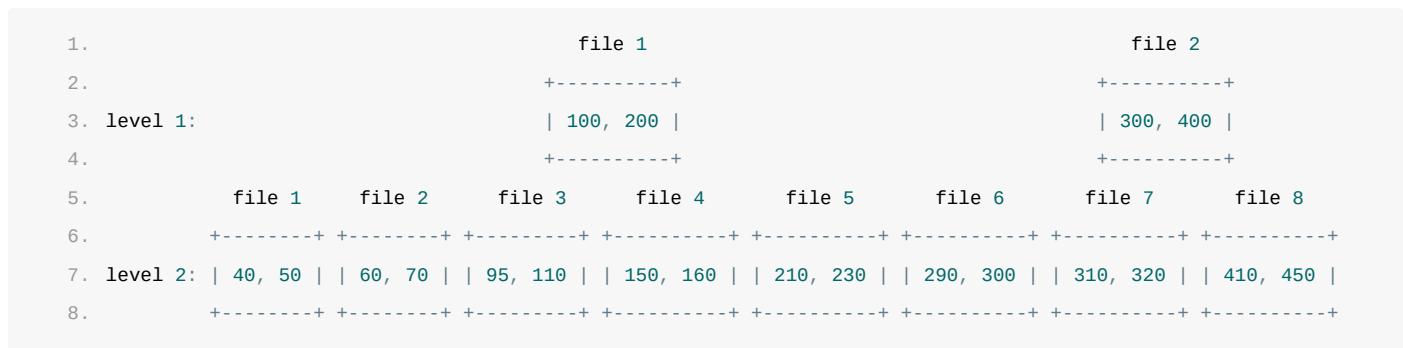
RocksDB maintains all `version`s in the data structure called `VersionSet`, which also remembers who is the “current” `version`. Since each `column family` is a separate LSM, it also has its own list of `version`s with one that is “current”. But there is only one `VersionSet` per DB that maintains `version`s for all column families.

For a Get() request, RocksDB goes through mutable memtable, list of immutable memtables, and SST files to look up the target key. SST files are organized in levels.

On level 0, files are sorted based on the time they are flushed. Their key range (as defined by `FileMetaData.smallest` and `FileMetaData.largest`) are mostly overlapped with each other. So it needs to look up every L0 file.

Compaction is scheduled periodically to pick up files from an upper level and merge them with files from lower level. As a result, key/values are moved from L0 down the LSM tree gradually. Compaction sorts key/values and split them into files. From level 1 and below, SST files are sorted based on key. Their key ranges are mutually exclusive. Instead of scanning through each SST file and checking if a key falls into its range, RocksDB performs a binary search based on `FileMetaData.largest` to locate a candidate file that can potentially contain the target key. This reduces complexity from  $O(N)$  to  $O(\log(N))$ . However,  $\log(N)$  can still be large for bottom levels. For a fan-out ratio of 10, level 3 can have 1000 files. That requires 10 comparisons to locate a candidate file. This is a significant cost for an in-memory database when you can do [several million gets per second](#).

One observation to this problem is that: after the LSM tree is built, an SST file's position in its level is fixed. Furthermore, its order relative to files from the next level is also fixed. Based on this idea, we can perform [fractional cascading](#) kind of optimization to narrow down the binary search range. Here is an example:



Level 1 has 2 files and level 2 has 8 files. Now, we want to look up key 80. A binary search based `FileMetaData.largest` tells you file 1 is the candidate. Then key 80 is compared with its `FileMetaData.smallest` and `FileMetaData.largest` to decide if it falls into the range. The comparison shows 80 is less than `FileMetaData.smallest` (100), so file 1 does not possibly contain key 80. We proceed to check level 2. Usually, we need to do binary search among all 8 files on level 2. But since we already know target key 80 is less than 100 and only file 1 to file 3 can contain key less than 100, we can safely exclude other files from the search. As a result, we cut down the search space from 8 files to 3 files.

Let's look at another example. We want to get key 230. A binary search on level 1 locates to file 2 (this also implies key 230 is larger than file 1's `FileMetaData.largest` 200). A comparison with file 2's range shows the target key is smaller than file 2's `FileMetaData.smallest` 300. Even though we couldn't find key on level 1, we have derived hints that target key is in range between 200 and 300. Any

files on level 2 that cannot overlap with [200, 300] can be safely excluded. As a result, we only need to look at file 5 and file 6 on level 2.

Inspired by this concept, we pre-build pointers at compaction time on level 1 files that point to a range of files on level 2. For example, file 1 on level 1 points to file 3 (on level 2) on the left and file 4 on the right. File 2 will point to level 2 files 6 and 7. At query time, these pointers are used to determine the actual binary search range based on comparison result.

Our benchmark shows that this optimization improves lookup QPS by ~5% for the setup mentioned [here](#).

This page describes the internal implementation details for the RocksDB Merge feature. It is aimed at expert RocksDB engineers and/or other Facebook engineers who are interested in understanding how Merge works.

If you are a user of RocksDB and you only want to know how to use Merge in production, go to the [Client Interface](#) page. Otherwise, it is assumed that you have already read that page.

## Context

Here is a high-level overview of the code-changes that we needed in order to implement Merge:

- We created an abstract base class called `MergeOperator` that the user needs to inherit from.
- We updated that `Get()`, `iteration`, and `Compaction()` call paths to call the `MergeOperator`'s `FullMerge()` and `PartialMerge()` functions when necessary.
- The major change needed was to implement “stacking” of merge operands, which we describe below.
- We introduced some other interface changes (i.e.: updated the `Options` class and `DB` class to support `MergeOperator`)
- We created a simpler `AssociativeMergeOperator` to make the user's lives easier under a very common use-case. Note this can be much more inefficient.

For the reader, if any of the above statements do not make sense at a high level, YOU PROBABLY SHOULD READ THE [Client Interface page first](#). Otherwise, we dive directly into the details below, and also talk about some design decisions and rationale for picking the implementations we did.

## The Interface

A quick reiteration of the interface (it is assumed that the reader is somewhat familiar with it already):

```

1. // The Merge Operator
2. //
3. // Essentially, a MergeOperator specifies the SEMANTICS of a merge, which only
4. // client knows. It could be numeric addition, list append, string
5. // concatenation, edit data structure, ... , anything.
6. // The library, on the other hand, is concerned with the exercise of this
7. // interface, at the right time (during get, iteration, compaction...)
8. class MergeOperator {
9. public:
10.    virtual ~MergeOperator() {}
11.
12.    // Gives the client a way to express the read -> modify -> write semantics
13.    // key:      (IN) The key that's associated with this merge operation.
14.    // existing: (IN) null indicates that the key does not exist before this op

```

```

15. // operand_list:(IN) the sequence of merge operations to apply, front() first.
16. // new_value: (OUT) Client is responsible for filling the merge result here
17. // logger: (IN) Client could use this to log errors during merge.
18. //
19. // Return true on success, false on failure/corruption/etc.
20. virtual bool FullMerge(const Slice& key,
21.                         const Slice* existing_value,
22.                         const std::deque<std::string>& operand_list,
23.                         std::string* new_value,
24.                         Logger* logger) const = 0;
25.
26. // This function performs merge(left_op, right_op)
27. // when both the operands are themselves merge operation types.
28. // Save the result in *new_value and return true. If it is impossible
29. // or infeasible to combine the two operations, return false instead.
30. virtual bool PartialMerge(const Slice& key,
31.                           const Slice& left_operand,
32.                           const Slice& right_operand,
33.                           std::string* new_value,
34.                           Logger* logger) const = 0;
35.
36. // The name of the MergeOperator. Used to check for MergeOperator
37. // mismatches (i.e., a DB created with one MergeOperator is
38. // accessed using a different MergeOperator)
39. virtual const char* Name() const = 0;
40. };

```

## RocksDB Data Model

Before going into the gory details of how merge works, let's try to understand the data model of RocksDB first.

In a nutshell, RocksDB is a versioned key-value store. Every change to the db is globally ordered and assigned a monotonically increasing sequence number. For each key, RocksDB keeps the history of operations. We denote each operation as OP<sub>i</sub>. A key (K) that experienced n changes, looks like this logically (physically, the changes could be in the active memtable, the immutable memtables, or the level files).

```
1. K:   OP1   OP2   OP3   ...   OPn
```

An operation has three properties: its type - either a Delete or a Put (now we have Merge too), its sequence number and its value (Delete can be treated as a degenerate case without a value). Sequence numbers will be increasing but not contiguous with regard to a single key, as they are globally shared by all keys.

When a client issues db->Put or db->Delete, the library literally appends the operation to the history. No checking of the existing value is done, probably for performance consideration (no wonder Delete remains silent if the key does not pre-exist...)

What about db->Get? It returns the state of a key with regard to a point in time, specified by a sequence number. The state of a key could be either non-existent or an opaque string value. It starts as non-existent. Each operation moves the key to a new state. In this sense, each key is a state machine with operations as transitions.

From the state machine point of view, Merge is really a generic transition that looks at the current state (existing value, or non-existence of that), combines it with the operand (the value associated with the Merge operation) and then produces a new value (state). Put, is a degenerate case of Merge, that pays no attention to the current state, and produces the new state solely based on its operand. Delete goes one step further - it doesn't even have an operand and always bring the key back to its original state - non-existent.

## Get

In principal, Get returns the state of a key at a specific time.

```

1. K:   OP1   OP2   OP3   ....   OPk   .... OPn
2.           ^
3.           |
4.           Get.seq

```

Suppose OPk is the most recent operation that's visible to Get:

```
k = max(i) {seq(OPi) <= Get.seq}
```

Then, if OPk is a Put or Delete, Get should simply return the value specified (if Put) or a NotFound status (if Delete). It can ignore previous values.

With the new Merge operation, we actually need to look backward. And how far do we go? Up to a Put or Delete (history beyond that is not essential anyways).

```

1. K:   OP1   OP2   OP3   ....   OPk   .... OPn
2.       Put   Merge  Merge  Merge
3.           ^
4.           |
5.           Get.seq
6. ----->

```

For the above example, Get should return something like:

```
Merge(...Merge(Merge(operand(OP2), operand(OP3)), operand(OP4)..., operand(OPk))))
```

Internally, RocksDB traverses the key history from new to old. The internal data structures for RocksDB support a nice “binary-search” style Seek function. So, provided a sequence number, it can actually return:  $k = \max(i) \{seq(OPi) \leq Get.seq\}$  relatively efficiently. Then, beginning with OPk, it will iterate through history from new to old as mentioned until a Put/Delete is found.

In order to actually enact the merging, rocksdb makes use of the two specified Merge-Operator methods: `FullMerge()` and `PartialMerge()`. The Client Interface page gives a good overview on what these functions mean at a high-level. But, for the sake of completeness, it should be known that `PartialMerge()` is an optional function, used to combine two merge operations (operands) into a single operand. For example, combining  $OP(k-1)$  with  $OP_k$  to produce some  $OP'$ , which is also a merge-operation type. Whenever `PartialMerge()` is unable to combine two operands, it returns false, signaling to rocksdb to handle the operands itself. How is this done? Well, internally, rocksdb provides an in-memory stack-like data structure (we actually use an STL Deque) to stack the operands, maintaining their relative order, until a Put/Delete is found, in which case `FullMerge()` is used to apply the list of operands onto the base-value.

The algorithm for `Get()` is as follows:

```

1. Get(key):
   Let stack = [ ];           // in reality, this should be a "deque", but stack is simpler to conceptualize for
2. this pseudocode
3. for each entry OPi from newest to oldest:
4.   if OPi.type is "merge_operand":
5.     push OPi to stack
       while (stack has at least 2 elements and (stack.top() and stack.second_from_top() can be partial-
6. merged)
7.       OP_left = stack.pop()
8.       OP_right = stack.pop()
9.       result_OP = client_merge_operator.PartialMerge(OP_left, OP_right)
10.      push result_OP to stack
11.    else if OPi.type is "put":
12.      return client_merge_operator.FullMerge(v, stack);
13.    else if v.type is "delete":
14.      return client_merge_operator.FullMerge(nullptr, stack);
15.
16. // We've reached the end (OP0) and we have no Put/Delete, just interpret it as empty (like Delete would)
17. return client_merge_operator.FullMerge(nullptr, stack);

```

Thus, RocksDB will “stack up” the operations until it reaches a Put or a Delete (or the beginning of the key history), and will then call the user-defined `FullMerge()` operation with the sequence/stack of operations passed in as a parameter. So, with the above example, it will start at  $OP_k$ , then go to  $OP_{k-1}$ , ..., etc. When RocksDB encounters  $OP_2$ , it will have a stack looking like  $[OP_3, OP_4, \dots, OP_k]$  of Merge operands (with  $OP_3$  being the front/top of stack). It will then call the user-defined `MergeOperator::FullMerge(key, existing_value = OP_2, operands = [OP_3, OP_4, \dots, OP_k])`. This should return the result to the user.

## Compaction

Here comes the fun part, the most crucial background process of rocksdb. Compaction is a process of reducing the history of a key, without affecting any externally observable state. What's an externally observable state? A snapshot basically,

represented by a sequence number. Let's look at an example:

|    |    |     |           |     |           |     |           |     |
|----|----|-----|-----------|-----|-----------|-----|-----------|-----|
| 1. | K: | OP1 | OP2       | OP3 | OP4       | OP5 | ...       | OPn |
| 2. |    |     | ^         |     | ^         |     | ^         |     |
| 3. |    |     |           |     |           |     |           |     |
| 4. |    |     | snapshot1 |     | snapshot2 |     | snapshot3 |     |

For each snapshot, we could define the Supporting operation as the most recent operation that's visible to the snapshot (OP2 is the Supporting operation of snapshot1, OP4 is the Supporting operation of snapshot2...).

Obviously, we could not drop any Supporting operations, without affecting externally observable states. What about other operations? Before the introduction of Merge operation, we could say bye to ALL non-supporting operations. In the above example, a full Compaction would reduce the history of K to OP2 OP4 and OPn. The reason is simple: Put's and Delete's are shortcuts, they hide previous operations.

With merge, the procedure is a bit different. Even if some merge operand is not a Supporting operation for any snapshot, we cannot simply drop it, because later merge operations may rely on it for correctness. Also, in fact, this means that we cannot even drop past Put or Delete operations, because there may be later merge operands that rely on them as well.

So what do we do? We proceed from newest to oldest, "stacking" (and/or PartialMerging) the merge operands. We stop the stacking and process the stack in any one of the following cases (whichever occurs first):

1. a Put/Delete is encountered - we call `FullMerge(value or nullptr, stack)`
2. End-of-key-history is encountered - we call `FullMerge(nullptr, stack)`
3. a Supporting operation (snapshot) is encountered - see below
4. End-of-file is encountered - see below

The first two cases are more-or-less similar to `Get()`. If you see a Put, call `FullMerge(value of put, stack)`. If you see a delete, likewise.

Compaction introduces two new cases, however. First, if a snapshot is encountered, we must stop the merging process. When this happens, we simply write out the un-merged operands, clear the stack, and continue compacting (starting with the Supporting operation). Similarly, if we have completed compaction ("end-of-file"), we can't simply apply `FullMerge(nullptr, stack)` because we may not have seen the beginning of the key's history; there may be some entries in some files that happen to not be included in compaction at the time. Hence, in this case, we also have to simply write out the un-merged operands, and clear the stack. In both of these cases, all merge operands become like "Supporting operations", and cannot be dropped.

The role of Partial Merge here is to facilitate compaction. Since it can be very likely that a supporting operation or end-of-file is reached, it can be likely that most merge operands will not be compacted away for a long time. Hence, Merge Operators

that support partial merge make it easier for compaction, because the left-over operands will not be stacked, but will be combined into single merge operands before being written out to the new file.

## Example

Let's walk through a concrete example as we come up with rules. Say a counter K starts out as 0, goes through a bunch of Add's, gets reset to 2, and then goes through some more Add's. Now a full Compaction is due (with some externally observable snapshots in place) - what happens?

(Note: In this example we assume associativity, but the idea is the same without PartialMerge as well)

```

1. K: 0 +1 +2 +3 +4 +5 2 +1 +2
2.           ^ ^
3.           | |
4.           snapshot1 snapshot2           snapshot3
5.
6. We show it step by step, as we scan from the newest operation to the oldest operation
7.
8. K: 0 +1 +2 +3 +4 +5 2 (+1 +2)
9.           ^ ^
10.          | |
11.         snapshot1 snapshot2           snapshot3
12.
13. A Merge operation consumes a previous Merge Operation and produces a new Merge operation (or a stack)
14.      (+1 +2) => PartialMerge(1,2) => +3
15.
16. K: 0 +1 +2 +3 +4 +5 2 +3
17.           ^ ^
18.           | |
19.         snapshot1 snapshot2           snapshot3
20.
21. K: 0 +1 +2 +3 +4 +5 (2 +3)
22.           ^ ^
23.           | |
24.         snapshot1 snapshot2           snapshot3
25.
26. A Merge operation consumes a previous Put operation and produces a new Put operation
27.      (2 +3) => FullMerge(2, 3) => 5
28.
29. K: 0 +1 +2 +3 +4 +5 5
30.           ^ ^
31.           | |
32.         snapshot1 snapshot2           snapshot3
33.
34. A newly produced Put operation is still a Put, thus hides any non-Supporting operations
35.      (+5 5) => 5
36.
37. K: 0 +1 +2 (+3 +4) 5

```

```

38.          ^
39.          |
40.          snapshot1  snapshot2          ^
41.          |
42. (+3  +4) => PartialMerge(3,4) => +7
43.
44. K:   0    +1    +2          +7          5
45.          ^    ^
46.          |    |
47.          snapshot1  snapshot2          snapshot3
48.
49. A Merge operation cannot consume a previous Supporting operation.
50. (+2    +7) can not be combined
51.
52. K:   0    (+1    +2)          +7          5
53.          ^    ^
54.          |    |
55.          snapshot1  snapshot2          snapshot3
56.
57. (+1  +2) => PartialMerge(1,2) => +3
58.
59. K:   0    +3    +7          5
60.          ^    ^
61.          |    |
62.          snapshot1  snapshot2          snapshot3
63.
64. K:   (0    +3)          +7          5
65.          ^    ^
66.          |    |
67.          snapshot1  snapshot2          snapshot3
68.
69. (0    +3) => FullMerge(0,3) => 3
70.
71. K:   3    +7          5
72.          ^    ^
73.          |    |
74.          snapshot1  snapshot2          snapshot3

```

To sum it up: During Compaction, if a Supporting operation is Merge, it will combine previous operations (via PartialMerge or stacking) until

- another Supporting operation is reached (in others words, we crossed snapshot boundary)
- a Put or a Delete operation is reached, where we convert the Merge operation to a Put.
- end-of-key-history is reached, where we convert Merge operation to a Put
- end-of-Compaction-Files is reached, where we treat it as crossing snapshot boundary

Note that we assumed that the Merge operation defined `PartialMerge()` in the example above. For operations without `PartialMerge()`, the operands will instead be combined on

a stack until one of the cases are encountered.

## Issues with this compaction model

In the event that a Put/Delete is not found, for example, if the Put/Delete happens to be in a different file that is not undergoing compaction, then the compaction will simply write out the keys one-by-one as if they were not compacted. The major issue with this is that we did unnecessary work in pushing them to the deque.

Similarly, if there is a single key with MANY merge operations applied to it, then all of these operations must be stored in memory if there is no PartialMerge. In the end, this could lead to a memory overflow or something similar.

**Possible future solution:** To avoid the memory overhead of maintaining a stack/deque, it might be more beneficial to traverse the list twice, once forward to find a Put/Delete, and then once in reverse. This would likely require a lot of disk IO, but it is just a suggestion. In the end we decided not to do this, because for most (if not all) workloads, the in-memory handling should be enough for individual keys. There is room for debate and benchmarking around this in the future.

## Compaction Algorithm

Algorithmically, compaction now works as follows:

```

1. Compaction(snaps, files):
2.   // <snaps> is the set of snapshots (i.e.: a list of sequence numbers)
3.   // <files> is the set of files undergoing compaction
4.   Let input = a file composed of the union of all files
5.   Let output = a file to store the resulting entries
6.
7.   Let stack = [];           // in reality, this should be a "deque", but stack is simpler to conceptualize in this
    pseudo-code
8.   for each v from newest to oldest in input:
9.     clear_stack = false
10.    if v.sequence_number is in snaps:
11.      clear_stack = true
12.    else if stack not empty && v.key != stack.top.key:
13.      clear_stack = true
14.
15.    if clear_stack:
16.      write out all operands on stack to output (in the same order as encountered)
17.      clear(stack)
18.
19.    if v.type is "merge_operand":
20.      push v to stack
21.      while (stack has at least 2 elements and (stack.top and stack.second_from_top can be partial-merged)):
22.        v1 = stack.pop();
23.        v2 = stack.pop();
24.        result_v = client_merge_operator.PartialMerge(v1, v2)
25.        push result_v to stack

```

```

26.     if v.type is "put":
27.         write client_merge_operator.FullMerge(v, stack) to output
28.         clear stack
29.     if v.type is "delete":
30.         write client_merge_operator.FullMerge(nullptr, stack) to output
31.         clear stack
32.
33.     If stack not empty:
34.         if end-of-key-history for key on stack:
35.             write client_merge_operator.FullMerge(nullptr, stack) to output
36.             clear(stack)
37.         else
38.             write out all operands on stack to output
39.             clear(stack)
40.
41.     return output

```

## Picking upper-level files in Compaction

Notice that the relative order of all merge operands should always stay fixed. Since all iterators search the database “level-by-level”, we never want to have older merge-operands in an earlier level than newer merge-operands. So, we also had to update compaction so that when it selects its file for compaction, it expands its upper-level files to also include all “earlier” merge-operands. Why does this change things? Because, when any entry is compacted, it will always move to a lower level. So if the merge-operands for a given key are spread over multiple files in the same level, but only some of those files undergo compaction, then that circumstance can happen where the newer merge-operands get pushed down to a later level.

**This was technically a bug in classic rocksdb!** Specifically, this issue was always there, but for most (if not all) applications without merge-operator, it can be assumed that there will be one version of each key per level (except for level-0), since compaction would always compress duplicate Puts to simply contain the latest put value. So this concept of swapping orders was irrelevant, except in level-0 (where compaction always includes all overlapping files). So this has been fixed.

**Some issues:** On malicious inputs, this could lead to always having to include many files during compaction whenever the system really only wanted to pick one file. This could slow things down, but benchmarking suggested that this wasn’t really an issue.

## Notes on Efficiency

A quick discussion about efficiency with merge and partial-merge.

**Having a stack of operands can be more efficient.** For instance, in a string-append example (assuming NO partial-merge), providing the user with a stacked set of string operands to append allows the user to amortize the cost of constructing the final string. For example, if I am given a list of 1000 small strings that I need to append,

I can use these strings to compute the final size of the string, reserve/allocate space for this result, and then proceed and copy all of the data into the newly allocated memory array. Instead, if I were forced to always partial merge, the system would have to perform the order of 1000 reallocations, once per string operand, each a relatively large size, and the number of bytes having to be allocated in total might be extremely large. In most use-cases this is probably not an issue; and in our benchmarking we found that it really only mattered on a largely in-memory database with hot-keys, but this is something to consider for “growing data”. In any case, we provide the user with a choice nonetheless.

The key point to take away from this is that there are cases where having a stack of operands (rather than a single operand) provides an asymptotic increase in the overall efficiency of the operations. For example, in the string-append operator above, the merge-operator can theoretically make the string append operation an amortized  $O(N)$ -time operation (where  $N$  is the size of the final string after all operations), whereas without a stack, we can have an  $O(N^2)$  time operation.

For more information, contact the rocksdb team. Or see the RocksDB wiki page.

# Overview

Repairer does best effort recovery to recover as much data as possible after a disaster without compromising consistency. It does not guarantee bringing the database to a time consistent state. Note: Currently there is a limitation that un-flushed column families will be lost after repair. This would happen even if the DB is in healthy state.

## Usage

Note the CLI command uses default options for repairing your DB and only adds the column families found in the SST files. If you need to specify any options, e.g., custom comparator, have column family-specific options, or want to specify the exact set of column families, you should choose the programmatic way.

## Programmatic

For programmatic usage, call one of the `RepairDB` functions declared in `include/rocksdb/db.h`.

## CLI

For CLI usage, first build `ldb`, our admin CLI tool:

```
1. $ make clean && make ldb
```

Now use the `ldb`'s `repair` subcommand, specifying your DB. Note it prints info logs to stderr so you may wish to redirect. Here I run it on a DB in `./tmp` where I've deleted the MANIFEST file:

```
1. $ ./ldb repair --db=./tmp 2>./repair-log.txt
2. $ tail -2 ./repair-log.txt
[WARN] [db/repair.cc:208] **** Repaired rocksdb ./tmp; recovered 1 files; 926bytes. Some data may have been
3. lost. ****
```

Looks successful. MANIFEST file is back and DB is readable:

```
1. $ ls tmp/
000006.sst CURRENT IDENTITY LOCK LOG LOG.old.1504116879407136 lost MANIFEST-000001 MANIFEST-000003
2. OPTIONS-000005
3. $ ldb get a --db=./tmp
4. b
```

Notice the `lost/` directory. It holds files containing data that was potentially lost during recovery.

# Repair Process

---

Repair process is broken into 4 phase:

- Find files
- Convert logs to tables
- Extract metadata
- Write Descriptor

## Find files

The repairer goes through all the files in the directory, and classifies them based on their file name. Any file that cannot be identified by name will be ignored.

## Convert logs to table

Every log file that is active is replayed. All sections of the file where the checksum does not match is skipped over. We intentionally give preference to data consistency.

## Extract metadata

We scan every table to compute

- smallest/largest for the table
- largest sequence number in the table

If we are unable to scan the file, then we ignore the table.

## Write Descriptor

We generate descriptor contents:

- log number is set to zero
- next-file-number is set to 1 + largest file number we found
- last-sequence-number is set to largest sequence# found across all tables
- compaction pointers are cleared
- every table file is added at level 0

## Possible optimizations

1. Compute total size and use to pick appropriate max-level M
2. Sort tables by largest sequence# in the table
3. For each table: if it overlaps earlier table, place in level-0, else place in level-M.
4. We can provide options for time consistent recovery and unsafe recovery (ignore checksum failure when applicable)
5. Store per-table metadata (smallest, largest, largest-seq#, ...) in the table's meta section to speed up ScanTable.

## Limitations

---

If the column family is created recently and not persisted in sst files by a flush, then it will be dropped during the repair process. With this limitation repair would might even damage a healthy db if its column families are not flushed yet.

This document outlines the implementation of Two Phase Commit in rocksdb.

This project can be decomposed into five areas of focus:

1. Modification of the WAL format
2. Extension of the existing transaction API
3. Modification of the write path
4. Modification of the recovery path
5. Integration with MyRocks

## Modification of WAL Format

The WAL consists of one or more logs. Each log is one or more serialized WriteBatches. During recovery WriteBatches are reconstructed from the logs. To modify the WAL format or extend its functionality we must only concern ourselves with the WriteBatch.

A WriteBatch is an ordered set of records (`Put(k,v)`, `Merge(k,v)`, `Delete(k)`, `SingleDelete(k)`) that represent RocksDB write operations. Each of these records has a binary string representation. As records are added to a WriteBatch their binary representation is appended to the WriteBatch's binary string representation. This binary string is prefixed with the starting sequence number of the batch followed by the number of records contained in the batch. Each record may be prefixed with a column family modifier record if the operation does not apply to the default column family.

A WriteBatch can be iterated over by extending the `WriteBatch::Handler`.

`MemTableInserter` is an extension of `WriteBatch::Handler` which inserts the operations contained in a WriteBatch into the appropriate column family `MemTable`.

An existing WriteBatch may have the logical representation:

```
Sequence(0);NumRecords(3);Put(a,1);Merge(a,1);Delete(a);
```

Modification of the WriteBatch format for 2PC includes the addition of four new records.

