RocksDB Overview

Jump to bottom

Itamasi edited this page on Jul 18, 2023 · 10 revisions

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:

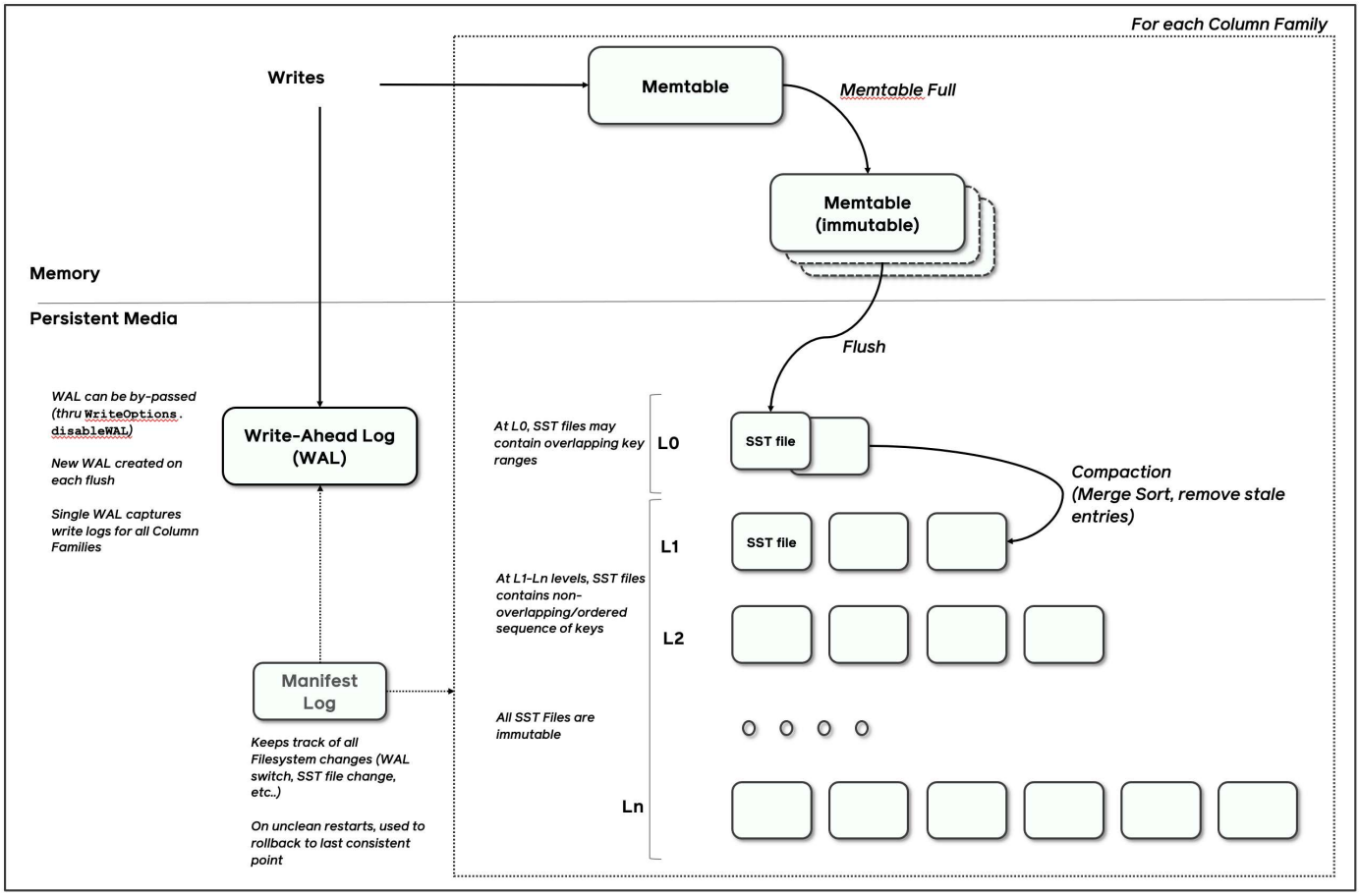
Newer versions of this software should be backward compatible, so that existing applications do not need to change when upgrading to newer releases of RocksDB. Unless using newly provided features, existing applications also should be able to revert to a recent old release. See RocksDB Compatibility 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** (aka. Write Ahead Log(WAL)). 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 default format of 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. `DeleteRange` API 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 range scan 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 range scan 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 `Options.prefix_extractor` to enable a key-prefix based filtering. When `Options.prefix_extractor` is set, a hash of the prefix is also added to the Bloom. An `Iterator` that specifies a key-prefix (in `ReadOptions`) will use the Bloom Filter to avoid looking into data files that do not contain keys with the specified key-prefix. See Prefix-Seek.

Persistence

RocksDB has a Write Ahead Log (WAL). All write operations (`Put`, `Delete` and `Merge`) 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 `fsync` 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 `fsync` 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 also maintains a Full File Checksum (see Full File Checksum and Checksum Handoff) and optionally per key-value checksum.

RocksDB dynamically detects and utilizes CPU checksum offload support.

Multi-Threaded Compactions

In the presence of ongoing writes, compactions are needed for space efficiency, read (query) efficiency, and timely data deletion. Compaction removes key-value bindings that have been deleted or overwritten, and re-organizes data for query efficiency. Compactions may occur in multiple threads if configured.

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. 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.

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 log file is used to record all the database state changes. 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.

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 compactions, 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(aka. Info 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 (see `DBOptions.info_log_level`). This LOG may be configured to roll at a specified periodicity. The logging interface is pluggable. Users can plug in a different logger