- `Prepare(xid)`
- `EndPrepare()`
- `Commit(xid)`
- `Rollback(xid)`

A 2PC capable WriteBatch may have the logical representation:

```
Sequence(0);NumRecords(6);Prepare(foo);Put(a,b);Put(x,y);EndPrepare();Put(j,k);Commit(foo);
```

It can be seen that `Prepare(xid)` and `EndPrepare()` are analogous to mating brackets which contain the operations belonging to transaction with ID 'foo'. `Commit(xid)` and `Rollback(xid)` mark that operations belonging to transaction with ID `xid` should be committed or rolled-back.

### Sequence ID Distribution

When a WriteBatch is inserted into a memtable (via MemTableInserter) the sequence ID of each operation is equal to the sequence ID of the WriteBatch plus the number of sequence ID consuming records previous to this operation in the WriteBatch. This implicit mapping of sequence ids within a WriteBatch will no longer hold with the addition of 2PC. Operations contained within a Prepare() enclosure will consume sequence IDs as if they were inserted starting at the location of their relative Commit() marker. This Commit() marker may be in a different WriteBatch or log from the prepared operations to which it applies.

### Backwards Compatibility

WAL formats are not versioned so we need to take note of backwards compatibility. A current version of RocksDB would not be able to recover itself from a WAL file containing 2PC markers. In fact it would fatal on the unrecognized record ids. It would be trivial, however, to patch a current version of RocksDB to be able to recover itself from this new WAL format just skipping over the prepared sections and unknown markers.

### Existing Progress

Progress had been made on this front and relevant discussion can be found at <https://reviews.facebook.net/D54093>

## Extension of Transaction API

For the time being we will only focus on 2PC for pessimistic transactions. The client must specify ahead of time if they intend to employ two phase commit semantics. For example, the client code could be imagined as:

```

1. TransactionDB* db;
2. TransactionDB::Open(Options(), TransactionDBOptions(), "foodb", &db);
3.
4. TransactionOptions txn_options;
5. txn_options.two_phase_commit = true
6. txn_options.xid = "12345";
7. Transaction* txn = db->BeginTransaction(write_options, txn_options);
8.
9. txn->Put(...);
10. txn->Prepare();
11. txn->Commit();
```

A transaction object now has more states that it can occupy so our enum of states now becomes:

```

1. enum ExecutionStatus {
2.     STARTED = 0,
3.     AWAITING_PREPARE = 1,
4.     PREPARED = 2,
5.     AWAITING_COMMIT = 3,
```

```

6.    COMMITTED = 4,
7.    AWAITING_ROLLBACK = 5,
8.    ROLLEDBACK = 6,
9.    LOCKS_STOLEN = 7,
10.   };

```

The transaction API will gain a new member function, `Prepare()`. `Prepare()` will call into `WriteImpl` with a context of its self giving `WriteImpl` and the `WriteThread` access to the `ExecutionStatus`, `XID`, and `WriteBatch`. `WriteImpl` will insert the `Prepare(xid)` marker followed by the contents of the `WriteBatch` followed by `EndPrepare()` marker. No memtable insertion will be issued. When the same transaction instance issued its commit, again, it calls into `WriteImpl()`. This time only a `Commit()` marker is inserted into the WAL on its behalf and the contents of the `WriteBatch` are inserted into the appropriate memtables. When `Rollback()` on the transactions is called the contents of the transactions are cleared and a call into `WriteImpl` to insert a `Rollback(xid)` marker is made if the transaction is in a prepared state.

These so-called ‘meta markers’ (`Prepare(xid)`, `EndPrepare()`, `Commit(xid)`, `Rollback(xid)`) will never be inserted directly into a write batch. The write path (`WriteImpl()`) will have the context of the transactions it is writing. It uses this context to insert the relevant markers directly into the WAL (So they are inserted into the aggregate `WriteBatch` right before being inserted into the WAL, but no other `WriteBatch`). During recovery these markers will be encountered by the `MemTableInserter` which he will use to reconstruct previously prepared transactions.

### Transaction Wallclock Expiration

Currently at the time of a transaction commit there is a callback which will fail the write if the transaction has expired. Similarly, if a transaction has expired then it is now eligible to have its locks stolen by other transactions. These mechanisms should still be in place for 2PC - the difference being that the expiration callback will be called at the time of preparation. If the transaction did not expire at the time of preparation then it cannot expire at the time of commit.

### TransactionDB Modification

To use transactions the client must open a `TransactionDB`. This `TransactionDB` instance is then used to create `Transactions`. This `TransactionDB` now keeps track of a mapping from `XID` to all two phase transactions which have been created. When a transaction is Deleted or Rolled-back it is removed from this mapping. There is also an API to query all outstanding prepared transactions. This is used during MyRocks recovery.

The `TransactionDB` also keeps track of a min heap of all log numbers containing a prepared section. When a transaction is ‘prepared’ its `WriteBatch` is written to a log, this log number is then stored in the transaction object and subsequently the min heap. When a transaction is committed its log number is deleted from the min heap, but it is not forgotten! It is now the duty of each memtable to keep track of the oldest log it needs to keep around until this is successfully flushed to L0.

## Modification of the Write Path

The write path can be decomposed into two main areas of focus. DBImpl::WriteImpl(...) and the MemTableInserter. Multiple client threads will call into WriteImpl. The first thread will be designated as the ‘leader’ while a number of following threads will be designated as ‘followers’. Both the leader and set of followers will be batched together into a logical group referred to as a ‘write group’. The leader will take all WriteBatches of the write group, concatenate them together and write this blob out to the WAL. Depending on the size of the write group and the current memtables’s willingness to support parallel writes the leader may insert all WriteBatches into the memtable or each thread may be left to insert his own WriteBatch into the memtable.

All memtable inserts are handled by MemTableInserter. This is an implementation of WriteBatch::Handler - a WriteBatch iterator handler. This handler iterates over all elements in a WriteBatch (Put, Delete, Merge, etc) and makes the appropriate call into the current MemTable. MemTableInserter will also handle in-place merges, deletes and updates.

Modification of the write path will include adding an optional parameter to DBImpl::WriteImpl. This optional parameter will be a pointer to the two phase transaction instance that is having his data written. This object will give the write path insight into the current state of two phase transaction. A 2PC transaction will call into WriteImpl once for preparation, once for commit, and once for roll-back - though commit and rollback are obviously exclusive operations.

```

1. Status DBImpl::WriteImpl(
2.   const WriteOptions& write_options,
3.   WriteBatch* my_batch,
4.   WriteCallback* callback,
5.   Transaction* txn
6. ) {
7.   WriteThread::Writer w;
8.   //...
9.   w.txn = txn; // writethreads also have txn context for memtable insert
10.
11. // we are now the group leader
12. int total_count = 0;
13. uint64_t total_byte_size = 0;
14. for (auto writer : write_group) {
15.   if (writer->CheckCallback(this)) {
16.     if (writer->ShouldWriteToMem())
17.       total_count += WriteBatchInternal::Count(writer->batch)
18.   }
19. }
20. const SequenceNumber current_sequence = last_sequence + 1;
21. last_sequence += total_count;
22.
23. // now we produce the WAL entry from our write group
24. for (auto writer : write_group) {
25.   // currently only optimistic transactions use callbacks

```

```

26.    // and optimistic transaction do not support 2pc
27.    if (writer->CallbackFailed()) {
28.        continue;
29.    } else if (writer->IsCommitPhase()) {
30.        WriteBatchInternal::MarkCommit(merged_batch, writer->txn->XID_);
31.    } else if (writer->IsRollbackPhase()) {
32.        WriteBatchInternal::MarkRollback(merged_batch, writer->txn->XID_);
33.    } else if (writer->IsPreparePhase()) {
34.        WriteBatchInternal::MarkBeginPrepare(merged_batch, writer->txn->XID_);
35.        WriteBatchInternal::Append(merged_batch, writer->batch);
36.        WriteBatchInternal::MarkEndPrepare(merged_batch);
37.        writer->txn->log_number_ = logfile_number_;
38.    } else {
39.        assert(writer->ShouldWriteToMem());
40.        WriteBatchInternal::Append(merged_batch, writer->batch);
41.    }
42. }
43. //now do MemTable Inserts for WriteGroup
44. }
```

`WriteBatchInternal::InsertInto` could then be modified to only iterate over writers having no Transaction associated or Transactions in the COMMIT state.

#### Modification of MemTableInserter for WritePath

As you can see above when a transactions is prepared the transaction takes note of what log number its prepared section resided in. At the time of insertion each MemTable must keep track of the minimum log number containing prepared data which has been inserted into him. This modification will take place in the MemTableInserter. We will discuss how this value is used in the log lifespan section.

## Modification of Recovery Path

The current recovery path is already pretty well suited for two phase commit. It iterates over all batches in all the logs in chronological order and feeds them, along the the log number, into the MemTableInserter. The MemTableInserter then iterates over each of these batches and inserts the values into the correct MemTable. Each MemTable knows what values it can ignore for insertion based on the current log number being recovered from.

To make recovery work for 2PC we must only modify the MemTableInserter to be aware of our four new 'meta markers'.

Keep this in mind: When a two phase transaction is committed it contains insertions that will act on multiple CFs (multiple memtables). These memtables will flush at different times. We still make use of the CF log number to avoid duplicate inserts for recovered, two phase, committed transaction.

Consider the following scenario:

1. Two Phase Transactions TXN inserts into CFA and CFB
2. TXN prepared to LOG 1
3. TXN marked as COMMITTED in LOG 2
4. TXN is inserted into MemTables
5. CFA is flushed to L0
6. CFA log\_number is now LOG 3
7. CFB has not been flushed and it still referencing LOG 1 prep section
8. CRASH RECOVERY
9. LOG 1 is still around because CFB was referencing LOG 1 prep section
10. Iterate over logs starting at LOG 1
11. CFB has prepared values reinserted into mem, again referencing LOG 1 prep section
12. CFA skips insertion from commit marker in LOG 2 because it is consistent to LOG 3
13. CFB is flushed to L0 and is now consistent to LOG 3
14. LOG 1, LOG 2 can now be released

### Rebuilding Transactions

As mentioned before, modification of the recovery path only required modification of MemTableInserter to handle the new meta-markers. Because at the time of recovery we can't have access to a full instance of a TransactionDB we must recreate hollow 'shill' transactions. This is essentially mapping of XID → (WriteBatch, log\_number) for all recovered prepared transactions. When we hit a Commit(xid) marker we attempt to look up the shill transaction for this xid and re-insert into Mem. If we hit a rollback(xid) marker we delete the shill transaction. At the end of recovery we are left with a set of all prepared transactions in shill form. We then recreate full transactions from these objects, acquiring the required locks. Rocks DB is now the same state it was before crash/shutdown.

### Log Lifespan

To find the minimum log that must be kept around we first find the minimum log\_number\_ of each column family.

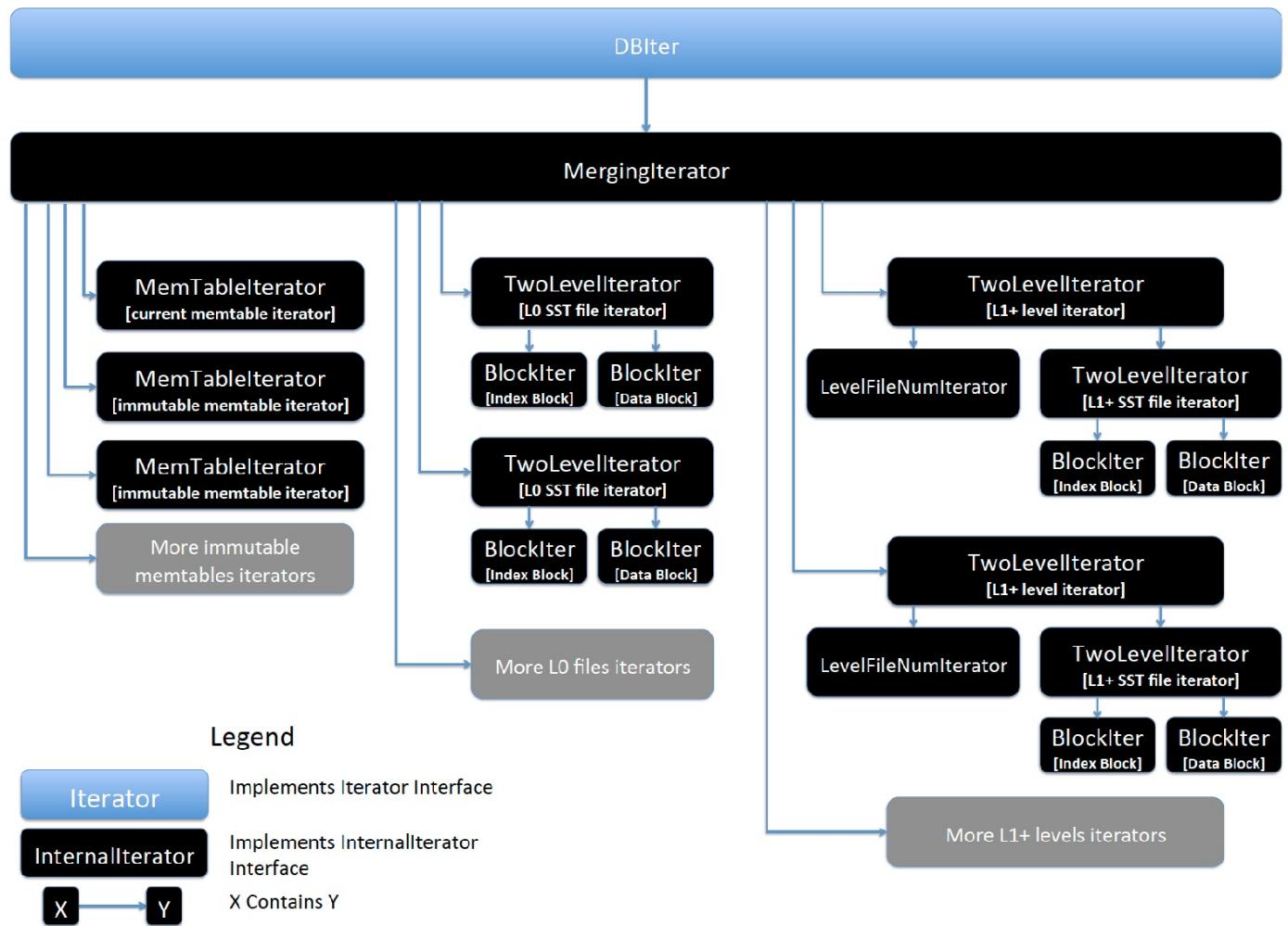
We must also consider the minimum value in the prepared sections heap in TransactionDB. This represents the earliest log containing a prep section that has not been committed.

We must also consider the minimum prep section log referenced by all MemTables and ImmutableMemTables that have not been flushed.

The minimum of these three values is the earliest log that still contains data not yet flushed to L0.

# RocksDB Iterator

RocksDB Iterator allows users to iterate over the DB forward and backward in a sorted manner. It also has the ability to seek to a specific key inside the DB, to achieve that .. the Iterator need to access the DB as a sorted stream. RocksDB Iterator implementation class is named `DBIter` , In this wiki page we will discuss how `DBIter` works and what it is composed of. In the following figure, you can see the design of `DBIter` and what it's composed of.



[Higher quality diagram](#)

## DBIter

Implementation: `db/db_iter.cc`

Interface: `Iterator`

`DBIter` is a wrapper around an `InternalIterator` (In this case a `MergingIterator` ). `DBIter` 's job is to parse InternalKeys exposed by the underlying `InternalIterator` and expose them as user keys.

Example:

The underlying `InternalIterator` exposed

```

1. InternalKey(user_key="Key1", seqno=10, Type=Put) | Value = "KEY1_VAL2"
2. InternalKey(user_key="Key1", seqno=9, Type=Put) | Value = "KEY1_VAL1"
3. InternalKey(user_key="Key2", seqno=16, Type=Put) | Value = "KEY2_VAL2"
4. InternalKey(user_key="Key2", seqno=15, Type=Delete) | Value = "KEY2_VAL1"
5. InternalKey(user_key="Key3", seqno=7, Type=Delete) | Value = "KEY3_VAL1"
6. InternalKey(user_key="Key4", seqno=5, Type=Put) | Value = "KEY4_VAL1"

```

But what `DBIter` will expose to the user is

```

1. Key="Key1" | Value = "KEY1_VAL2"
2. Key="Key2" | Value = "KEY2_VAL2"
3. Key="Key4" | Value = "KEY4_VAL1"

```

## MergingIterator

Implementation: [table/merging\\_iterator.cc](#)

Interface: `InternalIterator`

The `MergingIterator` is composed of many child iterators, `MergingIterator` is basically a heap for Iterators. In `MergingIterator` we put all child Iterators in a heap and expose them as one sorted stream.

Example:

The underlying child Iterators exposed

```

1. = Child Iterator 1 =
2. InternalKey(user_key="Key1", seqno=10, Type=Put) | Value = "KEY1_VAL2"
3.
4. = Child Iterator 2 =
5. InternalKey(user_key="Key1", seqno=9, Type=Put) | Value = "KEY1_VAL1"
6. InternalKey(user_key="Key2", seqno=15, Type=Delete) | Value = "KEY2_VAL1"
7. InternalKey(user_key="Key4", seqno=5, Type=Put) | Value = "KEY4_VAL1"
8.
9. = Child Iterator 3 =
10. InternalKey(user_key="Key2", seqno=16, Type=Put) | Value = "KEY2_VAL2"
11. InternalKey(user_key="Key3", seqno=7, Type=Delete) | Value = "KEY3_VAL1"

```

The `MergingIterator` will keep all child Iterators in a heap and expose them as one sorted stream

```

1. InternalKey(user_key="Key1", seqno=10, Type=Put) | Value = "KEY1_VAL2"
2. InternalKey(user_key="Key1", seqno=9, Type=Put) | Value = "KEY1_VAL1"
3. InternalKey(user_key="Key2", seqno=16, Type=Put) | Value = "KEY2_VAL2"

```

```

4. InternalKey(user_key="Key2", seqno=15, Type=Delete) | Value = "KEY2_VAL1"
5. InternalKey(user_key="Key3", seqno=7, Type=Delete) | Value = "KEY3_VAL1"
6. InternalKey(user_key="Key4", seqno=5, Type=Put) | Value = "KEY4_VAL1"

```

## MemtableIterator

Implementation: [db/memtable.cc](#)

Interface: [InternalIterator](#)

This is a wrapper around [MemtableRep::Iterator](#), Every memtable representation implements its own Iterator to expose the keys/values in the memtable as a sorted stream.

## BlockIter

Implementation: [table/block.h](#)

Interface: [InternalIterator](#)

This Iterator is used to read blocks from SST file, whether these blocks are index blocks or data blocks. Since SST file blocks are sorted and immutable, we load the block in memory and create a [BlockIter](#) for this sorted data.

## TwoLevelIterator

Implementation: [table/two\\_level\\_iterator.cc](#)

Interface: [InternalIterator](#)

A [TwoLevelIterator](#) is composed of 2 Iterators

- First level Iterator ([first\\_level\\_iter\\_](#))
- Second level Iterator ([second\\_level\\_iter\\_](#))

[first\\_level\\_iter\\_](#) is used to figure out the [second\\_level\\_iter\\_](#) to use, and [second\\_level\\_iter\\_](#) points to the actual data that we are reading.

Example:

RocksDB uses [TwoLevelIterator](#) to read SST files, [first\\_level\\_iter\\_](#) is a [BlockIter](#) on the SST file Index block and [second\\_level\\_iter\\_](#) is a [BlockIter](#) on a Data block.

Let's look at this simplified representation of an SST file, we have 4 Data blocks and 1 Index Block

```

1. [Data block, offset: 0x0000]
2. KEY1 | VALUE1
3. KEY2 | VALUE2
4. KEY3 | VALUE3

```

```

5.
6. [Data Block, offset: 0x0100]
7. KEY4 | VALUE4
8. KEY7 | VALUE7
9.

10. [Data Block, offset: 0x0250]
11. KEY8 | VALUE8
12. KEY9 | VALUE9
13.

14. [Data Block, offset: 0x0350]
15. KEY11 | VALUE11
16. KEY15 | VALUE15
17.

18. [Index Block, offset: 0x0500]
19. KEY3 | 0x0000
20. KEY7 | 0x0100
21. KEY9 | 0x0250
22. KEY15 | 0x0500

```

To read this file we will create a `TwoLevelIterator` with

- `first_level_iter_` => `BlockIter` over Index block
- `second_level_iter_` => `BlockIter` over Data block that will be determined by  
`first_level_iter_`

When we ask our `TwoLevelIterator` to Seek to `KEY8` for example, it will first use `first_level_iter_` (`BlockIter` over Index block) to figure out which block may contain this key. this will lead us to set the `second_level_iter_` to be (`BlockIter` over data block with offset `0x0250`). We will then use the `second_level_iter_` to find our key & value in the data block.

*Simulation Cache(SimCache)* can help users to predict the block cache performance metrics, e.t. hit, miss, of a specific simulation capacity (memory) under current work load, without actually using that much of memory.

## Motivation

It is able to help users tune their current block cache size, and determine how efficient they are using the memory. Also, it helps understand the cache performance with fast storage.

## Introduction

The basic idea of SimCache is to wrap the normal block cache with a key-only block cache configured with targeted simulation capacity. When insertion happens, we insert the key to both caches, but value is only inserted into normal cache. The size of the value is contributed to the capacities of both caches so that we simulate the behavior of a block cache with simulation capacity without using that much memory because in fact, the real memory usage only includes the total size of keys.

## How to use SimCache

Since SimCache is a wrapper on top of normal block cache. User has to create a block cache first with [NewLRUCache](#):

```
1. std::shared_ptr<rocksdb::Cache> normal_block_cache =
2.     NewLRUCache(1024 * 1024 * 1024 /* capacity 1GB */);
```

Then wrap the `normal_block_cache` with [NewSimCache](#) and set the `SimCache` as the `block_cache` field of `rocksdb::BlockBasedTableOptions` and then generate the

```
options.table_factory :
```

```
1. rocksdb::Options options;
2. rocksdb::BlockBasedTableOptions bbt_opts;
3. std::shared_ptr<rocksdb::Cache> sim_cache =
4.     NewSimCache(normal_block_cache,
5.      10 * 1024 * 1024 * 1024 /* sim_capacity 10GB */);
6. bbt_opts.block_cache = sim_cache;
7. options.table_factory.reset(new BlockBasedTableFactory(bbt_opts));
```

Finally, open the DB with `options`. Then the HIT/MISS value of SimCache can be acquired by calling `sim_cache->get_hit_counter()` and `sim_cache->get_miss_counter()`, respectively. Alternatively, if you don't want to store `sim_cache` and using Rocksdb ( $\geq v4.12$ ), you can get these statistics through Tickers `SIM_BLOCK_CACHE_HIT` and `SIM_BLOCK_CACHE_MISS` in [rocksdb::Statistics](#).

## Memory Overhead

People may concern the actual memory usage of SimCache, which can be estimated as:

```
sim_capacity * entry_size / (entry_size + block_size),
```

- $76 \leq \text{entry\_size} (\text{key\_size} + \text{other}) \leq 104$ ,
- BlockBasedTableOptions.block\_size = 4096 by default but is configurable.

Therefore, by default the actual memory overhead of SimCache is around **sim\_capacity \* 2%**.

# Introduction

---

For a very long time, disks were the means of persistent for datastores. With the introduction of SSD, we now have a persistent medium that is significantly faster than the traditional disks but with limited write endurance and capacity, enabling us to explore the opportunities of tiered storage architecture. Open source implementations like flash cache used SSD and disk as tiered storage outperforming disk for server applications. RocksDB Persistent read cache is an effort to take advantage of the tiered storage architecture in a device agnostic and operating system independent manner for the RocksDB ecosystem.



## Tiered Storage Vs Tiered Cache

---

RocksDB users can take advantage of the tiered storage architecture either by adopting tiered storage deployment approach or by adopting tiered cache deployment approach. With the tiered storage approach, you can distribute the contents of the LSM on multiple persistent storage tiers. With the tiered cache approach, users can use the faster persistent medium as a read cache serving frequently accessed parts of the LSM and enhance the overall performance of RocksDB.

Tiered cache has a few advantage in terms of data mobility since the cache is an add-on for performance. The store can continue to function without the cache.



## Key Features

---

### Hardware agnostic

The persistent read cache is a generic implementation and is not specifically designed for any kind of device in particular. Instead of designing for specific types of hardware, we have taken the approach of designing the cache to provide the user with a mechanism to describe the best way to access the device, and the IO paths will be configured to work as per the description.

Write code path can be described using the formula

{ Block Size, Queue depth, Access/Caching Technique }



Read code path can be described using the formula

{ Access/Caching Technique }



*Block Size* describes the size to read/write. In the case of SSDs, this would typically be erasure block size.

*Queue depth* is the parallelism at which the device exhibits the best performance.

*Access/Caching Technique* is used to describes the best way to access the device. Using direct IO access for example is suitable for certain devices/applications and buffered access is preferred for others.

## OS agnostic

Persistent read cache is build using RocksDB abstraction and is supported on all platforms where RocksDB is supported.

## Pluggable

Since this is a cache implementation, the cache may or may not be supplied on a restart.

# Design and Implementation Details

---

The implementation of Persistent Read Cache has three fundamental components.

## Block Lookup Index

This is a scalable in-memory hash index that maps a given LSM block address to a cache record locator. The cache record locator helps locate the block data in the cache. The cache record can be described as { file-id, offset, size }.



## File Lookup Index / LRU

The is a scalable in-memory hash index which allows for eviction based on LRU. This index maps a given file identifier to its reference object abstraction. The object abstraction can be used for reading data from the cache. When we run out of space on the persistent cache, we evict the least recently used file from this index.



## File Layout

The cache is stored in the file system as a sequence of files. Each file contains a sequence of records which contain data corresponding to a block on RocksDB LSM.



## API

---

Please follow the link below for the public API.

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/persistent\\_cache.h](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/persistent_cache.h)

This document is aimed at engineers familiar with the RocksDB codebase and conventions.

## Tombstone Fragments

---

The DeleteRange API does not provide any restrictions on the ranges it can delete (though if `start >= end`, the deleted range will be considered empty). This means that ranges can overlap and cover wildly different numbers of keys. The lack of structure in the range tombstones created by a user make it impossible to binary search range tombstones. For example, suppose a DB contained the range tombstones `[c, d)@4`, `[g, h)@7`, and `[a, z)@10`, and we were looking up the key `e@1`. If we sorted the tombstones by start key, then we would select `[c, d)`, which doesn't cover `e`. If we sorted the tombstones by end key, then we would select `[g, h)`, which doesn't cover `e`. However, we see that `[a, z)` does cover `e`, so in both cases we've looked at the wrong tombstone. If we left the tombstones in this form, we would have to scan through all of them in order to find a potentially covering tombstone. This linear search overhead on the read path becomes very costly as the number of range tombstones grows.

However, it is possible to transform these tombstones using the following insight: one range tombstone is equivalent to several contiguous range tombstones at the same sequence number; for example, `[a, d)@4` is equivalent to `[a, b)@4; [b, d)@4`. Using this fact, we can always “fragment” a list of range tombstones into an equivalent set of tombstones that are non-overlapping. From the example earlier, we can convert `[c, d)@4`, `[g, h)@7`, and `[a, z)@10` into: `[a, c)@10`, `[c, d)@10`, `[c, d)@4`, `[d, g)@10`, `[g, h)@10`, `[g, h)@7`, `[h, z)@10`.

Now that the tombstone fragments do not overlap, we can safely perform a binary search. Going back to the `e@1` example, binary search would find `[d, g)@10`, which covers `e@1`, thereby giving the correct result.

## Fragmentation Algorithm

---

See [db/range\\_tombstone\\_fragmenter.cc](#).

## Write Path

---

## Memtable

---

When `DeleteRange` is called, a kv is written to a dedicated memtable for range tombstones. The format is `start : end` (i.e., `start` is the key, `end` is the value). Range tombstones are not fragmented in the memtable directly, but are instead fragmented each time a read occurs.

See [Compaction](#) for details on how range tombstones are flushed to SSTs.

## SST Files

Just like in memtables, range tombstones in SSTs are not stored inline with point keys. Instead, they are stored in a dedicated meta-block for range tombstones. Unlike in memtables, however, range tombstones are fragmented and cached along with the table reader when the table reader is first created. Each SST file contains all range tombstones at that level that cover user keys overlapping with the file's key range; this greatly simplifies iterator seeking and point lookups.

See [Compaction](#) for details on how range tombstones are compacted down the LSM tree.

## Read Path

Due to the simplicity of the write path, the read path requires more work.

## Point Lookups

When a user calls `Get` and the point lookup progresses down the LSM tree, the range tombstones in the table being searched are first fragmented (if not fragmented already) and binary searched before checking the file's contents. This is done through `FragmentedRangeTombstoneIterator::MaxCoveringTombstoneSeqnum` (see [db/range\\_tombstone\\_fragmenter.cc](#)); if a tombstone is found (i.e., the return value is non-zero), then we know that there is no need to search lower levels since their merge operands / point values are deleted. We check the current level for any keys potentially written after the tombstone fragment, process the merge operands (if any), and return. This is similar to how finding a point tombstone stops the progression of a point lookup.

## Range Scans

The key idea for reading range deletions during point scans is to create a structure resembling the merging iterator used by `DBIter` for point keys, but for the range tombstones in the set of tables being examined.

Here is an example to illustrate how this works for forward scans (reverse scans are similar):

Consider a DB with a memtable containing the tombstone fragment `[a, b)@40, [a, b)@35`, and 2 L0 files `1.sst` and `2.sst`. `1.sst` contains the tombstone fragments `[a, c)@15, [d, f)@20`, and `2.sst` contains the tombstone fragments `[b, e)@5, [e, x)@10`. Aside from the merging iterator of point keys in the memtable and SST files, we also keep track of the following 3 data structures:

1. a min-heap of fragmented tombstone iterators (one iterator per table) ordered by

- end key (*active heap*)
- 2. an ordered set of fragmented tombstone iterators ordered by sequence number (*active seqnum set*)
- 3. a min-heap of tombstones ordered by start key (*inactive heap*)

The active heap contains all iterators pointing at tombstone fragments that cover the most recent (internal) lookup key, the active seqnum set contains the the iterators that are in the active heap (though for brevity we will only write out the seqnums), and the inactive heap contains iterators pointing at tombstone fragments that start after the most recent lookup key. Note that an iterator is not allowed to be in both the active and inactive set.

Suppose the internal merging iterator in `DBIter` points to the internal key `a@4`. The active iterators would be the tombstone iterator for `1.sst` pointing at `[a, c)@15` and the tombstone iterator for the memtable pointing at `[a, b)@40`, and the only inactive iterator would be the tombstone iterator for `2.sst` pointing at `[b, e)@5`. The active seqnum set would contain `{40, 15}`. From these data structures, we know that the largest covering tombstone has a seqnum of 40, which is larger than 4; hence, `a@4` is deleted and we need to check another key.

Next, suppose we then check `b@50`. In response, the active heap now contains the iterators for `1.sst` pointing at `[a, c)@15` and `2.sst` pointing at `[b, e)@5`, the inactive heap contains nothing, and the active seqnum set contains `{15, 5}`. Note that the memtable iterator is now out of scope and is not tracked by these data structures. (Even though the memtable iterator had another tombstone fragment, the fragment was identical to the previous one except for its seqnum (which is smaller), so it was skipped.) Since the largest seqnum is 15, `b@50` is not covered.

For implementation details, see [db/range\\_del\\_aggregator.cc](#).

## Compaction

---

During flushes and compactions, there are two primary operations that range tombstones need to support:

1. Identifying point keys in a “snapshot stripe” (the range of seqnums between two adjacent snapshots) that can be deleted.
2. Writing range tombstones to an output file.

For (1), we use a similar data structure to what was described for range scans, but create one for each snapshot stripe. Furthermore, during iteration we skip tombstones whose seqnums are out of the snapshot stripe’s range.

For (2), we create a merging iterator out of all the fragmented tombstone iterators within the output file’s range, create a new iterator by passing this to the fragmenter, and writing out each of the tombstone fragments in this iterator to the table builder.

For implementation details, see [db/range\\_del\\_aggregator.cc](#).

## File Boundaries

In a database with only point keys, determining SST file boundaries is straightforward: simply find the smallest and largest user key in the file. However, with range tombstones, the story gets more complicated. A simple policy would be to allow the start and end keys of a range tombstone to act as file boundaries (where the start key's sequence number would be the tombstone's sequence number, and the end key's sequence number would be `kMaxSequenceNumber`). This works fine for L0 files, but for lower levels, we need to ensure that files cover disjoint ranges of internal keys. If we encounter particularly large range tombstones that cover many keys, we would be forced to create excessively large SST files. To prevent this, we instead restrict the largest (user) key of a file to be no greater than the smallest user key of the next file; if a range tombstone straddles the two files, this largest key's sequence number and type are set to `kMaxSequenceNumber` and `kTypeRangeDeletion`. In effect, we pretend that the tombstone does not extend past at the beginning of the next file.

A noteworthy edge case is when two adjacent files have a user key that “straddles” the two files; that is, the largest user key of one file is the smallest user key of the next file. (This case is uncommon, but can happen if a snapshot is preventing an old version of a key from being deleted.) In this case, even if there are range tombstones covering this gap in the level, we do not change the file boundary using the range tombstone convention described above, since this would make the largest (internal) key smaller than it should be.

## Range Tombstone Truncation

In the past, if a compaction input file contained a range tombstone, we would also add all other files containing the same range tombstone to the compaction. However, for particularly large range tombstones (commonly created from operations like dropping a large table in SQL), this created unexpectedly large compactations. To make these sorts of compactations smaller, we allowed a compaction “clean cut” to stop at a file whose largest key had a “range tombstone footer” (sequence number and type of `kMaxSequenceNumber` and `kTypeRangeDeletion`).

Unfortunately, this conflicted with another RocksDB optimization that set all key seqnums to 0 in the bottommost level if there was only one version of each key (in that level). Consider the following example:

1. A DB has three levels (`Lmax == L2`) and no snapshots, with a range tombstone `[a, f)@10` in L1, and two files `1.sst` (point internal key range `[a@5-c@3]`) and `2.sst` (point internal key range `[e@20-g@7]`) in L1.
2. If `2.sst` is compacted down to L2, its internal key `e@20` has its seqnum set to 0.
3. A user creates an iterator and reaches the `e@0` (formerly `e@20`) key. The tombstone `[a, f)@10` in `1.sst` is considered active, and since its seqnum is greater

than 0, it is considered to have covered `e`; this is an error as the key should be exposed.

To prevent this, we needed to find a way to prevent the tombstone in `1.sst` from covering keys that were in `2.sst`; more generally, we needed to prevent range tombstones from being able to cover keys outside of their file range. This problem ended up having some very complex edge cases (see the tests in `db/db_range_del_test.cc` for some examples), so the solution has two parts: *internal key range tombstone boundaries* and *atomic compaction units*.

## Internal Key Range Tombstone Boundaries

---

With the goal of preventing range tombstones from covering keys outside of their range, a simple idea would be to truncate range tombstones at the user key boundaries of its file. However, range tombstones are end-key-exclusive, so even if they should cover the largest key in a file, truncating them at that key will prevent them from doing so. Although the file boundary extension logic discussed earlier prevents this problem in many cases, when a user key straddles adjacent SST files, the largest key in the smaller of the two files is a real point key and therefore unsuitable as a user key truncation point.

A less intuitive idea is to use internal keys as tombstone boundaries. Under this extension, a key is covered if it's included in the internal key range and has a seqnum less than the tombstone's seqnum (not the boundary seqnums). To provide expected behaviour, untruncated tombstones will have seqnums of `kMaxSequenceNumber` on the start and end keys. In the aforementioned edge case where the largest key straddles two SST files, we simply use the internal key with seqnum one less than the actual largest key as the tombstone boundary. (If the seqnum is 0 or it's an artificially extended boundary (from range tombstones), then we ignore this step: see the comments on `TruncatedRangeDelIterator` in `db/range_del_aggregator.cc` for an explanation.)

## Atomic Compaction Units

---

Even though we use internal keys in-memory as range tombstone boundaries, we still have to use user keys on disk, since that's what the original format requires. This means that we have to convert an internal key-truncated tombstone to a user key-truncated tombstone during compaction. However, with just user keys, we run the risk of a similar problem to the one described earlier, where a tombstone that actually covers the largest key in the file is truncated to not cover it in a compaction output file.

Broadly speaking, a solution to this problem would involve truncating the tombstone past the largest key of its file during compaction. One simple idea would be to use the compaction boundaries at the input level. However, because we can stop a compaction before the end of a tombstone (the primary motivation for these changes),

we can run into a situation like the following:

1. A DB has three levels ( $L_{max} == L_2$ ) with no snapshots, and a tombstone `[a, g)@10` is split between two  $L_1$  files: `1.sst` (point key range `[a@4-c@6]`) and `2.sst` (point key range `[e@12-k@7]`).
2. A  $L_1$ - $L_2$  compaction is issued to compact `2.sst` down to  $L_2$ . The key `e@12` is retained in output file `3.sst` and has its seqnum set to 0.
3. Another  $L_1$ - $L_2$  compaction is issued later to compact `1.sst` down to  $L_2$ , with key range `[a-f)`, which also pulls in `3.sst`. The range tombstone is truncated to `[a, f)@10`, so the `e@0` key (formerly `e@12`) is considered covered by the tombstone, which is incorrect.

The root of this problem is that a tombstone duplicated across files can be in different compactions with overlapping boundaries. To solve this, we must pick boundaries that are disjoint for *all* compactations involving those files, which we call *atomic compaction unit boundaries*. These boundaries exactly span one or more files which have overlapping boundaries (a file with a range tombstone end key as its largest key is not considered to overlap with the next file), and are computed at compaction time. Using atomic compaction unit boundaries, we allow the end key of a file to be included in a truncated range tombstone's boundaries, while also preventing those boundaries from including keys from previous compactions.

For implementation details, see [db/compaction.cc](#).

# Unordered Writes

This document summarizes the design of unordered\_write feature, and explains how WritePrepared transactions do still provide ordering in presence of `unordered_write=true`. The feature offers 40-130% higher write throughput compared to vanilla rocksdb.

## Background

When RocksDB executes write requests coming from concurrent write threads, it groups the write threads, assigns order to them, optionally writes them to WAL, and then performs each of the writes to memtables, either serially or concurrently (when `allow_concurrent_memtable_write=true`). The leader of the write group then waits for the writes to finish, updates the `last_visible_seq` to the last sequence number in the write group, and then let individual write threads to resume. At this point the next write group could be formed. When the user takes a snapshot of DB, it will be given the `last_visible_seq`. When the user reads from the snapshot, it only reads values with a sequence number  $\leq$  the snapshot's sequence number. This simple design offers powerful guarantees:

- Atomic reads: Either all of a write batch is visible to reads or none of it. This is thanks to the write group advancing the last visible sequence number to the end of the write batch in the group (in contrast to in the middle of a write batch).
- Read-your-own writes: When a write thread returns to the user, a subsequent read by the same thread will be able to see its own writes. This is because the write thread doesn't return until the leader of the write group updates the last visible sequence number to be larger or equal to the sequence number that is assigned to the write batch of the write thread.
- Immutable Snapshots: Since `last_visible_seq` is not advanced until all the writes in the write group are finished, the reads visible to the snapshot are immutable in the sense that it will not be affected by any in-flight or future writes as they would be performed with a sequence number larger than that of the snapshot.

The downside of this approach is that the entire write group has to wait for the slowest memtable write to finish before i) it can return to the user, or ii) the next write group can be formed. This negatively impacts the rocksdb's write throughput.

## Ordering in WritePrepared Txns

When configured with `two_write_queues`, `non-2pc` writes in WritePrepared Txns are performed in two steps:

1. The write batch is written to the underlying DB with the sequence number

prepare\_seq via the main write queue. The write group advances last\_visible\_seq as usual.

2. The prepare\_seq → commit\_seq pair is added to a commit table via a 2nd write queue. The commit\_seq is a normal sequence number allocated within the write queue. The write group advances a new global variable called last\_published\_seq to be the last commit\_seq in the write group.

The snapshots are fed with last\_published\_seq (in contrast with last\_visible\_seq). The reads will return that latest value with a commit\_seq <= the snapshot sequence number. The commit\_seq is obtained by doing a lookup in the commit table using the sequence number of each value. Note that the order in this approach is specified by commit\_seq, which is assigned in the 2nd write thread. The snapshot also depends on the last\_published\_seq that is updated in the 2nd write thread. Therefore the ordering that core rocksdb provides in the first write group is redundant, and without that the users of TransactionDB would still see ordered writes and enjoy reads from immutable snapshots.

## unordered\_write: Design

---

With unordered\_write=true, the writes to the main write queue in rocskdb goes through a different path:

1. Form a write group
2. The leader in the write group orders the writes, optionally persist them in the WAL, and updates last\_visible\_seq to the sequence number of the last write batch in the group.
3. The leader resumes all the individual write threads to perform their writes into memtable.
4. The next write group is formed while the memtable writes of the previous ones are still in flight.

Note that this approach still gives read-your-own-write properties but not atomic reads nor the immutable snapshot property. However as explained above, TransactionDB configured with WritePrepared transactions and two\_write\_queues is not affected by that as it uses a 2nd write queue to provide immutable snapshots to its reads.

- **read-your-own-write:** When a write thread returns to the user, its write batch is already placed in the memtables with sequence numbers lower than the snapshot sequence number. Therefore the writes a write thread is always visible to its subsequent reads.
- **immutable snapshots:** The reads can no longer benefit from immutable snapshots since a snapshot is fed with a sequence number larger than that of upcoming or in-flight writes. Therefore reads from that snapshots will see different views of the db depending on the subset of those in-flight writes that are landed by the time the read is performed.
- **atomic writes:** is no longer provided since the last\_visible\_seq which is fed to snapshot is larger than the sequence numbers of keys in upcoming insertion of the

write batch to the memtable. Therefore each prefix of the write batch that is landed on memtables is immediately visible to the readers who read from that snapshot.

If the user can tolerate the relaxed guarantee they can enjoy the higher throughput of undered\_write feature. Otherwise they would need to implement their own mechanism to advance the snapshot sequence number to a value that is guaranteed to be larger than any in-flight write. One approach is to use TransactionDB configured with WritePrpared and two\_write\_queues which would still offer considerably higher throughput than vanilla rocksdb.

## unordered\_write: Implementation

---

- If unordered\_write is true, the writes are first redirected to WriteImplWALOnly on the primary write queue where it:
  - groups the write threads
  - if a threshold is reached that requires memtable flush
    - wait on switch\_cv\_ until pending\_memtable\_writes\_.load() == 0;
    - Flush memtable
  - persists the writes in WAL if it is enabled
  - increases pending\_memtable\_writes\_ with number of write threads that will later write to memtbale
  - updates last\_visible\_seq to the last sequence number in the write group
- If the the write thread needs to write to memtable, it calls UnorderedWriteMemtable to
  - write to memtable (in concurrent with other in-flight writes)
  - decrease pending\_memtable\_writes\_
  - If (pending\_memtable\_writes\_.load() == 0) switch\_cv\_.notify\_all();

## WritePrepared Optimizations with unordered\_write

---

WritePrepared transaction by default makes use of a lock table to prevent write-write conflicts. This feature is however extra when TransactionDB is used only to provide ordering for vanilla rocksdb users, and can be disabled with TransactionDBOption::skip\_concurrency\_control=true. The only consequence of skipping concurrency control is the following anomaly after a restart, which does not seem to be problematic for vanilla rocksdb users:

- Thread t1 writes Value V1 to Key K and to K'
- Thread t2 writes Value V2 to Key K
- The two are are grouped together and written to the WAL with <t1, t2> order
- The two writes are however committed in opposite order: <Commit(t2), Commit(t1)>
- The readers might see {K=V2} or {K=V1, K'=V1} depending on their snapshot
- DB restarts

- During recovery the commit order is dictated by WAL write order: <Commit(t1) Commit(t2)>
- The reader see this as the db state: {K=V2,K'=V1} which is different than the DB state before the restart.

## Experimental Results

---

Benchmark:

```
TEST_TMPDIR=/dev/shm/ ~/db_bench --benchmarks=fillrandom --threads=32 --num=10000000 -
max_write_buffer_number=16 --max_background_jobs=64 --batch_size=8 --writes=3000000 -
level0_file_num_compaction_trigger=99999 --level0_slowdown_writes_trigger=99999 --
level0_stop_writes_trigger=99999 -enable_pipelined_write=false -disable_auto_compactions --transaction_db=true
1. --unordered_write=1 --disable_wal=1
```

Throughput with unordered\_write=true and using WritePrepared transaction

- WAL: +42%
- No-WAL: +34%

Throughput with unordered\_write=true

- WAL: +63%
- NoWAL: +131%

Note: this is an upper-bound on the improvement as it improves only the bottleneck of writing to memtable while the write throughput could also be bottlenecked by compaction speed and IO too.

## Future work

---

- Optimize WritePrepared
  - WritePrepared adds prepare\_seq to a heap. The main reason is to provide correctness for an almost impossible scenario that the commit table flows over prepare\_seq before updated with prepare\_seq → commit\_seq. The contention of the shared heap structure is currently a major bottleneck.
- Move WritePrepared engine to core rocksdb
  - To enjoy the ordering of WritePrepared, the users currently need to open their db with TransactionDB::Open. It would be more convenient if the feature is enabled with setting an option on the vanilla rocksdb engine.
- Alternative approaches to ordering
  - The gap between the write throughput of vanilla rocksdb with unordered\_write=true and the throughput of WritePrepared with unordered\_write is significant. We can explore simpler and more efficient ways of providing immutable snapshots in presence of unordered\_write feature.

## FAQ

Q: Do we need to use 2PC with WritePrepared Txns?

A: No. All the user needs to do is to open the same db with TransactionDB, and use it with the same standard API of vanilla rocksdb. The 2PC's feature of WritePrepared Txns is irrelevant here and can be ignored.

Q: 2nd write queue in WritePrepared transactions also does ordering between the commits. Why does not it suffer from the same performance problem of the main write queue in vanilla rocksdb?

A: The write to commit table is mostly as fast as updating an element in an array. Therefore it is much less vulnerable to the slow writes problem that is hurting the throughput of ordering in vanilla rocksdb.

Q: Is memtable size accurately enforced in unordered\_writes?

A: Not as much as before. From the moment that the threshold is reached until we wait for in-flight writes to finish, the memtable size could increase beyond the threshold.  
Unordered Writes: The Design

- [RocksJava Basics](#)
- [RocksJava Performance on Flash Storage](#)
- [JNI Debugging](#)
- [RocksJava API TODO](#)

RocksJava is a project to build high performance but easy-to-use Java driver for RocksDB.

RocksJava is structured in 3 layers:

1. The Java classes within the `org.rocksdb` package which form the RocksJava API. Java users only directly interact with this layer.
2. JNI code written in C++ that provides the link between the Java API and RocksDB.
3. RocksDB itself written in C++ and compiled into a native library which is used by the JNI layer.

(We try hard to keep RocksJava API in sync with RocksDB's C++ API, but it often falls behind. We highly encourage community contributions ... so please feel free to send us a Pull Request if you find yourself needing a certain API which is in C++ but not yet in Java.)

In this page you will learn the basics of RocksDB Java API.

## Getting Started

---

You may either use the pre-built Maven artifacts that we publish, or build RocksJava yourself from its source code.

### Maven

We also publish RocksJava artifacts to Maven Central, in case you just want to depend on the jar instead of building it on your own:

<https://search.maven.org/#search%7Cga%7C1%7Cg%3A%22org.rocksdb%22>.

We publish both an uber-like Jar (`rocksdbjni-X.X.X.jar`) which contains native libraries for all supported platforms alongside the Java class files, as well as smaller platform specific Jars (such as `rocksdbjni-X.X.X-linux64.jar`).

The simplest way to use RocksJava from a build system which supports Maven style dependencies is to add a dependency on RocksJava. For example, if you are using Maven:

```

1. <dependency>
2.   <groupId>org.rocksdb</groupId>
3.   <artifactId>rocksdbjni</artifactId>
4.   <version>6.6.4</version>
5. </dependency>
```

**NOTE Microsoft Windows Users:** If you are using the Maven Central compiled artifacts on Microsoft Windows, they were compiled using Microsoft Visual Studio 2015, if you don't have "Microsoft Visual C++ 2015 Redistributable" installed, then you will need to install it from <https://www.microsoft.com/en-us/download/details.aspx?id=48145>, or

otherwise build your own binaries from source code.

## Compiling from Source

To build RocksJava, you first need to set your `JAVA_HOME` environment variable to point to the location where Java SDK is installed (must be Java 1.7+). You must also have the prerequisites for your platform to compile the native library of RocksDB, see [INSTALL.md](#). Once `JAVA_HOME` is properly set and you have the prerequisites installed, simply running `make rocksdbjava` will build the Java bindings for RocksDB:

```
1. $ make -j8 rocksdbjava
```

This will generate `rocksdbjni.jar` and `librocksdbjni.so` (or `librocksdbjni.jnilib` on macOS) in the `java/target` directory under the `rocksdb` root directory. Specifically, `rocksdbjni.jar` contains the Java classes that defines the Java API for RocksDB, while `librocksdbjni.so` includes the C++ `rocksdb` library and the native implementation of the Java classes defined in `rocksdbjni.jar`.

To run the unit tests:

```
1. $ make jtest
```

[Facebook internal only] On Facebook devservers:

```
1. $ ROCKSDB_NO_FBCODE=1 make jtest
```

To clean:

```
1. $ make jclean
```

## Samples

We provided some samples [here](#), if you want to jump directly into code.

## Memory Management

Many of the Java Objects used in the RocksJava API will be backed by C++ objects for which the Java Objects have ownership. As C++ has no notion of automatic garbage collection for its heap in the way that Java does, we must explicitly free the memory used by the C++ objects when we are finished with them.

Any Java object in RocksJava that manages a C++ object will inherit from `org.rocksdb.AbstractNativeReference` which is designed to assist in managing and cleaning up any owned C++ objects when you are done with them. Two mechanisms are used for this:

1. `AbstractNativeReference#close()` .

This method should be explicitly invoked by the user when they have finished with a RocksJava object. If C++ objects were allocated and have not yet been freed then they will be released on the first invocation of this method.

To ease the use of this, this method overrides `java.lang.AutoCloseable#close()`, which enables it to be used with ARM (Automatic Resource Management) like constructs such as Java SE 7's `try-with-resources` statement.

2. `AbstractNativeReference#finalize()` .

This method is called by Java's Finalizer thread, when all strong references to the object have expired and just before the object is Garbage Collected.

Ultimately it delegates to `AbstractNativeReference#close()`. The user should however not rely on this, and instead consider it more of a last-effort fail-safe.

It will certainly make sure that owned C++ objects will be cleaned up when the Java object is collected. It does not however help to manage the memory of RocksJava as a whole, as the memory allocated on the heap in C++ for the native C++ objects backing the Java objects is effectively invisible to the Java GC process and so the JVM cannot correctly calculate the memory pressure for the GC. **Users should always explicitly call `AbstractNativeReference#close()`** on their RocksJava objects when they are done with them.

## Opening a Database

A `rocksdb` database has a name which corresponds to a file system directory. All of the contents of database are stored in this directory. The following example shows how to open a database, creating it if necessary:

```

1. import org.rocksdb.RocksDB;
2. import org.rocksdb.RocksDBException;
3. import org.rocksdb.Options;
4. ...
5. // a static method that loads the RocksDB C++ library.
6. RocksDB.loadLibrary();
7.
8. // the Options class contains a set of configurable DB options
9. // that determines the behaviour of the database.
10. try (final Options options = new Options().setCreateIfMissing(true)) {
11.
12.     // a factory method that returns a RocksDB instance
13.     try (final RocksDB db = RocksDB.open(options, "path/to/db")) {
14.
15.         // do something
16.     }
17. } catch (RocksDBException e) {
18.     // do some error handling

```

```

19. ...
20. }
21. ...

```

TIP: You may notice the `RocksDBException` class above. This exception class extends `java.lang.Exception` and encapsulates the `Status` class in the C++ rocksdb, which describes any internal errors of RocksDB.

## Reads and Writes

The database provides `put`, `remove`, and `get` methods to modify/query the database. For example, the following code moves the value stored under `key1` to `key2`.

```

1. byte[] key1;
2. byte[] key2;
3. // some initialization for key1 and key2
4.
5. try {
6.     final byte[] value = db.get(key1);
7.     if (value != null) { // value == null if key1 does not exist in db.
8.         db.put(key2, value);
9.     }
10.    db.delete(key1);
11. } catch (RocksDBException e) {
12.     // error handling
13. }

```

TIP: You can also control the `put` and `get` behavior using `WriteOptions` and `ReadOptions` by calling their polymorphic methods `RocksDB.put(WriteOptions opt, byte[] key, byte[] value)` and `RocksDB.get(ReadOptions opt, byte[] key)`.

TIP: To avoid creating a byte-array in `RocksDB.get()`, you can also use its parametric method `int RocksDB.get(byte[] key, byte[] value)` or `int RocksDB.get(ReadOptions opt, byte[] key, byte[] value)`, where the output value will be filled into the pre-allocated output buffer `value`, and its `int` returned value will indicate the actual length of the value associated with the input `key`. When the returned value is greater than `value.length`, this indicates the size of the output buffer is insufficient.

## Opening a Database with Column Families

A rocksdb database may have multiple column families. Column Families allow you to group similar key/values together and perform operations on them independently of other Column Families.

If you have worked with RocksDB before but not used Column Families explicitly, then you may be surprised to know, that in fact all of your operations were taking place on a Column Family, known as the `default` column family.

It is important to note that when working with Column Families in RocksJava, there is a very specific order of destruction that must be obeyed for the database to correctly free all resources and shutdown. The ordering is illustrated in this code example:

```
1. import org.rocksdb.RocksDB;
2. import org.rocksdb.Options;
3. ...
4. // a static method that loads the RocksDB C++ library.
5. RocksDB.loadLibrary();
6.
7. try (final ColumnFamilyOptions cf0pts = new ColumnFamilyOptions().optimizeUniversalStyleCompaction()) {
8.
9.     // list of column family descriptors, first entry must always be default column family
10.    final List<ColumnFamilyDescriptor> cfDescriptors = Arrays.asList(
11.        new ColumnFamilyDescriptor(RocksDB.DEFAULT_COLUMN_FAMILY, cf0pts),
12.        new ColumnFamilyDescriptor("my-first-columnfamily".getBytes(), cf0pts)
13.    );
14.
15.    // a list which will hold the handles for the column families once the db is opened
16.    final List<ColumnFamilyHandle> columnFamilyHandleList =
17.        new ArrayList<>();
18.
19.    try (final DBOptions options = new DBOptions()
20.         .setCreateIfMissing(true)
21.         .setCreateMissingColumnFamilies(true);
22.         final RocksDB db = RocksDB.open(options,
23.   "path/to/do", cfDescriptors,
24.   columnFamilyHandleList)) {
25.
26.        try {
27.
28.            // do something
29.
30.        } finally {
31.
32.            // NOTE frees the column family handles before freeing the db
33.            for (final ColumnFamilyHandle columnFamilyHandle :
34.                 columnFamilyHandleList) {
35.                columnFamilyHandle.close();
36.            }
37.        } // frees the db and the db options
38.    }
39. } // frees the column family options
40. ...
```

In April'14, we started building a [Java extension](#) for RocksDB. This page shows the benchmark results of RocksJava on flash storage. The benchmark results of RocksDB C++ on flash storage can be found [here](#).

## Setup

All of the benchmarks are run on the same machine. Here are the details of the test setup:

- Test with 1 billion key / value pairs. Each key is 16 bytes, and each value is 800 bytes. Total database raw size is ~1TB.
- Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz, 40 cores.
- 25 MB CPU cache, 144 GB Ram
- CentOS release 5.2 (Final).
- Experiments were run under Funtoo chroot environment.
- g++ (Funtoo 4.8.1-r2) 4.8.1
- Java(TM) SE Runtime Environment (build 1.7.0\_55-b13)
- Java HotSpot(TM) 64-Bit Server VM (build 24.55-b03, mixed mode)
- Commit [85f9bb4](#) was used in the experiment.
- 1G rocksdb block cache.
- Snappy 1.1.1 is used as the compression algorithm.
- JEMALLOC is not used.

## Bulk Load of keys in Sequential Order (Test 2)

This benchmark measures the performance of loading 1B keys into the database using RocksJava. The keys are inserted in sequential order. The database is empty at the beginning of this benchmark run and gradually fills up. No data is being read when the data load is in progress. Below is the bulk-load performance of RocksJava:

```
1. fillseq      :    2.48233 microseconds/op; 311.2 MB/s; 1000000000 ops done; 1 / 1 task(s) finished.
```

Similar to what we did in [RocksDB's C++ benchmark](#), RocksJava was configured to use multi-threaded compactations so that multiple threads could be simultaneously compacting non-overlapping key ranges in multiple levels. Our result shows that RocksJava is able to write 300+ MB/s, or ~400K writes per second.

Here is the command for bulk-loading the database using RocksJava.

```
bpl=10485760;overlap=10;mcz=0;del=300000000;levels=6;ctrig=4; delay=8; stop=12; wbn=3; mbc=20;
1. mb=67108864;wbs=134217728; dds=0; sync=false; t=1; vs=800; bs=65536; cs=1048576; of=500000; si=1000000;
```

```

./jdb_bench.sh --benchmarks=fillseq --disable_seek_compaction=true --mmap_read=false --statistics=true --
histogram=true --threads=$t --key_size=10 --value_size=$vs --block_size=$bs --cache_size=$cs --
bloom_bits=10 --compression_type=snappy --cache_numshardbits=4 --open_files=$of --verify_checksum=true --
db=/rocksdb-bench/java/b2 --sync=$sync --disable_wal=true --stats_interval=$si --compression_ratio=0.50 --
--disable_data_sync=$dds --write_buffer_size=$wbs --target_file_size_base=$mb --max_write_buffer_number=$wbn
--max_background_compactions=$mbc --level0_file_num_compaction_trigger=$ctrig --
level0_slowdown_writes_trigger=$delay --level0_stop_writes_trigger=$stop --num_levels=$levels --
delete_obsolete_files_period_micros=$del --max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --
--max_bytes_for_level_base=$bpl --use_existing_db=false --cache_remove_scan_count_limit=16 --num=10000000000
2.

```

## Random Read (Test 4)

This benchmark measures the random read performance of RocksJava with 1 Billion keys, where each key is 16 bytes and value is 800 bytes respectively. In this benchmark, RocksJava is configured with a block size of 4 KB, and Snappy compression is enabled. There were 32 threads in the benchmark application issuing random reads to the database. In addition, RocksJava is configured to verify checksums on every read.

The benchmark runs in two parts. In the first part, data was first loaded into the database by sequentially writing all the 1B keys to the database. Once the load is complete, it proceeds to the second part, in which 32 threads will be issuing random read requests concurrently. We only measure the performance of the second part.

```

readrandom      :    7.67180 micros/op; 101.4 MB/s; 1000000000 / 1000000000 found; 32 / 32 task(s)
1. finished.

```

Our result shows that RocksJava is able to read 100+ MB/s, or processes ~130K reads per second.

Here are the commands used to run the benchmark with RocksJava:

- echo "Load 1B keys sequentially into database...."

```
n=1000000000; r=1000000000; bpl=10485760;overlap=10;mcz=2;del=300000000;levels=6;ctrig=4; delay=8; stop=12;
wbn=3; mbc=20; mb=67108864;wbs=134217728; dds=1; sync=false; t=1; vs=800; bs=4096; cs=1048576; of=500000;
2. si=1000000;
```
- ./jdb\_bench.sh --benchmarks=fillseq --disable\_seek\_compaction=true --mmap\_read=false --statistics=true --

```
histogram=true --num=$n --threads=$t --value_size=$vs --block_size=$bs --cache_size=$cs --bloom_bits=10
--cache_numshardbits=6 --open_files=$of --verify_checksum=true --db=/rocksdb-bench/java/b4 --sync=$sync --
--disable_wal=true --compression_type=snappy --stats_interval=$si --compression_ratio=0.50 --
--disable_data_sync=$dds --write_buffer_size=$wbs --target_file_size_base=$mb --max_write_buffer_number=$wbn
--max_background_compactions=$mbc --level0_file_num_compaction_trigger=$ctrig --
level0_slowdown_writes_trigger=$delay --level0_stop_writes_trigger=$stop --num_levels=$levels --
delete_obsolete_files_period_micros=$del --min_level_to_compress=$mcz --
max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --max_bytes_for_level_base=$bpl --
3. use_existing_db=false
4.
```
- echo "Reading 1B keys in database in random order...."

```
bpl=10485760;overlap=10;mcz=2;del=300000000;levels=6;ctrig=4; delay=8; stop=12; wbn=3; mbc=20; mb=67108864;
6. wbs=134217728; dds=0; sync=false; t=32; vs=800; bs=4096; cs=1048576; of=500000; si=1000000;
```

```

./jdb_bench.sh --benchmarks=readrandom --disable_seek_compaction=true --mmap_read=false --statistics=true
--histogram=true --num=$n --reads=$r --threads=$t --value_size=$vs --block_size=$bs --cache_size=$cs --
bloom_bits=10 --cache_numshardbits=6 --open_files=$of --verify_checksum=true --db=/rocksdb-bench/java/b4
--sync=$sync --disable_wal=true --compression_type=none --stats_interval=$si --compression_ratio=0.50 --
disable_data_sync=$dds --write_buffer_size=$wbs --target_file_size_base=$mb --max_write_buffer_number=$wbn
--max_background_compactions=$mbc --level0_file_num_compaction_trigger=$ctrig --
level0_slowdown_writes_trigger=$delay --level0_stop_writes_trigger=$stop --num_levels=$levels --
delete_obsolete_files_period_micros=$del --min_level_to_compress=$mcz --
max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --max_bytes_for_level_base=$bp1 --
7. use_existing_db=true

```

## Multi-Threaded Read and Single-Threaded Write (Test 5)

This benchmark measures performance of randomly reading 100M keys out of database with 1B keys while there are updates being issued concurrently. Number of read keys in this benchmark is configured to 100M to shorten the experiment time. Similar to the setting we had in Test 4, each key is again 16 bytes and value is 800 bytes respectively, and RocksJava is configured with a block size of 4 KB and with Snappy compression enabled. In this benchmark, there are 32 dedicated read threads in the benchmark issuing random reads to the database while a separate thread issues random writes to the database at 10k writes per second. Below is the random read while writing performance of RocksJava:

```
1. readwhilewriting : 9.55882 microseconds/op; 81.4 MB/s; 100000000 / 100000000 found; 32 / 32 task(s) finished.
```

The result shows that RocksJava is able to read around 80 MB/s, or process ~100K reads per second, while updates are issued concurrently.

Here are the commands used to run the benchmark with RocksJava:

```

1. echo "Load 1B keys sequentially into database...."
2. dir="/rocksdb-bench/java/b5"
  num=1000000000; r=100000000; bpl=536870912; mb=67108864; overlap=10; mcz=2; del=300000000; levels=6;
  ctrig=4; delay=8; stop=12; wbn=3; mbc=20; wbs=134217728; dds=false; sync=false; vs=800; bs=4096;
3. cs=17179869184; of=500000; wps=0; si=10000000;
  ./jdb_bench.sh --benchmarks=fillseq --disable_seek_compaction=true --mmap_read=false --statistics=true --
  histogram=true --num=$num --threads=1 --value_size=$vs --block_size=$bs --cache_size=$cs --bloom_bits=10
  --cache_numshardbits=6 --open_files=$of --verify_checksum=true --db=$dir --sync=$sync --disable_wal=true
  --compression_type=snappy --stats_interval=$si --compression_ratio=0.5 --disable_data_sync=$dds --
  write_buffer_size=$wbs --target_file_size_base=$mb --max_write_buffer_number=$wbn --
  max_background_compactions=$mbc --level0_file_num_compaction_trigger=$ctrig --
  level0_slowdown_writes_trigger=$delay --level0_stop_writes_trigger=$stop --num_levels=$levels --
  delete_obsolete_files_period_micros=$del --min_level_to_compress=$mcz --
  max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --max_bytes_for_level_base=$bp1 --
4. use_existing_db=false
5. echo "Reading while writing 100M keys in database in random order...."
6. bpl=536870912;mb=67108864;overlap=10;mcz=2;del=300000000;levels=6;ctrig=4;delay=8;stop=12;wbn=3;mbc=20;wbs=134217728

```

```
./jdb_bench.sh --benchmarks=readwhilewriting --disable_seek_compaction=true --mmap_read=false --
statistics=true --histogram=true --num=$num --reads=$r --writes_per_second=10000 --threads=$t --
value_size=$vs --block_size=$bs --cache_size=$cs --bloom_bits=10 --cache_numshardbits=6 --open_files=$of
--verify_checksum=true --db=$dir --sync=$sync --disable_wal=false --compression_type=snappy --
stats_interval=$si --compression_ratio=0.5 --disable_data_sync=$dds --write_buffer_size=$wbs --
target_file_size_base=$mb --max_write_buffer_number=$wbn --max_background_compactions=$mbc --
level0_file_num_compaction_trigger=$ctrig --level0_slowdown_writes_trigger=$delay --
level0_stop_writes_trigger=$stop --num_levels=$levels --delete_obsolete_files_period_micros=$del --
min_level_to_compress=$mcz --max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --
7. max_bytes_for_level_base=$bpl --use_existing_db=true --writes_per_second=$wps
```

If you are a Java developer working with JNI code, debugging it can be particularly hard, for example if you are experiencing an unexpected SIGSEGV and don't know why.

There are several techniques which we can use to try and help get to the bottom of these:

1. Interpreting hs\_err\_pid files
2. ASAN
3. C++ Debugger

## Interpreting hs\_err\_pid files

If the JVM crashes whilst executing our native C++ code via JNI, then it will typically write an `error report file` to a file, which on Linux may be named like `/tmp/jvm-8666/hs_error.log`, or on a Mac may be named like `hs_err_pid76448.log` in the same location that the `java` process was launched from.

Such a error report file might look like (Mac):

```

1. #
2. # A fatal error has been detected by the Java Runtime Environment:
3. #
4. # SIGSEGV (0xb) at pc=0x00007ffff87283132, pid=76448, tid=5891
5. #
6. # JRE version: Java(TM) SE Runtime Environment (7.0_80-b15) (build 1.7.0_80-b15)
7. # Java VM: Java HotSpot(TM) 64-Bit Server VM (24.80-b11 mixed mode bsd-amd64 compressed oops)
8. # Problematic frame:
9. # C  [libsystem_c.dylib+0x1132]  strlen+0x12
10. #
11. # Failed to write core dump. Core dumps have been disabled. To enable core dumping, try "ulimit -c unlimited"
12. before starting Java again
13. #
14. # If you would like to submit a bug report, please visit:
15. #   http://bugreport.java.com/bugreport/crash.jsp
16. # The crash happened outside the Java Virtual Machine in native code.
17. # See problematic frame for where to report the bug.
18. #
19. -----
20. Current thread (0x00007fc3c2007800): JavaThread "main" [_thread_in_native, id=5891,
21. stack(0x000070000011a000, 0x000070000021a000)]
22.
23. siginfo: si_signo=SIGSEGV: si_errno=0, si_code=1 (SEGV_MAPERR), si_addr=0x00000000ffffffff
24.
25. Registers:
26. RAX=0x00000000ffffffff, RBX=0x00000000ffffffff, RCX=0x00000000ffffffff, RDX=0x00000000ffffffff
27. RSP=0x0000700000216450, RBP=0x0000700000216450, RSI=0x0000000000000007, RDI=0x00000000ffffffff
28. R8 =0x00000000ffffffff, R9 =0x00007fc3c143f8e8, R10=0x00000000ffffffff, R11=0x00000000102ac7f40
29. R12=0x00007fc3c288a600, R13=0x00007fc3c1442bf8, R14=0x00007fc3c288a638, R15=0x00007fc3c288a638
30. RIP=0x00007ffff87283132, EFLAGS=0x00000000000010206, ERR=0x0000000000000004

```

```

31. TRAPNO=0x000000000000000e
32.
33. Top of Stack: (sp=0x000070000216450)
34. 0x000070000216450: 000070000216490 0000000112bb538a
35. 0x000070000216460: 0000000000000000 0000000000000000
36. 0x000070000216470: 0000000000000000 00007fc3c289c800
37. 0x000070000216480: 00007fc3c288a638 00007fc3c143c0e8
38. 0x000070000216490: 0000700002166f0 0000000112bcd82c
39. 0x0000700002164a0: 303733312e34333a 00007fc3c288a608
40. 0x0000700002164b0: 000070000216ca8 697265766f636552
41. 0x0000700002164c0: 206d6f726620676e 74736566696e616d
42. 0x0000700002164d0: 4d203a656c696620 2d54534546494e41
43. 0x0000700002164e0: 00007fc3c289c800 00007fc3c143a870
44. 0x0000700002164f0: 000000000000060 00007fc3c1530e40
45. 0x000070000216500: 00007fc300000006 0000000100c2d000
46. 0x000070000216510: 00007fc3c1500000 000070000216df0
47. 0x000070000216520: 0000700000216550 0000700000216df8
48. 0x000070000216530: 0000700000216e00 00007fc3c28d7000
49. 0x000070000216540: 0000000100c30a00 0000000000000006
50. 0x000070000216550: 0000700000216590 00007fff91b94154
51. 0x000070000216560: 00007fc3c28a6808 0000000000000004
52. 0x000070000216570: 0000000100c47a00 0000000100c2d000
53. 0x000070000216580: 0000000100c48e00 000000000001400
54. 0x000070000216590: 0000700000216680 00007fff91b90ee5
55. 0x0000700002165a0: 000000000000001 00007fc3c1530e46
56. 0x0000700002165b0: 00007000002166a0 00007fff91b90a26
57. 0x0000700002165c0: 0000000000001400 0000000100c30a00
58. 0x0000700002165d0: 00007000002166c0 00007fff91b90a26
59. 0x0000700002165e0: 00007fc3c28a6808 0000000000000006
60. 0x0000700002165f0: 0000000100c47a00 0000000100c2d000
61. 0x000070000216600: 0000000000001400 0000000100c30a00
62. 0x000070000216610: 0000700000216700 00000000000006e8
63. 0x000070000216620: 0000000000001002 00000000000c31e00
64. 0x000070000216630: 0000000000001002 0000000100c48e00
65. 0x000070000216640: ff80000000001002 00000000c153ffff
66.
67. Instructions: (pc=0x00007fff87283132)
68. 0x00007fff87283112: 0e 01 f3 0f 7f 44 0f 01 5d c3 90 90 90 90 55 48
69. 0x00007fff87283122: 89 e5 48 89 f9 48 89 fa 48 83 e7 f0 66 0f ef c0
70. 0x00007fff87283132: 66 0f 74 07 66 0f d7 f0 48 83 e1 0f 48 83 c8 ff
71. 0x00007fff87283142: 48 d3 e0 21 c6 74 17 0f bc c6 48 29 d7 48 01 f8
72.
73. Register to memory mapping:
74.
75. RAX=0xffffffffffffffff is an unknown value
76. RBX=0xffffffffffffffff is an unknown value
77. RCX=0xffffffffffffffff is an unknown value
78. RDX=0xffffffffffffffff is an unknown value
79. RSP=0x000070000216450 is pointing into the stack for thread: 0x00007fc3c2007800
80. RBP=0x000070000216450 is pointing into the stack for thread: 0x00007fc3c2007800
81. RS1=0xffffffff0000000000000007 is an unknown value
82. RDI=0xffffffff0000000000000000 is an unknown value
83. R8 =0xffffffff00000000fffffc is an unknown value

```

```

84. R9 =0x00007fc3c143f8e8 is an unknown value
85. R10=0x00000000ffffffffff is an unknown value
86. R11=0x0000000102ac7f40 is at entry_point+0 in (nmethod*)0x0000000102ac7e10
87. R12=0x00007fc3c288a600 is an unknown value
88. R13=0x00007fc3c1442bf8 is an unknown value
89. R14=0x00007fc3c288a638 is an unknown value
90. R15=0x00007fc3c288a638 is an unknown value
91.
92.
93. Stack: [0x000070000011a000, 0x000070000021a000], sp=0x0000700000216450, free space=1009k
94. Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
95. C [libsystem_c.dylib+0x1132] strlen+0x12
   C [librocksdbjni-osx.jnilib+0x1c38a]
96. rocksdb::InternalKeyComparator::InternalKeyComparator(rocksdb::Comparator const*)+0x4a
   C [librocksdbjni-osx.jnilib+0x3482c] rocksdb::ColumnFamilyData::ColumnFamilyData(unsigned int,
std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&,
rocksdb::Version*, rocksdb::Cache*, rocksdb::WriteBufferManager*, rocksdb::ColumnFamilyOptions const&,
97. rocksdb::DBOptions const*, rocksdb::EnvOptions const&, rocksdb::ColumnFamilySet*)+0x7c
   C [librocksdbjni-osx.jnilib+0x382dd]
   rocksdb::ColumnFamilySet::CreateColumnFamily(std::__1::basic_string<char, std::__1::char_traits<char>,
98. std::__1::allocator<char> > const&, unsigned int, rocksdb::Version*, rocksdb::ColumnFamilyOptions const&)+0x7d
   C [librocksdbjni-osx.jnilib+0x12ca6f] rocksdb::VersionSet::CreateColumnFamily(rocksdb::ColumnFamilyOptions
99. const&, rocksdb::VersionEdit*)+0xaf
   C [librocksdbjni-osx.jnilib+0x12d831]
   rocksdb::VersionSet::Recover(std::__1::vector<rocksdb::ColumnFamilyDescriptor,
100. std::__1::allocator<rocksdb::ColumnFamilyDescriptor> > const&, bool)+0xc51
   C [librocksdbjni-osx.jnilib+0x80df4]
   rocksdb::DBImpl::Recover(std::__1::vector<rocksdb::ColumnFamilyDescriptor,
101. std::__1::allocator<rocksdb::ColumnFamilyDescriptor> > const&, bool, bool, bool)+0x244
   C [librocksdbjni-osx.jnilib+0x9c0aa] rocksdb::DB::Open(rocksdb::DBOptions const&,
std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&,
std::__1::vector<rocksdb::ColumnFamilyDescriptor, std::__1::allocator<rocksdb::ColumnFamilyDescriptor> >
const&, std::__1::vector<rocksdb::ColumnFamilyHandle*, std::__1::allocator<rocksdb::ColumnFamilyHandle*> >*,
102. rocksdb::DB**)+0xb0a
   C [librocksdbjni-osx.jnilib+0x9b138] rocksdb::DB::Open(rocksdb::Options const&, std::__1::basic_string<char,
103. std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**)+0x448
   C [librocksdbjni-osx.jnilib+0x1234c] std::__1::__function::__func<rocksdb::Status (*)>(rocksdb::Options
const&, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&,
rocksdb::DB**), std::__1::allocator<rocksdb::Status (*)>(rocksdb::Options const&, std::__1::basic_string<char,
std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**), rocksdb::Status
(rocksdb::Options const&, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
> const&, rocksdb::DB**)>::operator()(rocksdb::Options const&, std::__1::basic_string<char,
104. std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**&)+0x1c
   C [librocksdbjni-osx.jnilib+0xd84b] rocksdb_open_helper(JNIEnv_*, long, _jstring*,
std::__1::function<rocksdb::Status (rocksdb::Options const&, std::__1::basic_string<char,
105. std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**)>)+0x8b
106. C [librocksdbjni-osx.jnilib+0xd9ab] Java_org_rocksdb_RocksDB_open__JLjava_lang_String_2+0x4b
107. j org.rocksdb.RocksDB.open(JLjava_lang_String;)J+0
108. j org.rocksdb.RocksDB.open(Lorg/rocksdb/Options;Ljava_lang_String;)Lorg/rocksdb/RocksDB;+9
   j
109. org.rocksdb.util.BytewiseComparatorTest.openDatabase(Ljava/nio/file/Path;Lorg/rocksdb/AbstractComparator;)Lorg/rocksdb/RocksDB;+11
110. j org.rocksdb.util.BytewiseComparatorTest.java_vs_java_directBytewiseComparator()V+37
111. v ~StubRoutines::call_stub
   V [libjvm.dylib+0x2dc898] JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*,
112. Thread*)+0x22a
113. V [libjvm.dylib+0x2dc668] JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x28
   V [libjvm.dylib+0x468428] Reflection::invoke(instanceKlassHandle, methodHandle, Handle, bool,
114. objArrayHandle, BasicType, objArrayHandle, bool, Thread*)+0x9fc
115. V [libjvm.dylib+0x46888e] Reflection::invoke_method(oopDesc*, Handle, objArrayHandle, Thread*)+0x16e

```

```

116. V [libjvm.dylib+0x329247] JVM_InvokeMethod+0x166
    j sun.reflect.NativeMethodAccessorImpl.invoke0(Ljava/lang/reflect/Method;Ljava/lang/Object;
117. [Ljava/lang/Object;)Ljava/lang/Object;+0
118. j sun.reflect.NativeMethodAccessorImpl.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;+87
119. j sun.reflect.DelegatingMethodAccessorImpl.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;+6
120. j java.lang.reflect.Method.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;+57
121. j org.junit.runners.model.FrameworkMethod$1.runReflectiveCall()Ljava/lang/Object;+15
122. j org.junit.internal.runners.model.ReflectiveCallable.run()Ljava/lang/Object;+1
    j org.junit.runners.model.FrameworkMethod.invokeExplosively(Ljava/lang/Object;
123. [Ljava/lang/Object;)Ljava/lang/Object;+10
124. j org.junit.internal.runners.statements.InvokeMethod.evaluate()V+12
    j
125. org.junit.runners.ParentRunner.runLeaf(Lorg/junit/runners/model/Statement;Lorg/junit/runner/Description;Lorg/junit/runners/ParentRunner$StatementRunnable;)V+12
    j
126. org.junit.runners.BlockJUnit4ClassRunner.runChild(Lorg/junit/runners/model/FrameworkMethod;Lorg/junit/runner/notification/RunNotifier;)V+12
    j
127. org.junit.runners.BlockJUnit4ClassRunner.runChild(Ljava/lang/Object;Lorg/junit/runner/notification/RunNotifier;)V+12
128. j org.junit.runners.ParentRunner$3.run()V+12
129. j org.junit.runners.ParentRunner$1.schedule(Ljava/lang/Runnable;)V+1
130. j org.junit.runners.ParentRunner.runChildren(Lorg/junit/runner/notification/RunNotifier;)V+44
    j
131. org.junit.runners.ParentRunner.access$000(Lorg/junit/runners/ParentRunner;Lorg/junit/runner/notification/RunNotifier;)V+12
132. j org.junit.runners.ParentRunner$2.evaluate()V+8
133. j org.junit.runners.ParentRunner.run(Lorg/junit/runner/notification/RunNotifier;)V+20
134. j org.junit.runners.Suite.runChild(Lorg/junit/runner/Runner;Lorg/junit/runner/notification/RunNotifier;)V+2
135. j org.junit.runners.Suite.runChild(Ljava/lang/Object;Lorg/junit/runner/notification/RunNotifier;)V+6
136. j org.junit.runners.ParentRunner$3.run()V+12
137. j org.junit.runners.ParentRunner$1.schedule(Ljava/lang/Runnable;)V+1
138. j org.junit.runners.ParentRunner.runChildren(Lorg/junit/runner/notification/RunNotifier;)V+44
    j
139. org.junit.runners.ParentRunner.access$000(Lorg/junit/runners/ParentRunner;Lorg/junit/runner/notification/RunNotifier;)V+12
140. j org.junit.runners.ParentRunner$2.evaluate()V+8
141. j org.junit.runners.ParentRunner.run(Lorg/junit/runner/notification/RunNotifier;)V+20
142. j org.junit.runner.JUnitCore.run(Lorg/junit/runner/Runner;)Lorg/junit/runner/Result;+37
143. j org.junit.runner.JUnitCore.run(Lorg/junit/runner/Request;)Lorg/junit/runner/Result;+5
144. j org.junit.runner.JUnitCore.run(Lorg/junit/runner/Computer;[Ljava/lang/Class;)Lorg/junit/runner/Result;+6
145. j org.junit.runner.JUnitCore.run([Ljava/lang/Class;)Lorg/junit/runner/Result;+5
146. j org.rocksdb.test.RocksJunitRunner.main([Ljava/lang/String;)V+93
147. v ~StubRoutines::call_stub
    V [libjvm.dylib+0x2dc898] JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x22a
148. V [libjvm.dylib+0x2dc668] JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x28
    V [libjvm.dylib+0x31004e] jni_invoke_static(JNIEnv_*, JavaValue*, jobject, JNICALL, _jmethodID*, JNIArgumentPusher*, Thread*)+0xe6
149. V [libjvm.dylib+0x3092d5] jni_CallStaticVoidMethodV+0x9c
150. V [libjvm.dylib+0x31c28e] checked_jni_CallStaticVoidMethod+0x16f
151. C [java+0x30fe] JavaMain+0x91d
152. C [libsystem_pthread.dylib+0x399d] _pthread_body+0x83
153. C [libsystem_pthread.dylib+0x391a] _pthread_body+0x0
154. C [libsystem_pthread.dylib+0x1351] thread_start+0xd
155. C [libsystem_pthread.dylib+0x1351] thread_start+0xd
156. C [libsystem_pthread.dylib+0x1351] thread_start+0xd
157.
158. Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
159. j org.rocksdb.RocksDB.open(JLjava/lang/String;)J+0
160. j org.rocksdb.RocksDB.open(Lorg/rocksdb/Options;Ljava/lang/String;)Lorg/rocksdb/RocksDB;+9
    j
161. org.rocksdb.util.BytewiseComparatorTest.openDatabase(Ljava/nio/file/Path;Lorg/rocksdb/AbstractComparator;)Lorg/rocksdb/RocksDB;+9

```

```

162. j  org.rocksdb.util.BytewiseComparatorTest.java_vs_java_directBytewiseComparator()V+37
163. v ~StubRoutines::call_stub
    j sun.reflect.NativeMethodAccessorImpl.invoke0(Ljava/lang/reflect/Method;Ljava/lang/Object;
164. [Ljava/lang/Object;)Ljava/lang/Object;+0
165. j sun.reflect.NativeMethodAccessorImpl.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;+87
166. j sun.reflect.DelegatingMethodAccessorImpl.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;+6
167. j java.lang.reflect.Method.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;+57
168. j org.junit.runners.model.FrameworkMethod$1.runReflectiveCall()Ljava/lang/Object;+15
169. j org.junit.internal.runners.model.ReflectiveCallable.run()Ljava/lang/Object;+1
    j org.junit.runners.model.FrameworkMethod.invokeExplosively(Ljava/lang/Object;
170. [Ljava/lang/Object;)Ljava/lang/Object;+10
171. j org.junit.internal.runners.statements.InvokeMethod.evaluate()V+12
    j
172. org.junit.runners.ParentRunner.runLeaf(Lorg/junit/runners/model/Statement;Lorg/junit/runner/Description;Lorg/juni
    j
173. org.junit.runners.BlockJUnit4ClassRunner.runChild(Lorg/junit/runners/model/FrameworkMethod;Lorg/junit/runner/notifi
    j
174. org.junit.runners.BlockJUnit4ClassRunner.runChild(Ljava/lang/Object;Lorg/junit/runner/notification/RunNotifier;)V
175. j org.junit.runners.ParentRunner$3.run()V+12
176. j org.junit.runners.ParentRunner$1.schedule(Ljava/lang/Runnable;)V+1
177. j org.junit.runners.ParentRunner.runChildren(Lorg/junit/runner/notification/RunNotifier;)V+44
    j
178. org.junit.runners.ParentRunner.access$000(Lorg/junit/runners/ParentRunner;Lorg/junit/runner/notification/RunNotif
179. j org.junit.runners.ParentRunner$2.evaluate()V+8
180. j org.junit.runners.ParentRunner.run(Lorg/junit/runner/notification/RunNotifier;)V+20
181. j org.junit.runners.Suite.runChild(Lorg/junit/runner/Runner;Lorg/junit/runner/notification/RunNotifier;)V+2
182. j org.junit.runners.Suite.runChild(Ljava/lang/Object;Lorg/junit/runner/notification/RunNotifier;)V+6
183. j org.junit.runners.ParentRunner$3.run()V+12
184. j org.junit.runners.ParentRunner$1.schedule(Ljava/lang/Runnable;)V+1
185. j org.junit.runners.ParentRunner.runChildren(Lorg/junit/runner/notification/RunNotifier;)V+44
    j
186. org.junit.runners.ParentRunner.access$000(Lorg/junit/runners/ParentRunner;Lorg/junit/runner/notification/RunNotif
187. j org.junit.runners.ParentRunner$2.evaluate()V+8
188. j org.junit.runners.ParentRunner.run(Lorg/junit/runner/notification/RunNotifier;)V+20
189. j org.junit.runner.JUnitCore.run(Lorg/junit/runner/Runner;)Lorg/junit/runner/Result;+37
190. j org.junit.runner.JUnitCore.run(Lorg/junit/runner/Request;)Lorg/junit/runner/Result;+5
191. j org.junit.runner.JUnitCore.run(Lorg/junit/runner/Computer;[Ljava/lang/Class;)Lorg/junit/runner/Result;+6
192. j org.junit.runner.JUnitCore.run([Ljava/lang/Class;)Lorg/junit/runner/Result;+5
193. j org.rocksdb.test.RocksJUnitRunner.main([Ljava/lang/String;)V+93
194. v ~StubRoutines::call_stub
195.
196. ----- P R O C E S S -----
197.
198. ... truncated for brevity!

```

## Stack

The most interesting part of the trace is likely the stack frames. For example consider this frame:

```

C [librocksdbjni-osx.jnilib+0x1c38a]
1. rocksdb::InternalKeyComparator::InternalKeyComparator(rocksdb::Comparator const*)+0x4a

```

We can see that something went wrong from a function (in this case a constructor) in `rocksdb::InternalKeyComparator`, however how do we relate these back to file and line-numbers in our source code?

We have to translate the offsets provided in the trace:

## Mac OS X

On a Mac this would look like:

```
1. $ atos -o java/target/librocksdbjni-osx.jnilib 0x1c38a
2. ava_org_rocksdb_LOGGER_setInfoLogLevel (in librocksdbjni-osx.jnilib) (loggerjnicalback.cc:152)
```

## Linux

On a Linux system this would look like:

```
1. $ addr2line -e java/target/librocksjni-linux64.so 0x1c38a
```

\*\* TODO \*\*

## ASAN

ASAN (Google Address Sanitizer) attempts to detect a whole range of memory and range issues and can be compiled into your code, at runtime, it will report some memory or buffer-range violations.

## Mac (Apple LLVM 7.3.0)

1. Set JDK 7 as required by RocksJava

```
1. export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home
```

2. Ensure a clean start:

```
1. make clean jclean
```

3. Compile the Java test suite with ASAN compiled in:

```
1. DEBUG_LEVEL=2 COMPILE_WITH_ASAN=true make jtest_compile
```

4. Execute the entire Java Test Suite:

```
1. make jtest_run
```

or for a single test (e.g. `ComparatorTest`), execute:

```
1. cd java
   java -ea -Xcheck:jni -Djava.library.path=target -cp "target/classes:target/test-classes:test-libs/junit-4.12.jar:test-libs/hamcrest-core-1.3.jar:test-libs/mockito-all-1.10.19.jar:test-libs/cglib-2.2.2.jar:test-libs/assertj-core-1.7.1.jar:target/*" org.rocksdb.test.RocksJUnitRunner
2. org.rocksdb.ComparatorTest
```

**NOTE:** if you see an error like:

```
==20705==ERROR: Interceptors are not working. This may be because AddressSanitizer is loaded too late (e.g.
1. via dlopen). Please launch the executable with:
2. DYLD_INSERT_LIBRARIES=/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang
3. "interceptors not installed" && 0
```

Then you need to first execute:

```
$ export
1. DYLD_INSERT_LIBRARIES=/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/clang
```

**NOTE:** Also: [https://bugs.llvm.org/show\\_bug.cgi?id=31861](https://bugs.llvm.org/show_bug.cgi?id=31861)

If ASAN detects an issue, you will see output similar to the following:

```
1. Run: org.rocksdb.BackupableDBOptionsTest testing now -> destroyOldData
2. Run: org.rocksdb.BackupEngineTest testing now -> deleteBackup
3. =====
==80632==ERROR: AddressSanitizer: unknown-crash on address 0x7fd93940d6e8 at pc 0x00011cebe075 bp
4. 0x70000020ffe0 sp 0x70000020ffd8
5. WRITE of size 8 at 0x7fd93940d6e8 thread T0
#0 0x11cebe074 in rocksdb::PosixLogger::PosixLogger(__sFILE*, unsigned long long (*)(), rocksdb::Env*,
6. rocksdb::InfoLogLevel) posix_logger.h:47
#1 0x11cebc847 in rocksdb::PosixLogger::PosixLogger(__sFILE*, unsigned long long (*)(), rocksdb::Env*,
7. rocksdb::InfoLogLevel) posix_logger.h:53
#2 0x11ce9888c in rocksdb::(anonymous namespace)::PosixEnv::NewLogger(std::__1::basic_string<char,
std::__1::char_traits<char>, std::__1::allocator<char> > const&, std::__1::shared_ptr<rocksdb::Logger*>)
8. env_posix.cc:574
#3 0x11c09a3e3 in rocksdb::CreateLoggerFromOptions(std::__1::basic_string<char,
std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DBOptions const&,
9. std::__1::shared_ptr<rocksdb::Logger*>) auto_roll_logger.cc:166
#4 0x11c3a8a55 in rocksdb::SanitizeOptions(std::__1::basic_string<char, std::__1::char_traits<char>,
10. std::__1::allocator<char> > const&, rocksdb::DBOptions const&) db_impl.cc:143
#5 0x11c3ac2f3 in rocksdb::DBImpl::DBImpl(rocksdb::DBOptions const&, std::__1::basic_string<char,
11. std::__1::char_traits<char>, std::__1::allocator<char> > const&) db_impl.cc:307
#6 0x11c3b38b4 in rocksdb::DBImpl::DBImpl(rocksdb::DBOptions const&, std::__1::basic_string<char,
12. std::__1::char_traits<char>, std::__1::allocator<char> > const&) db_impl.cc:350
#7 0x11c4497bc in rocksdb::DB::Open(rocksdb::DBOptions const&, std::__1::basic_string<char,
std::__1::char_traits<char>, std::__1::allocator<char> > const&,
std::__1::vector<rocksdb::ColumnFamilyDescriptor, std::__1::allocator<rocksdb::ColumnFamilyDescriptor> >
const&, std::__1::vector<rocksdb::ColumnFamilyHandle*, std::__1::allocator<rocksdb::ColumnFamilyHandle*> >*,
13. rocksdb::DB**) db_impl.cc:5665
```

```

#8 0x11c447b74 in rocksdb::DB::Open(rocksdb::Options const&, std::__1::basic_string<char,
14. std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**) db_impl.cc:5633
    #9 0x11bff8ca4 in rocksdb::Status
    std::__1::__invoke_void_return_wrapper<rocksdb::Status>::__call<rocksdb::Status (*&)(rocksdb::Options const&,
    std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**),
    rocksdb::Options const&, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
    const&, rocksdb::DB**>(rocksdb::Status (*&&)(rocksdb::Options const&, std::__1::basic_string<char,
    std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**), rocksdb::Options const&&,
    std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&&,
15. rocksdb::DB*&&) __functional_base:437
    #10 0x11bff89ff in std::__1::__function::__func<rocksdb::Status (*)>(rocksdb::Options const&,
    std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**),
    std::__1::allocator<rocksdb::Status (*)>(rocksdb::Options const&, std::__1::basic_string<char,
    std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB**), rocksdb::Status
    (rocksdb::Options const&, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
    > const&, rocksdb::DB**)>::operator()(rocksdb::Options const&, std::__1::basic_string<char,
16. std::__1::char_traits<char>, std::__1::allocator<char> > const&, rocksdb::DB*&&) functional:1437
    #11 0x11bff269b in std::__1::function<rocksdb::Status (rocksdb::Options const&,
    std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> > const&,
    rocksdb::DB* )>::operator()(rocksdb::Options const&, std::__1::basic_string<char, std::__1::char_traits<char>,
17. std::__1::allocator<char> > const&, rocksdb::DB**) const functional:1817
    #12 0x11bfd6edb in rocksdb_open_helper(JNIEnv_*, long, _jstring*, std::__1::function<rocksdb::Status
    (rocksdb::Options const&, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>
18. > const&, rocksdb::DB**)>) rocksjni.cc:37
19. #13 0x11bfd723e in Java_org_rocksdb_RocksDB_open__JLjava_lang_String_2 rocksjni.cc:55
20. #14 0x10be77757 (<unknown module>)
21. #15 0x10be6b174 (<unknown module>)
22. #16 0x10be6b232 (<unknown module>)
23. #17 0x10be654e6 (<unknown module>)
    #18 0x10b6dc897 in JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)
24. (libjvm.dylib+0x2dc897)
    #19 0x10b6dc667 in JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)
25. (libjvm.dylib+0x2dc667)
    #20 0x10b868427 in Reflection::invoke(instanceKlassHandle, methodHandle, Handle, bool, objArrayHandle,
26. BasicType, objArrayHandle, bool, Thread*) (libjvm.dylib+0x468427)
    #21 0x10b86888d in Reflection::invoke_method(oopDesc*, Handle, objArrayHandle, Thread*)
27. (libjvm.dylib+0x46888d)
    #22 0x10b729246 in JVM_InvokeMethod (libjvm.dylib+0x329246)
29. #23 0x10be77757 (<unknown module>)
30. #24 0x10be6b232 (<unknown module>)
31. #25 0x10be6b232 (<unknown module>)
32. #26 0x10be6b8e0 (<unknown module>)
33. #27 0x10be6b232 (<unknown module>)
34. #28 0x10be6b232 (<unknown module>)
35. #29 0x10be6b232 (<unknown module>)
36. #30 0x10be6b232 (<unknown module>)
37. #31 0x10be6b057 (<unknown module>)
38. #32 0x10be6b057 (<unknown module>)
39. #33 0x10be6b057 (<unknown module>)
40. #34 0x10be6b057 (<unknown module>)
41. #35 0x10be6b057 (<unknown module>)
42. #36 0x10be6b057 (<unknown module>)
43. #37 0x10be6b057 (<unknown module>)
44. #38 0x10be6b705 (<unknown module>)
45. #39 0x10be6b705 (<unknown module>)
46. #40 0x10be6b057 (<unknown module>)
47. #41 0x10be6b057 (<unknown module>)
48. #42 0x10be6b057 (<unknown module>)

```

```

49.    #43 0x10be6b057 (<unknown module>)
50.    #44 0x10be6b057 (<unknown module>)
51.    #45 0x10be6b057 (<unknown module>)
52.    #46 0x10be6b057 (<unknown module>)
53.    #47 0x10be6b057 (<unknown module>)
54.    #48 0x10be6b705 (<unknown module>)
55.    #49 0x10be6b705 (<unknown module>)
56.    #50 0x10be6b057 (<unknown module>)
57.    #51 0x10be6b057 (<unknown module>)
58.    #52 0x10be6b057 (<unknown module>)
59.    #53 0x10be6b057 (<unknown module>)
60.    #54 0x10be6b232 (<unknown module>)
61.    #55 0x10be6b232 (<unknown module>)
62.    #56 0x10be6b232 (<unknown module>)
63.    #57 0x10be6b232 (<unknown module>)
64.    #58 0x10be654e6 (<unknown module>)
      #59 0x10b6dc897 in JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)
65. (libjvm.dylib+0x2dc897)
      #60 0x10b6dc667 in JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)
66. (libjvm.dylib+0x2dc667)
      #61 0x10b71004d in jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, JNICALL, _jmethodID*,
67. JNI_ArgumentPusher*, Thread*) (libjvm.dylib+0x31004d)
68. #62 0x10b7092d4 in jni_CallStaticVoidMethodV (libjvm.dylib+0x3092d4)
69. #63 0x10b71c28d in checked_jni_CallStaticVoidMethod (libjvm.dylib+0x31c28d)
70. #64 0x109fdd0fd in JavaMain (java+0x1000030fd)
71. #65 0x7fff8df9c99c in __pthread_body (libsystem_pthread.dylib+0x399c)
72. #66 0x7fff8df9c919 in __pthread_start (libsystem_pthread.dylib+0x3919)
73. #67 0x7fff8df9a350 in thread_start (libsystem_pthread.dylib+0x1350)
74.
75. AddressSanitizer can not describe address in more detail (wild memory access suspected).
SUMMARY: AddressSanitizer: unknown-crash posix_logger.h:47 in rocksdb::PosixLogger::PosixLogger(__sFILE*,
76. unsigned long long (*()), rocksdb::Env*, rocksdb::InfoLogLevel)
77. Shadow bytes around the buggy address:
78. 0xffffb27281a80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
79. 0xffffb27281a90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80. 0xffffb27281aa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
81. 0xffffb27281ab0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
82. 0xffffb27281ac0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
83. =>0xffffb27281ad0: 00 00 00 00 00 00 00 00 00 04 00 00 00[04]00 00
84. 0xffffb27281ae0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
85. 0xffffb27281af0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
86. 0xffffb27281b00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
87. 0xffffb27281b10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
88. 0xffffb27281b20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
89. Shadow byte legend (one shadow byte represents 8 application bytes):
90. Addressable: 00
91. Partially addressable: 01 02 03 04 05 06 07
92. Heap left redzone: fa
93. Heap right redzone: fb
94. Freed heap region: fd
95. Stack left redzone: f1
96. Stack mid redzone: f2
97. Stack right redzone: f3
98. Stack partial redzone: f4

```

```

99. Stack after return:      f5
100. Stack use after scope:  f8
101. Global redzone:        f9
102. Global init order:     f6
103. Poisoned by user:      f7
104. Container overflow:    fc
105. Array cookie:          ac
106. Intra object redzone:  bb
107. ASan internal:         fe
108. Left alloca redzone:   ca
109. Right alloca redzone:  cb
110. ==80632==ABORTING
111. make[1]: *** [run_test] Abort trap: 6
112. make: *** [jtest_run] Error 2

```

The output from ASAN shows a stack-trace with file names and line numbers of our C++ code that led to the issue, hopefully this helps shed some light on where the issue is occurring and perhaps why.

Unfortunately all of those `(<unknown module>)` are execution paths inside the JVM, ASAN cannot discover them because the JVM we are using was not itself build with support for ASAN. We could attempt to build our own JVM from the OpenJDK project and include ASAN, but at the moment that process for Mac OS X seems to be broken:  
<https://github.com/hgomez/obuildfactory/issues/51>.

**\*\* TODO \*\*** Note the path of the DSO for libasan on Mac OS X:

```
/Library/Developer/CommandLineTools/usr/lib/clang/7.3.0/lib/darwin/libclang_rt.asan_osx_dynamic.dylib
```

## Linux (CentOS 7) (GCC 4.8.5)

### 1. Set JDK 7 as required by RocksJava

```

1. export JAVA_HOME="/usr/lib/jvm/java-1.7.0-openjdk"
2. export PATH="${PATH}: ${JAVA_HOME}/bin"

```

You might also need to run `sudo alternatives --config java` and select OpenJDK 7.

### 2. Ensure a clean start:

```
1. make clean jclean
```

### 3. Compile the Java test suite with ASAN compiled in:

```
1. DEBUG_LEVEL=2 COMPILE_WITH_ASAN=true make jtest_compile
```

### 4. Execute the entire Java Test Suite:

```
1. LD_PRELOAD=/usr/lib64/libasan.so.0 make jtest_run
```

or for a single test (e.g. `ComparatorTest`), execute:

```
1. cd java
LD_PRELOAD=/usr/lib64/libasan.so.0 java -ea -Xcheck:jni -Djava.library.path=target -cp
"target/classes:target/test-classes:test-libs/junit-4.12.jar:test-libs/hamcrest-core-1.3.jar:test-
libs/mockito-all-1.10.19.jar:test-libs/cglib-2.2.2.jar:test-libs/assertj-core-1.7.1.jar:target/*"
2. org.rocksdb.test.RocksJUnitRunner org.rocksdb.ComparatorTest
```

If ASAN detects an issue, you will see output similar to the following:

```
1. Run: org.rocksdb.util.BytewiseComparatorTest testing now -> java_vs_java_directBytewiseComparator
2. ASAN:SIGSEGV
3. =====
==4665== ERROR: AddressSanitizer: SEGV on unknown address 0x0000fffffff0 (pc 0x7fd481f913e5 sp 0x7fd48599e308
4. bp 0x7fd48599e340 T1)
5. AddressSanitizer can not provide additional info.
6. #0 0x7fd481f913e4 (/usr/lib64/libc-2.17.so+0x1633e4)
7. #1 0x7fd48282da65 (/usr/lib64/libasan.so.0.0.0+0xfa65)
8. #2 0x7fd481be5944 (/usr/lib64/libstdc++.so.6.0.19+0xb944)
9. #3 0x7fd3c57bcfc2 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x714fc2)
10. #4 0x7fd3c57edb07 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x745b07)
11. #5 0x7fd3c57f215d (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x74a15d)
12. #6 0x7fd3c59f3774 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x94b774)
13. #7 0x7fd3c59eb598 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x943598)
14. #8 0x7fd3c58a2c11 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x7fac11)
15. #9 0x7fd3c58c64bf (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x81e4bf)
16. #10 0x7fd3c58c5bc8 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x81dbc8)
17. #11 0x7fd3c57a7bc4 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x6ffbc4)
18. #12 0x7fd3c57a5fc5 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x6fdfc5)
19. #13 0x7fd3c579bd80 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x6f3d80)
20. #14 0x7fd3c579bef7 (/home/arettter/rocksdb/java/target/librocksdbjni-linux64.so+0x6f3ef7)
21. #15 0x7fd47c86ae97 (+0x14e97)

22. Thread T1 created by T0 here:
23. #0 0x7fd482828c3a (/usr/lib64/libasan.so.0.0.0+0xac3a)
   #1 0x7fd4823fd7cf (/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.101-
24. 2.6.6.1.e17_2.x86_64/jre/lib/amd64/jli/libjli.so+0x97cf)
   #2 0x7fd4823f8386 (/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.101-
25. 2.6.6.1.e17_2.x86_64/jre/lib/amd64/jli/libjli.so+0x4386)
   #3 0x7fd4823f8e38 (/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.101-
26. 2.6.6.1.e17_2.x86_64/jre/lib/amd64/jli/libjli.so+0x4e38)
   #4 0x4000774 (/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.101-2.6.6.1.e17_2.x86_64/jre-abrt/bin/java+0x4000774)
27. #5 0x7fd481e4fb14 (/usr/lib64/libc-2.17.so+0x21b14)

29. ==4665== ABORTING
30. make[1]: *** [run_test] Error 1
31. make[1]: Leaving directory `/home/arettter/rocksdb/java'
32. make: *** [jtest_run] Error 2
```

The addresses presented in the stack-trace from GCC ASAN on Linux, can be translated into file and line-numbers by using `addr2line`, for example:

Given the stack frame (from above):

```
1. #3 0x7fd3c57bcfc2 (/home/aretter/rocksdb/java/target/librocksdbjni-linux64.so+0x714fc2)
```

We can translate it with the command:

```
1. $ addr2line -e java/target/librocksdbjni-linux64.so 0x714fc2
2. /home/aretter/rocksdb./db/dbformat.h:126
```

## Linux (Ubuntu 16.04) (GCC 5.4.0)

1. Set JDK 7 as required by RocksJava

```
1. export JAVA_HOME="/usr/lib/jvm/java-7-openjdk-amd64"
2. export PATH="${PATH}:${JAVA_HOME}/bin"
```

You might also need to run `sudo alternatives --config java` and select OpenJDK 7.

2. Ensure a clean start:

```
1. make clean jclean
```

3. Compile the Java test suite with ASAN compiled in:

```
1. DEBUG_LEVEL=2 COMPILE_WITH_ASAN=true make jtest_compile
```

4. Execute the entire Java Test Suite:

```
1. LD_PRELOAD=/usr/lib/gcc/x86_64-linux-gnu/5.4.0/libasan.so make jtest_run
```

or for a single test (e.g. `ComparatorTest`), execute:

```
1. cd java
LD_PRELOAD=/usr/lib/gcc/x86_64-linux-gnu/5.4.0/libasan.so java -ea -Xcheck:jni -
Djava.library.path=target -cp "target/classes:target/test-classes:test-libs/junit-4.12.jar:test-
libs/hamcrest-core-1.3.jar:test-libs/mockito-all-1.10.19.jar:test-libs/cglib-2.2.2.jar:test-
libs/assertj-core-1.7.1.jar:target/*" org.rocksdb.test.RocksJUnitRunner org.rocksdb.ComparatorTest
```

## C++ Debugger

When things get desperate you can also run your RocksJava tests through the C++ debugger, to trace the C++ JNI code in RocksJava.

## lldb (Mac)

- Set JDK 7 as required by RocksJava

```
1. export JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home"
2. export PATH="${PATH}:${JAVA_HOME}/bin"
```

- Ensure a clean start:

```
1. make clean jclean
```

- Compile the RocksJava statically:

```
1. DEBUG_LEVEL=2 make rocksdbjavastatic
```

- Start LLDB with a single RocksJava test:

```
lldb -- /Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home/bin/java -ea -Xcheck:jni -
Djava.library.path=target -cp "target/classes:target/test-classes:test-libs/junit-4.12.jar:test-libs/hamcrest-
core-1.3.jar:test-libs/mockito-all-1.10.19.jar:test-libs/cglib-2.2.2.jar:test-libs/assertj-core-
1.7.1.jar:target/*" org.rocksdb.test.RocksJUnitRunner org.rocksdb.ComparatorTest
```

- Using LLDB with RocksJava:

You can then start the RocksJava test under lldb:

```
1. (lldb) run
```

You will likely need to instruct gdb not to stop on internal SIGSEGV and SIGBUS signals generated by the JVM:

```
1. (lldb) pro hand -p true -s false SIGSEGV
2. (lldb) pro hand -p true -s false SIGBUS
```

## gdb (Linux)

- Set JDK 7 as required by RocksJava

```
1. export JAVA_HOME="/usr/lib/jvm/java-7-openjdk-amd64"
2. export PATH="${PATH}:${JAVA_HOME}/bin"
```

You might also need to run `sudo alternatives --config java` and select OpenJDK 7.

- Ensure a clean start:

```
1. make clean jclean
```

### 3. Compile the RocksJava statically:

```
1. DEBUG_LEVEL=2 make rocksdbjavastatic
```

### 4. Start GDB with a single RocksJava test:

```
gdb --args java -ea -Xcheck:jni -Djava.library.path=target -cp "target/classes:target/test-classes:test-libs/junit-4.12.jar:test-libs/hamcrest-core-1.3.jar:test-libs/mockito-all-1.10.19.jar:test-libs/cglib-2.2.2.jar:test-libs/assertj-core-1.7.1.jar:target/*" org.rocksdb.test.RocksJUnitRunner  
1. org.rocksdb.ComparatorTest
```

#### 1. Using GDB with RocksJava:

You will likely need to instruct gdb not to stop on internal SIGSEGV signals generated by the JVM:

```
1. gdb> handle SIGSEGV pass noprint nostop
```

You can then start the RocksJava test under gdb:

```
1. gdb> start
```

This page sets out the known TODO items for the RocksJava API, it also shows who is thinking/working on a particular topic; Through this mechanism hopefully we can avoid duplicating effort.

## Some recent user requests (as of June 2017):

In order of priority:

1. Load Options APIs to dynamically reload the options without needing a restart.  
([#2898](<https://github.com/facebook/rocksdb/pull/2898>))
2. Merge Operator API (#2282)
3. Compaction Filter API (#2483)
4. Ability to pass in native pointers/memory to reduce the JNI overhead
5. Shared block cache across all column-families and instances. (#3623)
6. Stats (Tickers and histograms) in sync with C++ API. (in-sync as of June 2017, due to #2429 and #2209)
7. Implement Statistics.getHistogramString (#2374).

## Upcoming changes to the Java API (outdated – last updated in July 2016)

1. Adjust RocksJava Comparator implementation - We analyzed the current implementation and noticed a significant loss of performance using the current implementation. So we decided to do the following steps in order
  - Analyze which one of the comparator implementations is performing better either `DirectComparator` or `Comparator`
  - Outline a proper way to use Custom-C++-Comparators with RocksJava.
  - Remove everything but one Comparator implementation. Depending on the analysis listed above.
  - Document the performance penalties in related JavaDoc.
  - `FindShortestSeparator` and `FindShortSuccessor` shall only do something if the Java method is implemented. What's currently not the case.
1. Rework `WBWIIIterator` to use both `Slice` and `DirectSlice` (see above).
2. Introduce `final` on variables/members everywhere they are immutable.
3. Implement `ldb` for Java. For example, the ability to specify the Comparator which implemented in Java. [@adamretter](#)
4. Custom merge operator for Java. At the moment it is only possible to use merge operators which are available in C++ but not implementing custom functionality solely in Java. Decision: will not be implemented.
5. Expose an AbstractLogger. RocksDB C++ api allows to provide a custom Logger. This shall also be possible from Java side and allows to attach RocksDB logging to

- application logging facilities.
- 6. Document the performance penalties if log level is too verbose.
- 7. Port remaining functionality in `db.h` to RocksJava. [@fyrz](#)
- 8. Update Statistics/HistogramData to 3.10 [@fyrz](#)
- 9. Build isolation. Building Java API should not require building RocksDB. You should be able to use a Java API build with a separate existing RocksDB installation. The Java API native aspect will instead indirectly depend on a shared or static RocksDB lib. [@adamretter](#)
- 10. Expose optimistic locking [@fyrz](#)

## Planned changes for 4.x

---

- 1. Move on to Java-8, especially because Java-7 is EOL this year.
- 2. Look at Java 8 Long#unsigned operations.
- 3. Consider whether we should add an UnsignedLong, UnsignedInt class types of our own to prevent users from sending invalid DBOptions.
- 4. Restructure the package layout within the Java part.
- 5. Implement ARM (Automatic Resource Management) e.g. `try-with-resources` Java 7 use via `Closeable` / `AutoCloseable` for iterators, db, write batches etc. Along with this change we will remove the auto-cleanup for c++ resources using `finalize`. Instead we will throw an exception if a C++ resource is going to be finalized without freeing the native handle first. [DONE @adamretter](#)
- 6. Consider converting callbacks to lambda expressions

- [Lua CompactionFilter](#)

**Deprecated and removed since RocksDB 6.0**

RocksDB CompactionFilter offers a way to remove / filter expired key / value pairs based on custom logic in background. Now we have implemented an extension on top of it which allows users to implement their custom CompactionFilter in Lua! This feature is available in RocksDB 5.0.

## Benefits

---

Developing CompactionFilter in Lua has the following benefits:

- On-the-fly change: updating Lua Compaction Filter can be done on the fly without shutting down your RocksDB instance.
- Individual binary unit: as updating Lua Compaction Filter can be done on the fly, it no longer requires rebuilding the binary as required by the C++ compaction filter. This is a huge benefit for services built on top of RocksDB who maintains custom CompactionFilters for their customers.
- Safer: errors in CompactionFilter will no longer result in core dump. Lua engine captures all the exceptions.

## How to Use?

---

Using RocksLuaCompactionFilter is simple. All you need to do is the following steps:

- Build RocksDB with LUA\_PATH set to the root directory of Lua.
- Config RocksLuaCompactionFilterOptions with your Lua script. (more details are described in the next section)
- Use your RocksLuaCompactionFilterOptions in step 1 to construct a RocksLuaCompactionFilterFactory.
- Pass your RocksLuaCompactionFilterFactory in step 2 to ColumnFamilyOptions::compaction\_filter\_factory.

## Example

---

Here's a simple RocksLuaCompactionFilter that filter out any keys whose initial is less than `r` :

```

1.  lua::RocksLuaCompactionFilterOptions lua_opt;
2.  // removes all keys whose initial is less than 'r'
3.  lua_opt.lua_script =
4.      "function Filter(level, key, existing_value)\n"
5.      "    if key:sub(1,1) < 'r' then\n"
6.          "        return true, false, \"\"\n"
7.      "    end\n"
8.      "    return false, false, \"\"\n"
9.  "end\n"

```

```

10.    "\n"
11.    "function FilterMergeOperand(level, key, operand)\n"
12.    "  return false\n"
13.    "end\n"
14.    "function Name()\n"
15.    "  return \"KeepsAll\"\n"
16.    "end\n";
17.
18. // specify error log.
19. auto* rocks_logger = new facebook::rocks:: RocksLogger(
20.     "RocksLuaTest",
21.     true, // print error message in GLOG
22.     true, // print error message to scribe, available in LogView `RocksDB ERROR`
23.     nullptr);
24.
25. // Create RocksLuaCompactionFilter with the above lua_opt and pass it to Options
26. rocksdb::Options options;
27. options.compaction_filter_factory =
28.     std::make_shared<rocksdb::lua::RocksLuaCompactionFilterFactory>(lua_opt);
29. ...
30. // open FbRocksDB with the above options
31. rocksdb::DB* db;
32. auto status = openRocksDB(options, "RocksDBWithLua", &db);

```

## Config RocksLuaCompactionFilterOptions

Here we introduce how to config RocksLuaCompactionFilterOptions in more detail. The definition of RocksLuaCompactionFilterOptions can be found in include/rocksdb/utilities/lua/rocks\_lua\_compaction\_filter.h.

### Config the Lua Script (RocksLuaCompactionFilter::script)

The first and the most important parameter is RocksLuaCompactionFilterOptions::script, which is where your Lua compaction filter will be implemented. Your Lua script must implement the required functions, which are Name() and Filter().

```

1. // The lua script in string that implements all necessary CompactionFilter
2. // virtual functions. The specified lua_script must implement the following
3. // functions, which are Name and Filter, as described below.
4. //
5. // 0. The Name function simply returns a string representing the name of
6. //      the lua script. If there's any error in the Name function, an
7. //      empty string will be used.
8. //      --- Example
9. //          function Name()
10. //              return "DefaultLuaCompactionFilter"
11. //          end
12. //

```

```

13. //
14. // 1. The script must contains a function called Filter, which implements
15. //     CompactionFilter::Filter() , takes three input arguments, and returns
16. //     three values as the following API:
17. //
18. //     function Filter(level, key, existing_value)
19. //         ...
20. //         return is_filtered, is_changed, new_value
21. //     end
22. //
23. // Note that if ignore_value is set to true, then Filter should implement
24. // the following API:
25. //
26. //     function Filter(level, key)
27. //         ...
28. //         return is_filtered
29. //     end
30. //
31. // If there're any error in the Filter() function, then it will keep
32. // the input key / value pair.
33. //
34. // -- Input
35. // The function must take three arguments (integer, string, string),
36. // which map to "level", "key", and "existing_value" passed from
37. // RocksDB.
38. //
39. // -- Output
40. // The function must return three values (boolean, boolean, string).
41. // - is_filtered: if the first return value is true, then it indicates
42. //     the input key / value pair should be filtered.
43. // - is_changed: if the second return value is true, then it indicates
44. //     the existing_value needs to be changed, and the resulting value
45. //     is stored in the third return value.
46. // - new_value: if the second return value is true, then this third
47. //     return value stores the new value of the input key / value pair.
48. //
49. // -- Examples
50. //     -- a filter that keeps all key-value pairs
51. //     function Filter(level, key, existing_value)
52. //         return false, false, ""
53. //     end
54. //
55. //     -- a filter that keeps all keys and change their values to "Rocks"
56. //     function Filter(level, key, existing_value)
57. //         return false, true, "Rocks"
58. //     end
59.
60. std::string lua_script;

```

An optimization without using value (RocksLuaCompactionFilter::ignore\_value) In case your CompactionFilter never uses value to determine whether to keep or discard a key / value pair, then setting RocksLuaCompactionFilterOptions::ignore\_value=true and

implement the simplified Filter() API. Our result shows that this optimization can save up to 40% CPU overhead introduced by LuaCompactionFilter:

```

1. // If set to true, then existing_value will not be passed to the Filter
2. // function, and the Filter function only needs to return a single boolean
3. // flag indicating whether to filter out this key or not.
4. //
5. //     function Filter(level, key)
6. //         ...
7. //         return is_filtered
8. //     end
9. bool ignore_value = false;

```

The simplified Filter() API only takes two input arguments and returns only one boolean flag indicating whether to keep or discard the input key.

## Error Log Configuration (RocksLuaCompactionFilterOptions::error\_log)

When RocksLuaCompactionFilter hit any error, it will act as no-op and always return false for any key / value pair that cause errors. Developers can config RocksLuaCompactionFilterOptions::error\_log to log any Lua errors:

```

1. // When specified a non-null pointer, the first "error_limit_per_filter"
2. // errors of each CompactionFilter that is lua related will be included
3. // in this log.
4. std::shared_ptr<Logger> error_log;

```

Note that for each compaction job, we only log the first few Lua errors in error\_log to avoid generating too many error messages, and the number of errors it reports per compaction job can be configured via error\_limit\_per\_filter. The default number is one.

```

1. // The number of errors per CompactionFilter will be printed
2. // to error_log.
3. int error_limit_per_filter = 1;

```

## Dynamic Updating Lua Script

To update the Lua script while the RocksDB database is running, simply call the `SetScript()` API of your RocksLuaCompactionFilterFactory:

```

1. // Change the Lua script so that the next compaction after this
2. // function call will use the new Lua script.
3. void SetScript(const std::string& new_script);

```

- [Performance on Flash Storage](#)
- [In Memory Workload Performance](#)
- [Read-Modify-Write \(Merge\) Performance](#)
- [Delete A Range Of Keys](#)
- [Write Stalls](#)
- [Pipelined Write](#)
- [MultiGet Performance](#)
- [Tuning Guide](#)
- [Memory usage in RocksDB](#)
- [Speed-Up DB Open](#)
- [Implement Queue Service Using RocksDB](#)

These benchmarks measure RocksDB performance when data resides on flash storage. (The benchmarks on this page were generated in June 2020 with RocksDB 6.10.0 unless otherwise noted)

## Setup

---

All of the benchmarks are run on the same AWS instance. Here are the details of the test setup:

- Instance type: m5d.2xlarge 8 CPU, 32 GB Memory, 1 x 300 NVMe SSD.
- Kernel version: Linux 4.14.177-139.253.amzn2.x86\_64
- File System: XFS with discard enabled

To understand the performance of the SSD card, we ran an [fio](#) test and observed 117K IOPS of 4KB reads (See [Performance Benchmarks#fio test results](#) for outputs).

All tests were executed against by executing benchmark.sh with the following parameters (unless otherwise specified): NUM\_KEYS=9000000000 NUM\_THREADS=32 CACHE\_SIZE=6442450944 For long-running tests, the tests were executed with a duration of 5400 seconds (DURATION=5400)

All other parameters used the default values, unless explicitly mentioned here. Tests were executed sequentially against the same database instance. The db\_bench tool was generated via "make release".

The following test sequence was executed:

## Test 1. Bulk Load of keys in Random Order (benchmark.sh bulkload)

---

```
NUM_KEYS=9000000000 NUM_THREADS=32 CACHE_SIZE=6442450944 benchmark.sh bulkload
```

Measure performance to load 900 million keys into the database. The keys are inserted in random order. The database is empty at the beginning of this benchmark run and gradually fills up. No data is being read when the data load is in progress.

## Test 2. Random Write (benchmark.sh overwrite)

---

```
NUM_KEYS=9000000000 NUM_THREADS=32 CACHE_SIZE=6442450944 DURATION=5400 benchmark.sh  
overwrite
```

Measure performance to randomly overwrite keys into the database. The database was first created by the previous benchmark.

## Test 3. Multi-threaded read and single-threaded write (benchmark.sh readwhilewriting)

---

```
NUM_KEYS=9000000000 NUM_THREADS=32 CACHE_SIZE=6442450944 DURATION=5400 benchmark.sh
readwhilewriting
```

Measure performance to randomly read keys and ongoing updates to existing keys. The database from Test #2 was used as the starting point.

## Test 4. Random Read (benchmark.sh randomread)

---

```
NUM_KEYS=9000000000 NUM_THREADS=32 CACHE_SIZE=6442450944 DURATION=5400 benchmark.sh
randomread
```

Measure random read performance of a database.

The following shows results of these tests using various releases and parameters.

## Scenario 1: RocksDB 6.10, Different Block Sizes

---

The test cases were executed with various block sizes. The Direct I/O (DIO) test was executed with an 8K block size. In the “RL” tests, a timed rate-limited operation was placed before the reported operation. For example, between the “bulkload” and “overwrite” operations, a 30-minute “rate-limited overwrite (limited at 2MB/sec) was conducted. This timed operation was meant as a means to help guarantee any flush or other background operation happened before the “timed reported” operation, thereby creating more predictability in the percentile performance numbers.

### Test Case 1 : benchmark.sh bulkload

---

8K: Complete bulkload in 4560 seconds  
 4K: Complete bulkload in 5215 seconds  
 16K:  
 Complete bulkload in 3996 seconds  
 DIO: Complete bulkload in 4547 seconds  
 8K RL:  
 Complete bulkload in 4388 seconds

| Block | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50 | I |
|-------|---------|--------|---------|-------|--------|-------|--------|---------|-----|---|
| 8K    | 924468  | 370.3  | 0.2     | 157.1 | 157.1  | 1.0   | 167.5  | 1.1     | 0.5 | ( |
| 4K    | 853217  | 341.8  | 0.2     | 165.3 | 165.3  | 1.0   | 165.9  | 1.2     | 0.5 | ( |
| 16K   | 1027567 | 411.6  | 0.1     | 149.0 | 149.0  | 1.0   | 181.6  | 1.0     | 0.5 | ( |

|       |        |       |     |       |       |     |       |     |     |   |
|-------|--------|-------|-----|-------|-------|-----|-------|-----|-----|---|
| DIO   | 921342 | 369.0 | 0.2 | 156.6 | 156.6 | 1.0 | 167.0 | 1.1 | 0.5 | ( |
| 8K RL | 989786 | 396.5 | 0.2 | 159.4 | 159.4 | 1.0 | 179.5 | 1.0 | 0.5 | ) |

## Test Case 2 : benchmark.sh overwrite

| Block | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50   |
|-------|---------|--------|---------|-------|--------|-------|--------|---------|-------|
| 8K    | 85756   | 34.3   | 0.1     | 161.4 | 739.9  | 4.5   | 142.2  | 373.1   | 9.7   |
| 4K    | 79856   | 32.0   | 0.2     | 166.0 | 716.9  | 4.3   | 136.3  | 400.7   | 9.7   |
| 16K   | 93678   | 37.5   | 0.1     | 174.4 | 825.0  | 4.7   | 156.8  | 341.6   | 9.4   |
| DIO   | 85655   | 34.3   | 0.1     | 163.9 | 734.9  | 4.4   | 140.7  | 373.6   | 9.7   |
| 8K RL | 85542   | 34.3   | 0.1     | 161.2 | 757.8  | 4.7   | 143.6  | 748.1   | 340.5 |

## Test Case 3 : benchmark.sh readwhilewriting

| Block | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50   |
|-------|---------|--------|---------|-------|--------|-------|--------|---------|-------|
| 8K    | 89285   | 28.0   | 0.1     | 4.2   | 199.6  | 47.5  | 37.9   | 358.4   | 281.1 |
| 4K    | 116759  | 36.2   | 0.1     | 3.6   | 203.8  | 56.6  | 38.9   | 274.1   | 224.4 |
| 16K   | 64393   | 20.4   | 0.1     | 4.1   | 194.0  | 47.3  | 36.8   | 496.9   | 402.3 |
| DIO   | 98698   | 30.9   | 0.1     | 3.9   | 197.4  | 50.6  | 37.6   | 324.2   | 257.7 |
| 8K RL | 101598  | 31.9   | 0.1     | 3.2   | 97.2   | 30.3  | 18.4   | 629.9   | 587.5 |

## Test Case 4 : benchmark.sh randomread

| Block | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50   |
|-------|---------|--------|---------|-------|--------|-------|--------|---------|-------|
| 8K    | 101647  | 32.0   | 0.1     | 0.0   | 3.9    | 0     | .7     | 314.8   | 410.7 |
| 4K    | 130846  | 40.7   | 0.1     | 0.0   | 1.0    | 0     | .1     | 244.6   | 291.7 |
| 16K   | 70884   | 22.6   | 0.1     | 0.0   | 1.3    | 0     | .2     | 451.4   | 547.5 |
| DIO   | 144737  | 45.5   | 0.1     | 0.1   | 0.7    | 7.0   | .1     | 221.1   | 239.8 |
| 8K RL | 105790  | 33.4   | 0.1     | 0.0   | 0.0    | 0     | 605.0  | 683.0   | 807.9 |

## Scenario 2: RocksDB 6.10, 2K Value size, 100M Keys.

The test cases were executed with the default block size and a value size of 2K. Only 100M keys were written to the database. Complete bulkload in 2018 seconds

| Test             | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/ |
|------------------|---------|--------|---------|-------|--------|-------|--------|-------|
| bulkload         | 272448  | 537.3  | 0.1     | 85.3  | 85.3   | 1.0   | 242.6  | 3.7   |
| overwrite        | 22940   | 45.2   | 0.1     | 229.3 | 879.4  | 3.8   | 169.0  | 1394. |
| readwhilewriting | 87093   | 154.2  | 0.1     | 5.4   | 162.6  | 30.1  | 31.0   | 367.4 |
| readrandom       | 95666   | 169.9  | 0.1     | 0.0   | 0.0    | 0     | 0      | 334.5 |

## Scenario 3: Different Versions of RocksDB

These tests were executed against different versions of RocksDB, by checking out the corresponding branch and doing a “make release”.

**Test Case 1 : NUM\_KEY=900000000  
NUM\_THREADS=32 CACHE\_SIZE=6442450944  
benchmark.sh bulkload**

6.10.0: Complete bulkload in 4560 seconds 6.3.6: Complete bulkload in 4584 seconds  
6.0.2: Complete bulkload in 4668 seconds

| Version | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50 |
|---------|---------|--------|---------|-------|--------|-------|--------|---------|-----|
| 6.10.0  | 924468  | 370.3  | 0.2     | 157.1 | 157.1  | 1.0   | 167.5  | 1.1     | 0.5 |
| 6.3.6   | 921714  | 369.2  | 0.2     | 156.7 | 156.7  | 1.0   | 167.1  | 1.1     | 0.5 |
| 6.0.2   | 933665  | 374.0  | 0.2     | 158.7 | 158.7  | 1.0   | 169.2  | 1.1     | 0.5 |

**Test Case 2 : NUM\_KEY=900000000  
NUM\_THREADS=32 CACHE\_SIZE=6442450944  
DURATION=5400 benchmark.sh overwrite**

| Version | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50 |
|---------|---------|--------|---------|-------|--------|-------|--------|---------|-----|
| 6.10.0  | 85756   | 34.3   | 0.1     | 161.4 | 739.9  | 4.5   | 142.2  | 373.1   | 9.7 |
| 6.3.6   | 92328   | 37.0   | 0.2     | 174.0 | 818.4  | 4.7   | 155.4  | 346.6   | 8.9 |
| 6.0.2   | 86767   | 34.8   | 0.2     | 164.8 | 740.4  | 4.4   | 141.4  | 368.8   | 9.8 |

**Test Case 3 : NUM\_KEY=900000000  
NUM\_THREADS=32 CACHE\_SIZE=6442450944  
DURATION=5400 benchmark.sh readwhilewriting**

| Version | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50   |
|---------|---------|--------|---------|-------|--------|-------|--------|---------|-------|
| 6.10.0  | 89285   | 28.0   | 0.1     | 4.2   | 199.6  | 47.5  | 37.9   | 358.4   | 281.1 |
| 6.3.6   | 90189   | 28.6   | 0.1     | 4.1   | 213.1  | 51.9  | 40.6   | 354.8   | 288.1 |
| 6.0.2   | 90140   | 28.3   | 0.1     | 4.1   | 209.8  | 51.1  | 39.9   | 355.0   | 290.1 |

Test Case 4 : NUM\_KEYS=9000000000  
NUM\_THREADS=32 CACHE\_SIZE=6442450944  
DURATION=5400 benchmark.sh readrandom

| Version | ops/sec | mb/sec | Size-GB | L0_GB | Sum_GB | W-Amp | W-MB/s | usec/op | p50   |
|---------|---------|--------|---------|-------|--------|-------|--------|---------|-------|
| 6.10.0  | 101647  | 32.0   | 0.1     | 0.0   | 3.9    | 0     | .7     | 314.8   | 410.7 |
| 6.3.6   | 100168  | 31.8   | 0.1     | 0.0   | 0.9    | 0     | .1     | 319.5   | 411.3 |
| 6.9.2   | 101023  | 31.8   | 0.1     | 0.0   | 6.0    | 0     | 1.1    | 316.8   | 412.5 |

## Appendix

### fio test results

```
[\$ fio --randrepeat=1 --ioengine=sync --direct=1 --gtod_reduce=1 --name=test --filename=/data/test_file --
1. bs=4k --iodepth=64 --size=4G --readwrite=randread --numjobs=32 --group_reporting
2. test: (g=0): rw=randread, bs=4K-4K/4K/4K-4K, ioengine=sync, iodepth=64
3. ...
4. fio-2.14
5. Starting 32 processes
Jobs: 3 (f=3): [_(_),r(1),_(_),E(1),_(_),r(1),_(_),r(1),E(1)] [100.0% done] [445.3MB/0KB/0KB /s] [114K/0/0
6. iops] [eta 00m:00s]
7. test: (groupid=0, jobs=32): err= 0: pid=28042: Fri Jul 24 01:36:19 2020
8. read : io=131072MB, bw=469326KB/s, iops=117331, runt=285980msec
9. cpu : usr=1.29%, sys=3.26%, ctx=33585114, majf=0, minf=297
10. IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
11. submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
12. complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
13. issued : total=r=33554432/w=0/d=0, short=r=0/w=0/d=0, drop=r=0/w=0/d=0
14. latency : target=0, window=0, percentile=100.00%, depth=64
15.
16. Run status group 0 (all jobs):
17. READ: io=131072MB, aggrb=469325KB/s, minb=469325KB/s, maxb=469325KB/s, mint=285980msec, maxt=285980msec
18.
19. Disk stats (read/write):
20. nvme1n1: ios=33654742/61713, merge=0/40, ticks=8723764/89064, in_queue=8788592, util=100.00%
```

```
]$ fio --randrepeat=1 --ioengine=libaio --direct=1 --gtod_reduce=1 --name=test --filename=/data/test_file --
1. bs=4k --iodepth=64 --size=4G --readwrite=randread
2. test: (g=0): rw=randread, bs=4K-4K/4K-4K-4K, ioengine=libaio, iodepth=64
3. fio-2.14
4. Starting 1 process
5. Jobs: 1 (f=1): [r(1)] [100.0% done] [456.3MB/0KB/0KB /s] [117K/0/0 iops] [eta 00m:00s]
6. test: (groupid=0, jobs=1): err= 0: pid=28385: Fri Jul 24 01:36:56 2020
7. read : io=4096.0MB, bw=547416KB/s, iops=136854, runt= 7662msec
8. cpu : usr=22.20%, sys=48.81%, ctx=144112, majf=0, minf=73
9. IO depths : 1=0.1%, 2=0.1%, 4=0.1%, 8=0.1%, 16=0.1%, 32=0.1%, >=64=100.0%
10. submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
11. complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.1%, >=64=0.0%
12. issued : total=r=1048576/w=0/d=0, short=r=0/w=0/d=0, drop=r=0/w=0/d=0
13. latency : target=0, window=0, percentile=100.00%, depth=64
14.
15. Run status group 0 (all jobs):
16. READ: io=4096.0MB, aggrb=547416KB/s, minb=547416KB/s, maxb=547416KB/s, mint=7662msec, maxt=7662msec
17.
18. Disk stats (read/write):
19. nvme1n1: ios=1050868/1904, merge=0/1, ticks=374836/2900, in_queue=370532, util=98.70%
```

## Previous Results

- July 2018: [Performance Benchmark 201807](#)
- 2014: [Performance Benchmark 2014](#)



The goal of these benchmarks is to measure the performance of RocksDB when the data resides in RAM. The system is configured to store transaction logs on persistent storage so that data is not lost on machine reboots.

## TL;DR

---

4.5M - 7M read QPS for point lookups with sustained writes. 4M - 6M QPS prefix range scans with sustained writes.

## Setup

---

All of the benchmarks are run on the same machine. Here are common setup used for following benchmark tests. Some configs vary based on tests and will be mentioned in the corresponding subsection.

- 2 Intel Xeon E5-2660 @ 2.2GHz, total 16 cores (32 with HT)
- 20MB CPU cache, 144GB Ram
- CentOS release 6.3 (Kernel 3.2.51)
- Commit (c90d446ee7b87682b1e0ec7e0c778d25a90c6294) from master branch
- 2 max writer buffers of 128MB each
- Level style compaction
- PlainTable SST format, with BloomFilter enabled
- HashSkipList memtable format, with BloomFilter disabled
- WAL is enabled and kept on spinning disk. However, they are not archived and no DB backup is performed during the benchmark
- Each key is of size 20 bytes, each value is of size 100 bytes
- No compression
- Database is initially loaded with 500M unique keys by using db\_bench's `filluniquerandom` mode, then read performance is measured for a 7200-second run in `readwhilewriting` mode with 32 reader threads. Each reader thread issues random key request, which is guaranteed to be found. Another dedicated writer thread issues write requests in the meantime.
- Performance are measured for light write rate (10K writes/sec => ~1.2MB/sec) and moderate write rate (80K writes/sec => ~9.6 MB/sec) respectively.
- Total database size is ~73GB at steady state, data files stored in tmpfs
- Statistics is disabled during the benchmark and perf\_context level is set to 0
- `jemalloc` memory allocator

## Test 1. Point Lookup

---

In this test, whole key is indexed for fast point lookup (i.e. `prefix_extractor = NewFixedPrefixTransform(20)`). When using this setup, range query is not supported. It provides maximum throughput for point lookup queries.

## 80K writes/sec

Below is the reported read performance when write rate is set to 80K writes/sec:

```
1. readwhilewriting :      0.220 micros/op 4553529 ops/sec; (1026765999 of 1026765999 found)
```

The actual sustained write rate is found to be at ~52K writes/sec. This test is CPU-bound, with ~98.4% of CPU observed in user space and ~1.6% of CPU spent in kernel during the measurement.

Here is the command for filling up the DB:

```
./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=20 --keys_per_prefix=0 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
1. bloom_locality=1 --benchmarks=filluniquerandom --use_existing_db=0 --num=524288000 --threads=1
```

Here is the command for running readwhilewriting benchmark:

```
./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=20 --keys_per_prefix=0 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
bloom_locality=1 --duration=7200 --benchmarks=readwhilewriting --use_existing_db=1 --num=524288000 --
1. threads=32 --writes_per_second=81920
```

## 10K writes/sec

Below is the reported read performance when write rate is set to 10K writes/sec:

```
1. readwhilewriting :      0.142 micros/op 7054002 ops/sec; (1587250999 of 1587250999 found)
```

The actual sustained write rate is found to be at ~10K writes/sec. This test is CPU-bound, with ~99.5% of CPU observed in user space and ~0.5% of CPU spent in kernel during the measurement.

Here is the command for filling up the DB:

```
./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=20 --keys_per_prefix=0 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
-min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/_WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
1. bloom_locality=1 --benchmarks=fillunique-random --use_existing_db=0 --num=524288000 --threads=1
```

Here is the command for running readwhilewriting benchmark:

```
./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=20 --keys_per_prefix=0 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
-min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/_WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
bloom_locality=1 -duration=7200 --benchmarks=readwhilewriting --use_existing_db=1 --num=524288000 --
1. threads=32 --writes_per_second=10240
```

## Test 2. Prefix Range Query

In this test, a key prefix size of 12 bytes is indexed (i.e. `prefix_extractor = NewFixedPrefixTransform(12)`). `db_bench` is configured to generate approximately 10 keys per unique prefix. In this setting, `Seek()` can be performed within a given prefix as well as iteration on the returned iterator. However, behavior of scanning beyond the prefix boundary is not defined. This is a fairly common access pattern for graph data. Point lookup throughput is measured under this setting:

### 80K writes/sec

Below is the reported read performance when write rate is set to 80K writes/sec:

```
1. readwhilewriting : 0.251 micros/op 3979207 ops/sec; (893448999 of 893448999 found)
```

The actual sustained write rate is found to be at ~67K writes/sec. This test is CPU-bound, with ~98.0% of CPU observed in user space and ~2.0% of CPU spent in kernel during the measurement.

Here is the command for filling up the DB:

```

./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=12 --keys_per_prefix=10 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
-min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
1. bloom_locality=1 --benchmarks=filluniquerandom --use_existing_db=0 --num=524288000 --threads=1

```

Here is the command for running readwhilewriting benchmark:

```

./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=12 --keys_per_prefix=10 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
-min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
bloom_locality=1 -duration=7200 --benchmarks=readwhilewriting --use_existing_db=1 --num=524288000 --
1. threads=32 --writes_per_second=81920

```

## 10K writes/sec

Below is the reported read performance when write rate is set to 10K writes/sec:

```
1. readwhilewriting :      0.168 microseconds/op 5942413 ops/sec; (1334998999 of 1334998999 found)
```

The actual sustained write rate is found to be at ~10K writes/sec. This test is CPU-bound, with ~99.4% of CPU observed in user space and ~0.6% of CPU spent in kernel during the measurement.

Here is the command for filling up the DB:

```

./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=12 --keys_per_prefix=10 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
-min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/0_WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
1. bloom_locality=1 --benchmarks=filluniquerandom --use_existing_db=0 --num=524288000 --threads=1

```

Here is the command for running readwhilewriting benchmark:

```
./db_bench --db=/mnt/db/rocksdb --num_levels=6 --key_size=20 --prefix_size=12 --keys_per_prefix=10 --
value_size=100 --cache_size=17179869184 --cache_numshardbits=6 --compression_type=none --compression_ratio=1 --
min_level_to_compress=-1 --disable_seek_compaction=1 --hard_rate_limit=2 --write_buffer_size=134217728 --
max_write_buffer_number=2 --level0_file_num_compaction_trigger=8 --target_file_size_base=134217728 --
max_bytes_for_level_base=1073741824 --disable_wal=0 --wal_dir=/data/users/rocksdb/0_WAL_LOG --sync=0 --
disable_data_sync=1 --verify_checksum=1 --delete_obsolete_files_period_micros=314572800 --
max_background_compactions=4 --max_background_flushes=0 --level0_slowdown_writes_trigger=16 --
level0_stop_writes_trigger=24 --statistics=0 --stats_per_interval=0 --stats_interval=1048576 --histogram=0 --
use_plain_table=1 --open_files=-1 --mmap_read=1 --mmap_write=0 --memtablerep=prefix_hash --bloom_bits=10 --
bloom_locality=1 --duration=7200 --benchmarks=readwhilewriting --use_existing_db=1 --num=524288000 --
1. threads=32 --writes_per_second=10240
```

# Setup

We ran the benchmarks on the following machine:

- 16 CPUs, HT enabled -> 32 vCPUs, Intel Xeon E5-2660 @ 2.20GHz
- LSI 1.8TB Flash card (empty)
- 144GB RAM
- Linux 3.2.45

The goal of these benchmarks is to demonstrate the benefit of Merge operators on read-modify-write workloads, e.g. counters. Both benchmarks were executed using a single thread.

## Update random benchmark

We ran “update random” benchmark that executes 50.000.000 iterations of:

1. Read a random key
2. Write a new value to the random key

Each value was 8 bytes, simulating uint64\_t counter. Write Ahead Log was turned off.

Here are the exact benchmark parameters we used:

```
bpl=10485760;overlap=10;mcz=2;del=300000000;levels=6;ctrig=4; delay=8; stop=12; mbc=20; r=50000000; t=10;
vs=8; bs=65536; si=1000000; time ./db_bench --benchmarks=updaterandom --disable_seek_compaction=1 --
mmap_read=0 --statistics=1 --histogram=1 --num=$r --threads=$t --value_size=$vs --block_size=$bs --
db=/data/sdb/ --disable_wal=1 --stats_interval=$si --max_background_compactions=$mbc --
level0_file_num_compaction_trigger=$ctrig --level0_slowdown_writes_trigger=$delay --
level0_stop_writes_trigger=$stop --num_levels=$levels --delete_obsolete_files_period_micros=$del --
min_level_to_compress=$mcz --max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --
1. max_bytes_for_level_base=$bpl --use_existing_db=0
```

Here is the result of the benchmark:

```
1. 29.852 micros/op 33498 ops/sec; ( updates:50000000 found:45003817)
2. Total time 248 minutes and 46 seconds
```

## Merge operator update random

Using merge operator, we are able to perform read-modify-write using only one operator. In this benchmark, we performed 50.000.000 iterations of:

- Execute “uint64add” merge operator on a random key, which adds 1 to the value associated with the key

As in previous benchmark, each value was 8 bytes and Write Ahead Log was turned off.

Here are the exact benchmark parameters we used:

```
bpl=10485760;overlap=10;mcz=2;del=300000000;levels=6;ctrig=4; delay=8; stop=12; mbc=20; r=50000000; t=10;
vs=8; bs=65536; si=1000000; time ./db_bench --benchmarks=mergerandom --merge_operator=uint64add --
disable_seek_compaction=1 --mmap_read=0 --statistics=1 --histogram=1 --num=$r --threads=$t --value_size=$vs --
block_size=$bs --db=/data/sdb --disable_wal=1 --stats_interval=$si --max_background_compactions=$mbc --
level0_file_num_compaction_trigger=$ctrig --level0_slowdown_writes_trigger=$delay --
level0_stop_writes_trigger=$stop --num_levels=$levels --delete_obsolete_files_period_micros=$del --
min_level_to_compress=$mcz --max_grandparent_overlap_factor=$overlap --stats_per_interval=1 --
1. max_bytes_for_level_base=$bpl --use_existing_db=0
```

Here is the result of the benchmark:

```
1. 9.444 micros/op 105892 ops/sec; ( updates:50000000)
2. Total time 78 min and 53 sec
```

**Note:** the techniques described in this page are obsolete. For users of RocksDB 5.18+, the native `DeleteRange` function is a better alternative for all known use cases.

In many cases, people want to drop a range of keys. For example, in MyRocks, we encoded rows from one table by prefixing with table ID, so that when we need to drop a table, all keys with the prefix needs to be dropped. For another example, if we store different attributes of a user with key with the format `[user_id][attribute_id]`, then if a user deletes the account, we need to delete all the keys prefixing `[user_id]`.

The standard way of deleting those keys is to iterate through all of them and issue `Delete()` to them one by one. This approach works when the number of keys to delete is not large. However, there are two potential drawbacks of this solution:

1. the space occupied by the data will not be reclaimed immediately. We'll wait for compaction to clean up the data. This is usually a concern when the range to delete takes a significant amount of space of the database.
2. the chunk of tombstones may slow down iterators.

There are two other ways you can do to delete keys from a range:

The first way is to issue a `DeleteFilesInRange()` to the range. The command will remove all SST files only containing keys in the range to delete. For a large chunk, it will immediately reclaim most space, so it is a good solution to problem 1. One thing needs to be taken care of is that after the operations, some keys in the range may still exist in the database. If you want to remove all of them, you should follow up with other operations, but it can be done in a slower pace. Also, be aware that, `DeleteFilesInRange()` will be removed despite of existing snapshots, so you shouldn't expect to be able to read data from the range using existing snapshots any more.

Another way is to apply compaction filter together with `CompactRange()`. You can write a compaction filter that can filter out keys from deleted range. When you want to delete keys from a range, call `CompactRange()` for the range to delete. While the compaction finishes, the keys will be dropped. We recommend you to turn

`CompactionFilter::IgnoreSnapshots()` to true to make sure keys are dropped even if you have outstanding snapshots. Otherwise, you may not be able to fully remove all the keys in the range from the system. This approach can also solve the problem of reclaiming data, but it introduces more I/Os than the `DeleteFilesInRange()` approach. However, `DeleteFilesInRange()` cannot remove all the data in the range. So a better way is to first apply `DeleteFilesInRange()`, and then issue `CompactRange()` with compaction filter.

Problem 2 is a harder problem to solve. One way is to apply `DeleteFilesInRange() + CompactRange()` so that all keys and tombstones for the range are dropped. It works for large ranges, but it is too expensive if we frequently drop small ranges. The reason is that `DeleteFilesInRange()` is unlikely to delete any file and `CompactRange()` will delete far more data than needed because it needs to execute compactations against all SST files containing any key in the deletion range. For the use cases where `CompactRange()` is too expensive, there are still two ways to reduce the harm:

1. if you never overwrite existing keys, you can try to use `DB::SingleDelete()` instead

of `Delete()` to kill tombstones sooner. Tombstones will be dropped after it meets the original keys, rather than compacted to the last level.

2. use `NewCompactOnDeletionCollectorFactory()` to speed up compaction when there are chunks of tombstones.

RocksDB has extensive system to slow down writes when flush or compaction can't keep up with the incoming write rate. Without such a system, if users keep writing more than the hardware can handle, the database will:

- Increase space amplification, which could lead to running out of disk space;
- Increase read amplification, significantly degrading read performance.

The idea is to slow down incoming writes to the speed that the database can handle. However, sometimes the database can be too sensitive to a temporary write burst, or underestimate what the hardware can handle, so that you may get unexpected slowness or query timeouts.

To find out whether your DB is suffering from write stalls, you can look at:

- LOG file, which will contain info log when write stalls are triggered;
- Compaction stats found in LOG file.

## Causes of Write Stalls

---

Stalls may be triggered for the following reasons:

- **Too many memtables.** When the number of memtables waiting to flush is greater or equal to `max_write_buffer_number`, writes are fully stopped to wait for flush finishes. In addition, if `max_write_buffer_number` is greater than 3, and the number of memtables waiting for flush is greater than or equal to `max_write_buffer_number - 1`, writes are stalled. In these cases, you will get info logs in LOG file similar to:

```
Stopping writes because we have 5 immutable memtables (waiting for flush), max_write_buffer_number is set to 5
```

```
Stalling writes because we have 4 immutable memtables (waiting for flush), max_write_buffer_number is set to 5
```

- **Too many level-0 SST files.** When the number of level-0 SST files reaches `level0_slowdown_writes_trigger`, writes are stalled. When the number of level-0 SST files reaches `level0_stop_writes_trigger`, writes are fully stopped to wait for level-0 to level-1 compaction reduce the number of level-0 files. In these cases, you will get info logs in LOG file similar to

```
Stalling writes because we have 4 level-0 files
```

```
Stopping writes because we have 20 level-0 files
```

- **Too many pending compaction bytes.** When estimated bytes pending for compaction reaches `soft_pending_compaction_bytes`, writes are stalled. When estimated bytes pending for compaction reaches `hard_pending_compaction_bytes`, write are fully stopped to wait for compaction. In these cases, you will get info logs in LOG file similar to

```
Stalling writes because of estimated pending compaction bytes 500000000
```

```
Stopping writes because of estimated pending compaction bytes 1000000000
```

Whenever stall conditions are triggered, RocksDB will reduce write rate to `delayed_write_rate`, and could possibly reduce write rate to even lower than `delayed_write_rate` if estimated pending compaction bytes accumulates. One thing worth to note is that slowdown/stop triggers and pending compaction bytes limit are per-column family, and write stalls apply to the whole DB, which means if one column family triggers write stall, the whole DB will be stalled.

## Non-blocking Writes

If a write slowdown/stop is triggered, application threads that do Put/Merge/Delete etc. will block. If a slowdown is in effect, each write will sleep for sometime (typically 1ms) before proceeding. If writes are stalled, the thread can be blocked indefinitely. If blocking the thread is not desirable, applications can avoid it by setting `no_slowdown = true` in `writeOptions`. All writes with this option will be immediately returned with `Status::Incomplete()` if they could not be completed due to a slowdown/stall.

Internally, RocksDB tries to batch write requests from different threads together before writing to the WAL in order to increase performance. However, writes with `no_slowdown` set will not be batched with writes that don't have it, which might result in a slight performance hit.

## Write Stall mitigation

There are multiple options you can tune to mitigate write stalls. If you have some workload which can tolerate write stalls and some don't, you can set some writes to [Low Priority Write](#) to avoid stalling in those latency-critical writes.

If write stalls are triggered by pending flushes, you can try:

- Increase `max_background_flushes` to have more flush threads.
- Increase `max_write_buffer_number` to have smaller memtable to flush.

If write stalls are triggered by too many level-0 files or too many pending compaction bytes, compaction is not fast enough to catch up with writes. Note that anything reduce write amplification will reduce the bytes need to write by compactations, thus speeds up compaction. Options to try:

- Increase `max_background_compactions` to have more compaction threads.
- Increase `write_buffer_size` to have large memtable, to reduce write amplification.
- Increase `min_write_buffer_number_to_merge`.

You can also set stop/slowdown triggers and pending compaction bytes limits to huge

number to avoid hitting write stall. Also take a look at “What’s the fastest way to load data into RocksDB?” in our [FAQ](#) if you are bulk loading data to RocksDB.

The pipelined write feature added in RocksDB 5.5 is to improve concurrent write throughput in case WAL is enabled. By default, a single write thread queue is maintained for concurrent writers. The thread gets to the head of the queue becomes write batch group leader and responsible for writing to WAL and memtable for the batch group.

One observation is that WAL writes and memtable writes are sequential, and by making them run in parallel we can increase throughput. For one single writer WAL writes and memtable writes have to run sequentially. With concurrent writers, once the previous writer finishes its WAL write, the next writer waiting in the write queue can start writing to the WAL while the previous writer still has its memtable write ongoing. This is what pipelined writes do.

To enable pipelined write, simply set `Options.enable_pipelined_write=true`. db\_bench benchmark shows 20% write throughput improvement with concurrent writers and WAL enabled, when DB is stored in ramfs and compaction throughput is not the bottleneck.

## Benchmarks

---

We run db\_bench on tempfs with 8 threads writing concurrently with WAL enabled. Memtable is the default skip list memtable of 64MB. LZ4 compression is enabled. With pipelined write we got roughly 30% improvement on write throughput. Raw result:  
<https://gist.github.com/yiwu-arbug/3b5a5727e52f1e58d1c10f2b80cec05d>

# Introduction

There is a lot of complexity in the underlying RocksDB implementation to lookup a key. The complexity results in a lot of computational overhead, mainly due to cache misses when probing bloom filters, virtual function call dispatches, key comparisons and IO. Users that need to lookup many keys in order to process an application level request end up calling `Get()` in a loop to read the required KVs. By providing a `MultiGet()` API that accepts a batch of keys, it is possible for RocksDB to make the lookup more CPU efficient by reducing the number of virtual function calls and pipelining cache misses. Furthermore, latency can be reduced by doing IO in parallel.

## Read Path

A typical RocksDB database instance has multiple levels, with each level containing a few tens to hundreds of SST files. A point lookup goes through the following stages (in order to keep it simple, we ignore merge operands and assume everything is a Put)

-

1. The mutable memtable is looked up. If a bloom filter is configured for memtable, the filter is probed using either the whole key or prefix. If the result is positive, the memtable rep lookup happens.
2. If the key was not found, 0 or more immutable memtables are looked up using the same process as #1
3. Next, the SST files in successive levels are looked up as follows -
  - i. In L0, every SST file is looked up in reverse chronological order
  - ii. For L1 and above, each level has a vector of SST file metadata objects, with each metadata object containing, among other things, the highest and lowest key in the file. A binary search is performed in this vector to determine the file that overlaps the desired key. There is an auxiliary index that uses pre-calculated information about file ranges in the lsm to determine the set of files overlap a given file in the next level. A full binary search is performed in L1, and this index is used to narrow down the binary search bound in subsequent levels. This is known as fractional cascading.
  - iii. Once a candidate file is found, the file's bloom filter block is loaded (either from the block cache or disk) and probed for the key. The probe is likely to result in a CPU cache miss. In many cases, the bottommost level will not have a bloom filter.
  - iv. If the probe result is positive, the SST file index block is loaded and binary searched to find the target data block. The filter and index blocks may have to be read from disk, but typically they are either pinned in memory or accessed frequently enough to be found in the block cache.
  - v. The data block is loaded and binary searched to find the key. Data block lookups are more likely to miss in the block cache and result in an IO. It is important to note that each block cache lookup is also likely to result in a

- CPU cache miss, since the block cache is indexed by a hash table.
- 4. Step #3 is repeated for each level, with the only difference in L2 and higher being the fractional cascading for SST file lookup.

## MultiGet Performance Optimizations

---

Let us consider the case of a workload with good locality of reference. Successive point lookups in such a workload are likely to repeatedly access the same SST files and index/data blocks. For such workloads, MultiGet provides the following optimizations -

1. When `options.cache_index_and_filter_block=true` is set, filter and index blocks for an SST file are fetched from the block cache on each key lookup. On a system with many threads performing reads, this results in significant lock contention on the LRU mutex. MultiGet looks up the filter and index block in the block cache only once for a whole batch of keys overlapping an SST file key range, thus drastically reducing the LRU mutex contention.
2. In steps 1, 2 and 3c, CPU cache misses occur due to bloom filter probes. Assuming a database with 6 levels and most keys being found in the bottommost level, with an average of 2 L0 files, we will have ~6 cache misses due to filter lookups in SST files. There may be an additional 1-2 cache misses if memtable bloom filters are configured. By batching the lookups at each stage, the filter cache line accesses can be pipelined, thus hiding the cache miss latency.
3. In a large database, data block reads are highly likely to require IO. This introduces latency. MultiGet has the capability to issue IO requests for multiple data blocks in the same SST file in parallel, thus reducing latency. This depends on support for parallel reads in the same thread from the underlying `Env` implementation. On Linux, `PosixEnv` has the capability to do parallel IO for `MultiGet()` using the IO Uring interface. IO Uring is a new asynchronous IO implementation introduced in the Linux kernel starting from 5.1.

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:

```

1. ** 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)
2. Comp(sec) Comp(cnt) Avg(sec) Stall(sec) Stall(cnt) Avg(ms)      KeyIn   KeyDrop
-----
3.

  L0      2/0       15    0.5     0.0     0.0     0.0     32.8     32.8     0.0     0.0     0.0     23.0
4. 1457      4346    0.335    0.00     0     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
5. 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
6. 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
7. 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
8. 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
9. 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
10. 91        42      2.164    0.00     0     0.00    11718977     0
11. Flush(GB): accumulative 32.786, interval 0.091
    Stalls(secs): 0.000 level0_slowdown, 0.000 level0_numfiles, 0.000 memtable_compaction, 0.000
12. leveln_slowdown_soft, 0.000 leveln_slowdown_hard
    Stalls(count): 0 level0_slowdown, 0 level0_numfiles, 0 memtable_compaction, 0 leveln_slowdown_soft, 0
13. leveln_slowdown_hard
14.
15. ** DB Stats **

16. Uptime(secs): 128748.3 total, 300.1 interval
    Cumulative writes: 1288457363 writes, 14173030838 keys, 357293118 batches, 3.6 writes per batch, 3055.92 GB
17. user ingest, stall micros: 7067721262
18. 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,
19. stall micros: 112418835
20. 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:

```
1. options.env->SetBackgroundThreads(num_threads, Env::Priority::HIGH);
2. 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:

```
1. 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:

```
1. write_buffer_size = 512MB;
2. max_write_buffer_number = 5;
3. 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 compactations 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  $(512 \text{ MB} + 512 \text{ MB} + 5\text{GB} + 51\text{GB} + 512\text{GB}) / (500\text{GB}) = 1.14$ . 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  $\text{number\_of\_level0\_files} + \text{number\_of\_non\_empty\_levels}$ .

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.

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

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

```
1. rocksdb::BlockBasedTableOptions table_options;
2. table_options.index_type = rocksdb::BlockBasedTableOptions::kHashSearch;
3. table_options.block_size = 4 * 1024;
4. 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.

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

LZ4 compression reduces CPU usage, but increases storage space.

```
1. 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.

```
1. options.options.compaction_style = kCompactionStyleLevel;
2. options.level0_file_num_compaction_trigger = 10;
3. options.level0_slowdown_writes_trigger = 20;
4. options.level0_stop_writes_trigger = 40;
5. options.write_buffer_size = 64 * 1024 * 1024;
6. options.target_file_size_base = 64 * 1024 * 1024;
7. 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???

```
1. options.max_background_compactions = 1
2. 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 compactations occur on different shards. Otherwise, storage wouldn't be saturated with only 2 threads writing to storage.

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

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

```
1. 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????

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

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

```
1. options.options.compaction_style = kCompactionStyleLevel;
2. options.write_buffer_size = 67108864; // 64MB
3. options.max_write_buffer_number = 3;
4. options.target_file_size_base = 67108864; // 64MB
5. options.max_background_compactions = 4;
6. options.level0_file_num_compaction_trigger = 8;
7. options.level0_slowdown_writes_trigger = 17;
8. options.level0_stop_writes_trigger = 24;
9. options.num_levels = 4;
10. options.max_bytes_for_level_base = 536870912; // 512MB
11. 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 filters and reduce the delta encoding restart interval:

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

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

```
1. 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:

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

Keep all the files open:

```
1. 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 compactations, 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:

```
1. 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:

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

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

```
1. options.memtable_factory.reset(rocksdb::NewHashLinkRepFactory(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:

```
1. options.memtable_prefix_bloom_bits = 10000000;
2. 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:

```
1. options.compaction_style = kUniversalCompaction;
2. options.compaction_options_universal.size_ratio = 10;
3. options.compaction_options_universal.min_merge_width = 2;
4. options.compaction_options_universal.max_size_amplification_percent = 1;
5. options.level0_file_num_compaction_trigger = 1;
6. options.level0_slowdown_writes_trigger = 8;
7. options.level0_stop_writes_trigger = 16;
```

Tune bloom filter to minimize memory accesses:

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

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

```
1. 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.

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

Multiple DBs sharing the same compaction pool of 2:

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

Settings for WAL logs:

```
1. 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.

```
1. 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](#).

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

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

```
1. rocksdb::BlockBasedTableOptions table_options;
2. table_options.block_cache = rocksdb::NewLRUCache(1 * 1024 * 1024 * 1024LL);
3. rocksdb::Options options;
4. 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 the OS's page cache for raw file blocks, usually containing compressed data. In a way, RocksDB's cache is two-tiered: block cache and page cache. Counter-intuitively, 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 block cache is using, you can call a function `GetUsage()` on block cache object:

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

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

```
1. > 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:

- 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.
- 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.

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

- 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. An exception to `cache_index_and_filter_blocks=true` is for L0 when setting `pin_l0_filter_and_index_blocks_in_cache=true`, which can be a good compromise setting.
- 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.

However, regardless of options, by default each column family in each database instance will have its own block cache instance, with its own memory limit. To share a single block cache, set `block_cache` in your various `BlockBasedTableOptions` to use the same `shared_ptr`, or share the same `BlockBasedTableOptions` for your various factories, or share the same `BlockBasedTableFactory` for `table_factory` in your various `Options` or `ColumnFamilyOptions`, etc.

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

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

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

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

In `partitioned index/filters` the indexes and filters for each partition are always 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 unless using many column families and/or database instances. However, memtable size inversely affects write amplification: more memory to the memtable yields less write amplification. 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:

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

In MongoRocks, the equivalent call is

```
1. > 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.

Similar to block cache, by default memtable sizes are per column family, per database instance. Use a [Write Buffer Manager](#) to cap memtable memory across column families and/or database instances.

## 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:

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

When users reopen a DB, here are some steps that may take longer than expected:

## Manifest replay

---

The `MANIFEST` file contains history of all file operations of the DB since the last time DB was opened, and is replayed during DB open. If there are too many updates to replay, it takes a long time. This can happen when:

- SST files were too small so file operations were too frequently. If this is the case, try to solve the small SST file problem. Maybe memtable is flushed too often, which generates small L0 files, or target file size is too small so that compaction generates small files. You can try to adjust the configuration accordingly
- DB simply runs for too long and accumulates too many historic updates. Either way, you can try to set `options.max_manifest_file_size` to force a new manifest file to be generated when it hits the maximum size, to avoid replaying for too long.

## WAL replaying

---

All the WAL files are replayed if it contains any useful data.

If your memtable size is large, the replay can be long. So try to shrink the memtable size.

Another common reason that WAL files to replay is too large is that, one of the column families gets too slow writes, which holds logs from being deleted. When DB reopens, all those log files are read, just to replay updates from this column family. In this case, set a proper `options.max_total_wal_size` value. The low-traffic column families will be flushed to limit the total WAL files to replay to under this threshold. see [Write Ahead Log](#).

## Reading footer and meta blocks of all the SST files

---

When `options.max_open_files` is set to `-1`, during DB open, all the SST files will be opened, with their footer and metadata blocks to be read. This is random reads from disk. If you have a lot of files and a relatively high latency device, especially spinning disks, those random reads can take a long time. Two options can help mitigate the problem:

- `options.max_file_opening_threads` allows reading those files in parallel. Making this number higher usually works well on high bandwidth devices, like SSD.
- Set `options.skip_stats_update_on_db_open=false`. This allows RocksDB to do one fewer read per file.
- Tune the LSM-tree to reduce the number of SST file is also helpful.

## Opening DB with

```
options.paranoid_checks == true &&
options.skip_checking_sst_file_sizes_on_db_open == false
```

By default, `options.paranoid_checks` is `true` and `options.skip_checking_sst_file_sizes_on_db_open` is `false`. With this configuration, RocksDB will query the underlying file system for the sizes of all live SST files during DB open, which can be slow if there are many SST files, especially when the SST files reside on remote storage. If this occurs, you can set `options.skip_checking_sst_file_sizes_on_db_open` to true and keep `paranoid_checks` true.

## Opening too many DBs one by one

Some users manage multiple DBs per service and open DBs one-by-one. If they have multi-core server, they can use a thread pool and open those DBs in parallel.

Many users implement a queue service using RocksDB. In these services, from each queue, new items are added with a higher sequence ID and removed from the smallest sequence ID. Users usually read from a queue in sequence ID increasing order.

## Key Encoding

You can simply encode it as `<queue_id, sequence_id>`, where `queue_id` is fixed length encoded and `sequence_id` is encoded as big endian.

While iterating keys, a user can create an iterator and seek to `<queue_id, target_sequence_id>` and iterate from there.

## Old Item Deletion Problem

Since the oldest items are deleted, there can be a large amount of “tombstones” in the beginning of each `queue_id`. As a result, the two queries might be exceptionally slow:

- `Seek(<queue_id, 0>)`
- While you are in the last sequence ID of a `queue_id` and try to call `Next()`

To mitigate a problem, you can remember the first and last sequence ID of each `queue_id`, and never iterate over the range.

As another way to solve the second problem, you can set an end key of your iterate when you iterate inside a `queue_id`, by letting `ReadOptions.iterate_upper_bound` point to `<queue_id, max_int>` `<queue_id + 1>`. We encourage you always set it no matter whether you see the slowness problem caused by deletions.

## Checking new sequence IDs of a `queue_id`

If a user finishes processing the last `sequence_id` of a `queue_id`, and keep polling new item to be created, just `Seek(<queue_id, last_processed_id>)` and call `Next()` and see whether the next key is still for the same `<queue_id>`. Make sure `ReadOptions.iterate_upper_bound` points to `<queue_id + 1>` to avoid slowness for the item deletion problem.

If you want to further optimize this use case, to avoid binary search of the whole LSM tree each time, consider using `TailingIterator` (or `ForwardIterator` called in some parts of the codes)

(<https://github.com/facebook/rocksdb/blob/master/include/rocksdb/options.h#L1235-L1241>).

## Reclaiming space of deleted items faster

The queue service is a good use case of `CompactOnDeletionCollector`, which prioritizes ranges with more deletes when scheduling compactions. Set `ImmutableCFOptions::table_properties_collector_factories` to the factory defined here:

[https://github.com/facebook/rocksdb/blob/master/include/rocksdb/utilities/table\\_properties\\_collectors.h#L23-L27](https://github.com/facebook/rocksdb/blob/master/include/rocksdb/utilities/table_properties_collectors.h#L23-L27)

The page lists major projects being actively developed, or has been planned for future development.

## RocksDB on Remote Storage

---

### API between RocksDB and underlying storage

We recently completed a major refactoring of the `rocksdb::Env` class by separating the storage related interfaces into a class of its own, called `rocksdb::FileSystem`. In the long-term, the storage interfaces in `Env` will be deprecated and the main purpose of `Env` will be to abstract core OS functionality that RocksDB needs. The relevant PRs are <https://github.com/facebook/rocksdb/pull/5761> and <https://github.com/facebook/rocksdb/pull/6552>.

Over time, we will implement new functionality enabled by this separation -

1. Richer error handling - A compliant `FileSystem` implementation can return information about an IO error, such as whether its transient/retryable, permanent data-loss, file scope or entire file system etc. in `IOStatus`, which will allow RocksDB to do more intelligent error handling.
2. Fail fast - For file systems that allow callers to provide a timeout for an IO, RocksDB can provide better SLAs for user reads by providing an option to specify a deadline, and failing a Get/MultiGet as soon as the deadline is exceeded. This is an ongoing project.

## User Defined Timestamps

---

<https://docs.google.com/document/d/1FcDj0M8-pJzCajCa9waQkox6DKJIWZAHm36buK4wfqs/edit#heading=h.uxub5284i1ti>

## BlobDB

---

`BlobDB` is RocksDB's implementation of key-value separation, originally inspired by the [WiscKey paper](#). Large values (blobs) are stored in separate blob files, and only references to them are stored in RocksDB's LSM tree. By separating value storage from the LSM tree, BlobDB provides an alternative way of reducing write amplification, instead of tuning compactions. BlobDB is used in production at Facebook.

## File Checksums

---

See <https://github.com/facebook/rocksdb/wiki/Full-File-Checksum#the-next-step>

## Per Key/Value Checksum

---

# Encryption at Rest

---

## MultiGet()

See [MultiGet Performance](#) for background. We have the following related projects in various stages of planning and implementation -

- Support partitioned filter and index - The first phase of MultiGet provided significant performance improvement for full filter block and index, through various techniques such as reusing blocks, reusing index iterators, prefetching CPU cachelines etc. We plan to extend these to partitioned filters and indexes.
- Parallelize file reads in a single level - Currently MultiGet can parallelize reads to the same SST file. We plan to enhance this by parallelizing reads across all files in a single LSM level, thus benefiting more workloads.
- Deadline/timeouts - Users will be able to specify a deadline for a MultiGet request, and RocksDB will abort the request if the deadline is exceeded.
- Limit cumulative value size - Users will be able to specify an upper limit on the total size of values read by MultiGet, in order to control memory overhead.

## Bloom Filter Improvements

---

First phase complete, including

- Fixed flaws in old Bloom implementation for Block-based table with a new implementation (incl [Issue 4120](#) and [5857](#)), enabled using `format_version=5`. More detail [on wiki](#) and [in the main PR](#).
- [Allow non-integer bits / key settings](#) for finer granularity in control
- [Expose details like LSM level to custom FilterPolicy](#), for experimentation

Planned:

- Minimize memory internal fragmentation on generated filters (<https://github.com/facebook/rocksdb/pull/6427>)
- Investigate use of different bits/key for different levels (as in [Monkey](#))
- Investigate use of alternative data structures, most likely based on perfect hashing static functions. See [Xor filter](#), modified with [“fuse graph” construction](#). Or even [sgauss](#). We don't expect much difference in query times, but the primary trade-off to be between construction time and memory footprint for a given false positive rate. It's likely that L0 will continue to construct Bloom filters (fast memtable flushes) while compaction will spend more time to generate more compact structures.
- Re-vamp how filters are configured (based on above developments), probably moving away from bits/key as a proxy for accuracy.

## Improving Testing

---

## Adaptive Compaction

---

## Improving RocksDB Backups

---

Including <https://github.com/facebook/rocksdb/issues/6521>

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

# Building on Windows

This is a simple explanation of how to build RocksDB in **Win10 + Visual Studio 2015** (Method 1) or using `vcppkg` with **Win10 + Visual Studio 2017/2019** (Method).

## Method 1 (Win10 + Visual Studio 2015)

This is a simple step-by-step explanation of how I was able to build RocksDB (or RocksJava) and all of the 3rd-party libraries on Microsoft Windows 10. The Windows build system was already in place, however it took some trial-and-error for me to be able to build the 3rd-party libraries and incorporate them into the build.

### Pre-requisites

1. Microsoft Visual Studio 2015 (Community) with “Desktop development with C++” installed
2. `CMake` - I used version 3.14.2 installed from the 64bit MSI installer
3. `Git` - I used the Windows Git Bash.
4. `Mercurial` - I used the 64bit MSI installer
5. `wget`

### Steps

Create a directory somewhere on your machine that will be used a container for both the RocksDB source code and that of its 3rd-party dependencies. On my machine I used `C:\Users\aretter\code` , from hereon in I will just refer to it as `%CODE_HOME%` ; which can be set as an environment variable, i.e. `SET CODE_HOME=C:\Users\aretter\code` .

All of the following is executed from the “**Developer Command Prompt for VS2015**”:

### Build GFlags

```
1. cd %CODE_HOME%
2. wget https://github.com/gflags/gflags/archive/v2.2.0.zip
3. unzip v2.2.0.zip
4. cd gflags-2.2.0
5. mkdir target
6. cd target
7. cmake -G "Visual Studio 14 Win64" ..
```

Open the project in Visual Studio, create a new x64 Platform by copying the Win32 platform and selecting x64 CPU. Close Visual Studio.

```
1. msbuild gflags.sln /p:Configuration=Debug /p:Platform=x64
2. msbuild gflags.sln /p:Configuration=Release /p:Platform=x64
```

The resultant static library can be found in `%CODE_HOME%\gflags-`

`2.2.0\target\lib\Debug\gflags_static.lib` or `%CODE_HOME%\gflags-2.2.0\target\lib\Release\gflags_static.lib`.

## Build Snappy

```
1. cd %CODE_HOME%
2. wget https://github.com/google/snappy/archive/1.1.7.zip
3. unzip 1.1.7.zip
4. cd snappy-1.1.7
5. mkdir build
6. cd build
7. cmake -DCMAKE_GENERATOR_PLATFORM=x64 ..
8. msbuild Snappy.sln /p:Configuration=Debug /p:Platform=x64
9. msbuild Snappy.sln /p:Configuration=Release /p:Platform=x64
```

The resultant static library can be found in `%CODE_HOME%\snappy-1.1.7\build\Debug\snappy.lib` or `%CODE_HOME%\snappy-1.1.7\build\Release\snappy.lib`.

## Build LZ4

```
1. cd %CODE_HOME%
2. wget https://github.com/lz4/lz4/archive/v1.9.2.zip
3. unzip v1.9.2.zip
4. cd lz4-1.9.2
5. cd visual\VS2010
6. devenv lz4.sln /upgrade
7. msbuild lz4.sln /p:Configuration=Debug /p:Platform=x64
8. msbuild lz4.sln /p:Configuration=Release /p:Platform=x64
```

The resultant static library can be found in `%CODE_HOME%\lz4-`

`1.9.2\visual\VS2010\bin\x64_Debug\liblz4_static.lib` or `%CODE_HOME%\lz4-`

`1.9.2\visual\VS2010\bin\x64_Release\liblz4_static.lib`.

## Build ZLib

```
1. cd %CODE_HOME%
2. wget http://zlib.net/zlib1211.zip
3. unzip zlib1211.zip
4. cd zlib-1.2.11\contrib\vstudio\vc14
```

Edit the file `zlibvc.vcxproj`, changing `<command>cd ..\..\contrib\masm\masmx64 bld_ml64.bat</command>` to `<command>cd ..\..\masm\masmx64 bld_ml64.bat</command>`. Add a new line after `masmx64`.

```

1. "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64_x86\vcvarsamd64_x86.bat"
2. msbuild zlibvc.sln /p:Configuration=Debug /p:Platform=x64
3. msbuild zlibvc.sln /p:Configuration=Release /p:Platform=x64
4. copy x64\ZlibDllDebug\zlibwapi.lib x64\ZlibStatDebug\
5. copy x64\ZlibDllRelease\zlibwapi.lib x64\ZlibStatRelease\

```

The resultant static library can be found in `%CODE_HOME%\zlib-`  
`1.2.11\contrib\vstudio\vc14\x64\ZlibStatDebug\zlibstat.lib` or `%CODE_HOME%\zlib-`  
`1.2.11\contrib\vstudio\vc14\x64\ZlibStatRelease\zlibstat.lib`.

## Build ZStd

```

1. wget https://github.com/facebook/zstd/archive/v1.4.4.zip
2. unzip v1.4.4.zip
3. cd zstd-1.4.4\build\VS2010
4. devenv zstd.sln /upgrade
5. msbuild zstd.sln /p:Configuration=Debug /p:Platform=x64
6. msbuild zstd.sln /p:Configuration=Release /p:Platform=x64

```

The resultant static library can be found in `%CODE_HOME%\zstd-`  
`1.4.4\build\VS2010\bin\x64_Debug\libzstd_static.lib` or `%CODE_HOME%\zstd-`  
`1.4.4\build\VS2010\bin\x64_Release\libzstd_static.lib`.

## Build RocksDB

```

1. cd %CODE_HOME%
2. git clone https://github.com/facebook/rocksdb.git
3. cd rocksdb

```

Edit the file `%CODE_HOME%\rocksdb\thirdparty.inc` to have these changes:

```

1. set(GFLAGS_HOME $ENV{THIRDPARTY_HOME}/gflags-2.2.0)
2. set(GFLAGS_INCLUDE ${GFLAGS_HOME}/target/include)
3. set(GFLAGS_LIB_DEBUG ${GFLAGS_HOME}/target/lib/Debug/gflags_static.lib)
4. set(GFLAGS_LIB_RELEASE ${GFLAGS_HOME}/target/lib/Release/gflags_static.lib)
5.
6. set(SNAPPY_HOME $ENV{THIRDPARTY_HOME}/snappy-1.1.7)
7. set(SNAPPY_INCLUDE ${SNAPPY_HOME} ${SNAPPY_HOME}/build)
8. set(SNAPPY_LIB_DEBUG ${SNAPPY_HOME}/build/Debug/snappy.lib)
9. set(SNAPPY_LIB_RELEASE ${SNAPPY_HOME}/build/Release/snappy.lib)
10.
11. set(LZ4_HOME $ENV{THIRDPARTY_HOME}/lz4-1.9.2)
12. set(LZ4_INCLUDE ${LZ4_HOME}/lib)
13. set(LZ4_LIB_DEBUG ${LZ4_HOME}/visual/VS2010/bin/x64_Debug/liblz4_static.lib)
14. set(LZ4_LIB_RELEASE ${LZ4_HOME}/visual/VS2010/bin/x64_Release/liblz4_static.lib)
15.
16. set(ZLIB_HOME $ENV{THIRDPARTY_HOME}/zlib-1.2.11)
17. set(ZLIB_INCLUDE ${ZLIB_HOME})

```

```

18. set(ZLIB_LIB_DEBUG ${ZLIB_HOME}/contrib/vstudio/vc14/x64/zlibStatDebug/zlibstat.lib)
19. set(ZLIB_LIB_RELEASE ${ZLIB_HOME}/contrib/vstudio/vc14/x64/zlibStatRelease/zlibstat.lib)
20.
21. set(ZSTD_HOME ${ENV{THIRDPARTY_HOME}}/zstd-1.4.4)
22. set(ZSTD_INCLUDE ${ZSTD_HOME}/lib ${ZSTD_HOME}/lib/dictBuilder)
23. set(ZSTD_LIB_DEBUG ${ZSTD_HOME}/build/VS2010/bin/x64_Debug/libzstd_static.lib)
24. set(ZSTD_LIB_RELEASE ${ZSTD_HOME}/build/VS2010/bin/x64_Release/libzstd_static.lib)

```

And then finally to compile RocksDB:

- **NOTE:** The default CMake build will generate MSBuild project files which include the `/arch:AVX2` flag. If you have this CPU extension instruction set, then the generated binaries will also only work on other CPU's with AVX2. If you want to create a build which has no specific CPU extensions, then you should also pass the `-DPORTABLE=1` flag in the `cmake` arguments below.
- **NOTE:** The build options below include `-DXPRESS=1` which enables Microsoft XPRESS compression. This requires Windows 10 or newer to work reliably and is not backwards compatible with older versions of Windows. At present we build RocksJava releases without XPRESS.

```

1. mkdir build
2. cd build
3. set JAVA_HOME="C:\Program Files\Java\jdk1.7.0_80"
4. set THIRDPARTY_HOME=C:/Users/areetter/code
5. cmake -G "Visual Studio 14 Win64" -DJNI=1 -DGFLAGS=1 -DSNAPPY=1 -DLZ4=1 -DZLIB=1 -DZSTD=1 -DXPRESS=1 ..
6. msbuild rocksdb.sln /p:Configuration=Release

```

## Method 2 (Win10 + Visual Studio 2017/2019)

This is a very simple step-by-step explanation of how I was able to build RocksDB on Microsoft Windows 10. It should be very easy for users who uses vcpkg to install RocksDB. However, vcpkg build RocksDB as a shared library by default. There are two things we need to do since we have installed vcpkg already.

### Step 1

- `cd %home%\vcpkg\ports\rocksdb`, `%home%` is the path where you installed your vcpkg
- `set(VCPKG_LIBRARY_LINKAGE static)` to the top of portfile.cmake and then running `vcpkg install rocksdb:x64-windows`. If you have installed rocksdb as a static library, run `vcpkg remove rocksdb:x64-windows` before install command. The resultant static library can be found in `%home%\vcpkg\packages\rocksdb_x64-windows\lib\rocksdb.lib` rather than `%home%\vcpkg\packages\rocksdb_x64-windows\lib\rocksdb-shared.lib`.

### Step 2

- included Shlwapi.lib and Rpcrt4.lib as the input of your project linker manually.

- Pluggable WAL
- Data encryption
- Warm block cache after flush and compactions in a smart way
- Queryable Backup
- Improve sub-compaction by making data partition more evenly
- Tools to collect operations to a database and replay them
- Customized bloom filter for data blocks
- Build a new compaction style optimized for time series data
- Implement YCSB benchmark scenarios in db\_bench
- Improve DB recovery speed when WAL files are large (parallel replay WAL)
- use thread pools to do readahead + decompress and compress + write-behind. Igor started on this. When manual compaction is multi-threaded then we can use RocksDB as a fast external sorter – load keys in random order with compaction disabled, then do manual compaction.
- expose merge operator in MongoDB + RocksDB
- SQLite + RocksDB
- Snappy compression for WAL writes. Maybe this is only done for large writes and maybe we add a field to WriteOptions so a user can request it.

## Research Projects

---

### Adaptive write-optimized algorithms

---

The granularity of tuning in RocksDB is a column family (CF), which is essentially a separate LSM tree. However many times the data in a column family is composed of various workloads with diverse characteristic, each requiring a different tuning. In the case of MyRocks this could be because CF stores multiple indexes – some are heavy on point operations, others heavy on range operations, some are read heavy, others are read-only, others are read+write. Even within an index there is variety – there will be hot spots that get writes, some data will be write once, some write N times and then it becomes read-only.

Tuning workloads and split indexes into different column families, would take too much resources. When there is variety within one index there is not much that can be done currently. A clever algorithm should be able to handle such cases.

### Self-tuning DBs

---

Each LSM tree has many configuration parameters, which makes optimizing them very difficult even for experts. The typical black-box machine learning algorithms that rely on availability on many data points would not work here since generating each new data point would require running a DB at large scale, which requires non-trivial time and resource budget. Gathering data points from existing production systems is also not practical due to security concerns of sharing such data to outside the

organization.

## Sub-optimal, non-tunable LSM

---

There is a long tail of users with small workloads, who are ok with sub-optimal performance but do not have the engineering resources to fine tune them. A list of default value for configuration also do not always work for them since it could give unacceptably bad performance for some particular workloads. An LSM tree that does not have any tuning knobs yet provides a reasonable performance for any given workload (even the corner cases) is much practical for the silent, long-tail of users.

## Bloom filter for range queries

---

LSM trees have the drawback of bad read amplification. To mitigate this problem bloom filters are essential to LSM trees. A bloom filter can tell with a good probability whether a key exist in the SST file. When servicing a SQL query however many queries are not just point lookups and involve a range specifier on the key. An example is "Select \* from T where  $10 < c1 < 20$ " when the specific key (composed of  $c1$ ) is not available. A bloom filter that could operate on ranges could be helpful in such cases.

## RocksDB on pure-NVM

---

How to optimize RocksDB when it uses non-volatile memory (NVM) as the storage device?

## RocksDB on hierarchical storage

---

RocksDB is usually run on two-level storage setup: RAM + SSD or RAM + HDD, where RAM is volatile and SSD, and HDD are non-volatile storage. There are interesting design choices of how to split data between the two storage. For example, RAM could be used solely as a cache for blocks, or it could preload all the indexes and filters, or given partitioned index/filters only preload the top-level to the RAM. The design choice become much more interesting when we have a multi-layer hierarchy of storage devices: RAM + NVM + SSD + HDD (or any combination of them) each with different storage characteristics. What is the optimal way to split data among the hierarchy.

## Time series DBs

---

How we can optimize RocksDB based on the particular characteristics of time series databases? For example can we do better encoding of keys knowing that they are integers in ascending order? Can we do better encoding of values knowing that they are floating point numbers with high correlation between adjacent numbers? What is an optimal compaction strategy given the patterns of data distribution in a time series DB? Do such data show different characteristics in different levels of the LSM tree

and we can we leverage such information for more efficient data layout for each level?  
etc.

## 2019.12 6th Annual RocksDB Summit at Facebook

## 2017.12 10th RocksDB Meetup

- [Kickoff: A quick overview of RocksDB new features](#) (by Sagar Vemuri)
- [Cassandra on RocksDB](#) (by Pengchao Wang)
- [MySQL and RocksDB](#) (by Herman Lee)
- [MyRocks In Production](#) (by Domas Mituzas)
- [State Management in Kafka Streams using RocksDB](#) (by Guozhang Wang)
- [Are you a Tortoise or a Hare](#) (by Matthew Von-Maszewski)

## 2017.10 HPTS

- [Workload Diversity with RocksDB](#) (Siying Dong)

## 2016.12 8th RocksDB Meetup [playlist](#)

- [Lightning Talks](#) (Facebook Engineers)
  - Yueh-Hsuan Chiang – Lua Compaction Filters
  - Karthikeyan Radhakrishnan – Read Cache
  - Andrew Kryczka – Improved Statistics/Delete Range
  - Yi Wu – Iterator Assisted Inserts
  - Anirban Rahut – Large Value support
  - Aaron Gao – New iterator features
  - Islam AbdelRahman – Bulk loading
  - Laurent Demainly – RocksDB cloning at the speed of light with WDT
  - Arun Sharma – Iterlib: A zero copy iterator library
- [RocksDB Future](#) (Siying Dong @ Faceboook)
- [The updated state of MyRocks, MongoRocks and RocksDB](#) (Mark Callaghan @ Facebook)
- [Talk from Percona](#) (George Lorch @ Percona)
- [Rocksplicator](#) (Bo Liu @ Pinterest)
- [RocksOS: Plug-in RocksDB Into ObjectStore](#) (Praveen Rao @ Microsoft)

## 2016.06 6th RocksDB Meetup [video](#)

- [Improving RocksDB's Write Scalability](#) (Nathan Bronson @ Facebook)
- [Counting with Domain Specific Databases](#) (Yunjing Xu @ Smyte)

## 2016.04 Percona Live

- [MyRocks, MongoRocks and RocksDB](#) (Mark Callaghan @ Facebook)
- [RocksDB: Key-Value Store Optimized for Flash-Based SSD](#) (Siying Dong @ Facebook)
- [Running MongoRocks in production](#) (Igor Canadi @ Facebook)

## 2016.02 The Hive Think Tank “Rocking the Database World with RocksDB” [Video] ([https://www.youtube.com/watch?v=g2PglBY\\_18w](https://www.youtube.com/watch?v=g2PglBY_18w))

- Open Discussion on RocksDB Roadmap and Vision (Dhruba Borthakur @ Facebook)
- Ceph + RocksDB (Sage Weil)
- MongoDB + RocksDB (Igor Canadi @ Facebook)
- Use of RocksDB at Rakuten (Qian Zhu @ Rakuten)
- MySQL + RocksDB (Siying Dong @ Facebook)

## 2015.12 5th RocksDB Meetup [Video Playlist] (<https://www.youtube.com/playlist?list=PLb0IAmt7-GS292rNhzcGTi5c0dcTuK70v>)

- RocksDB Transactions (Anthony Giardullo @ Facebook)
- RealPin: A Highly Customizable Object Retrieval System (Bo Liu @ Pinterest)
- RocksDB on Open-Channel SSDs (Javier González @ CNEX Labs)
- RocksDB for Personalized Search At Airbnb (Tao Xu @ Airbnb)
- CockroachDB’s MVCC model (Spencer Kimball @ CockroachDB)
- Fine-tuning RocksDB (Praveen Krishnamoorthy @ Samsung Inc)

## 2015.10.22 The Databaseology Lectures @ CMU

- The Journey From Faster To Better (Mark Callaghan & Igor Canadi @ Facebook)

## 2015.09.29 HPTS

- RocksDB: Challenges of LSM-Trees in Practice (Siying Dong @ Facebook)

## 2015.09.23 Percona Live Europe

- RocksDB storage engine for MySQL and MongoDB (Igor Canadi @ Facebook)

## 2015.07.02 4th RocksDB Meetup

- Transactions on RocksDB (Anthony Giardullo @ Facebook)

## 2015.05.20 XLDDB

- Lightning Talk “MySQL + RocksDB for better storage efficiency than InnoDB” (Siying Dong @ Facebook)

## 2014.12 2nd RocksDB Meetup

- [Talks from Facebook RocksDB Team](#) (Igor Canadi, Yueh-Hsuan Chiang and Siying Dong @ Facebook)
- [Building queues that are Rocks solid](#) (Reed Allman @ Iron.io)
- [RocksDB usage at LinkedIn](#) (Ankit Gupta and Naveen Somasundaram @ LinkedIn)

## 2014.08.05 Flash Memory Summit

- [RocksDB](#) (Siying Dong @ Facebook)

## 2014.03.27 1st RocksDB Meetup

- [Supporting a 1PB In-Memory Workload](#) (Haobo Xu @ Facebook)
- [Column Families in RocksDB](#) (Igor Canadi @ Facebook)
- [“Lockless” Get\(\) in RocksDB](#) (Lei Jin @ Facebook)
- [Prefix Hashing in RocksDB](#) (Siying Dong @ Facebook)

## 2013.11.14 Open Source RocksDB

- [The History of RocksDB](#) (Dhruba Borthakur @ Facebook)

**Optimizing Space Amplification in RocksDB**, CIDR 2017, Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, Michael Stumm [link](#)

**Reducing DRAM footprint with NVM in Facebook**, Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. In Proceedings of the Thirteenth EuroSys Conference (EuroSys '18). Association for Computing Machinery, New York, NY, USA, Article 42, 1-13. [link](#)

**Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy**, Amy Tai, Andrew Kryczka, Shobhit O. Kanaujia, Kyle Jamieson, Michael J. Freedman, and Asaf Cidon. 2019. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 977-991, [link](#).

**Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook**, Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du, 18th USENIX Conference on File and Storage Technologies (FAST20), USA, 209–223. [link](#).

# RocksDB Features that are not in LevelDB

---

We stopped maintaining this page since 2016. New features are not added to the lists.

## Performance

---

- Multithread compaction
- Multithread memtable inserts
- Reduced DB mutex holding
- Optimized level-based compaction style and universal compaction style
- Prefix bloom filter
- Memtable bloom filter
- Single bloom filter covering the whole SST file
- Write lock optimization
- Improved Iter::Prev() performance
- Fewer comparator calls during SkipList searches
- Allocate memtable memory using huge page.

## Features

---

- Column Families
- Transactions and WriteBatchWithIndex
- Backup and Checkpoints
- Merge Operators
- Compaction Filters
- RocksDB Java
- Manual Compactions Run in Parallel with Automatic Compactions
- Persistent Cache
- Bulk loading
- Forward Iterators/ Tailing iterator
- Single delete
- Delete files in range
- Pin iterator key/value

## Alternative Data Structures And Formats

---

- Plain Table format for memory-only use cases
- Vector-based and hash-based memtable format
- Clock-based cache (coming soon)
- Pluggable information log
- Annotate transaction log write with blob (for replication)

## Tunability

---

- Rate limiting
- Tunable Slowdown and Stop threshold
- Option to keep all files open
- Option to keep all index and bloom filter blocks in block cache
- Multiple WAL recovery modes
- Fadvise hints for readahead and to avoid caching in OS page cache
- Option to pin indexes and bloom filters of L0 files in memory
- More Compression Types: zlib, lz4, zstd
- Compression Dictionary
- Checksum Type: xxhash
- Different level size multiplier and compression type for each level.

## Manageability

---

- Statistics
- Thread-local profiling
- More commands in command-line tools
- User-defined table properties
- Event listeners
- More DB Properties
- Dynamic option changes
- Get options from a string or map
- Persistent options to option files

Performance is a strength of RocksDB. You are welcome to ask the questions if you see RocksDB is unexpectedly slow, or wonder whether there is any room of significant improvements. When you ask a performance-related question, providing more information will increase your chance of getting an answer sooner. In many cases, people can't tell much from a simple description of symptom. Answers to following questions are usually helpful:

1. Is your problem for Get(), iterator, or write?
2. either the answer of 1 is read or write, which metric(s) is unexpected to you?  
And what the value of the unexpected metric(s).
  - i. throughput. For this problem, how many threads are you using?
  - ii. average latency.
  - iii. latency outlier.
3. What's your storage media?
  - i. SSD
  - ii. single hard drive
  - iii. hard drive array
  - iv. ramfs/tmpfs
  - v. HDFS or other remote storage?

More information about your configuration and DB state will be helpful:

- Set-up: do you use many RocksDB instances in single service, or one single one? How large is your expected DB size? How many column families do you use per DB instance?
- RocksDB release you are using.
- Build flags. If you are using `make`, the best and easiest way to provide the information is to share `make_config.mk` after running `make`. If you cannot provide full information, there are several information related to performance:
  - what's the platform? Linux, Windows, OS X, etc.
  - which allocator are you using? `jemalloc`, `tcmalloc`, `glibc malloc`, or others?  
If it is `jemalloc`, `make_config.mk` should show `JEMALLOC=1`. If it is `tcmalloc`, you can find it "`-ltcmalloc`" in `PLATFORM_LDFLAGS`.
  - is calculating CRC using SSE instruction supported and turned on? This information will be printed out in the header of the info log file, like this  
"Fast CRC32 supported: 1"
  - is `fallocate` supported and turned on?
- RocksDB Options you are using. You can provide your option file. Your option file is under your DB directory, naming as `OPTIONS-xxxxx`. Or you can also copy the header part of your information log file, which can be found under your DB directory, usually named as `LOG`, and `LOG.old.xxxxx`. The options files usually have RocksDB options printed out in the header part. If you cannot provide answer to the two, tell us some of your critical options will also be helpful (if you never set it, you can tell us default):
  - write buffer size
  - `level0_file_num_compaction_trigger`
  - `target_file_size_base`

- compression
  - compaction style
  - If leveled compaction:
    - max\_bytes\_for\_level\_base
    - max\_bytes\_for\_level\_multiplier
    - level\_compaction\_dynamic\_level\_bytes
  - If universal compaction:
    - size\_ratio
    - max\_size\_amplification\_percent
  - block cache size
  - Bloom filter setting
  - Any uncommon options you set
- LSM-tree structure. You can generate a report of LSM-tree summary by calling DB::GetProperty() with property “rocksdb.stats”. Another way to find the structure of LSM-tree is to use “ldb –manifest\_dump” on your manifest file, which is MANIFEST-xxxxxx file under the directory of your DB.
  - Disk I/O stats while the problem happens. You can use the command you like. The command I usually use is “iostat -kxt 1”.
  - Your workload characteristics:
    - key and value size
    - whether read and write are spiky
    - whether and how do you delete the data?
  - If possible, share your information log files. By default, they are under your DB directory, named LOG and LOG.old.xxxxx.
  - Hardware setting. We know there are cases where the concrete hardware setting can't be shared, but even if you can tell us the memory and number of cores are in which range, or order of magnitude, that will be helpful.
  - If there is a way to easily reproduce the performance problem, reproduce instruction will be helpful.

# Articles about RocksDB

---

Herein we try and maintain a list of relevant 3rd-party articles about RocksDB.

## Getting Started

---

[Migrating from LevelDB to RocksDB](#) by Leonidas Galanis. 16th January 2015.

[The Journey from Faster to Better](#) by Mark Callaghan and Igor Canadi. 22nd October 2015.

## Working with RocksDB

---

[Integrating RocksDB with MongoDB](#) by Igor Canadi. 22nd April 2015.

[WriteBatchWithIndex: Utility for Implementing Read-Your-Own-Writes](#) by Siying Dong. 27th February 2015.

[How to persist in-memory RocksDB database?](#) by Igor Canadi. 27th March 2014.

[Reading RocksDB options from a file](#) by Leonidas Galanis. 24th February 2015.

[How to backup RocksDB?](#) by Igor Canadi. 27th March 2014.

## Internals

---

[State of RocksDB](#) by Igor Canadi. 4th December 2014.

[Geo-spatial Features in RocksDB](#) by Igor Canadi and Yin Wang. 12th November 2014.

[“Lockless” Get\(\) in RocksDB? Scale with CPU Cores](#) by Lei Jin. 27th March 2014.

[Under the Hood: Building and open-sourcing RocksDB](#) by Dhruba Borthakur. 21st November 2013.

## MyRocks

---

[MyRocks Deep Dive](#) by Yoshinori Matsunobu. 18th April 2016.

## Showcases

---

[how innodb lost its advantage](#) by Domas Mituzas. 9th April 2015.

[How RocksDB is used in osquery](#) by Mike Arpaia and Ted Reed. 30th April 2015.

[MongoDB + RocksDB at Parse](#) by Charity Majors. 22nd April 2015.

[Stream Processing with Samza @ LinkedIn](#) by Naveen Somasundaram. 4th December 2014.

[Rock Solid Queues @ Iron.io](#) by Reed Allman. 4th December 2014.

## Benchmarks

---

[RocksDB & ForestDB via the ForestDB benchmark, part 1](#) by Mark Callaghan. 8th June 2015. See also: [http://smalldatum.blogspot.com/2015/06/rocksdb-forestdb-via-forestdb-benchmark\\_8.html](http://smalldatum.blogspot.com/2015/06/rocksdb-forestdb-via-forestdb-benchmark_8.html) [http://smalldatum.blogspot.com/2015/06/rocksdb-forestdb-via-forestdb-benchmark\\_73.html](http://smalldatum.blogspot.com/2015/06/rocksdb-forestdb-via-forestdb-benchmark_73.html) [http://smalldatum.blogspot.com/2015/06/rocksdb-forestdb-via-forestdb-benchmark\\_9.html](http://smalldatum.blogspot.com/2015/06/rocksdb-forestdb-via-forestdb-benchmark_9.html)

[Comparing LevelDB and RocksDB, take 2](#) by Mark Callaghan. April 27th 2015.

[Benchmarking the leveldb family](#) by Mark Callaghan. July 7th 2014.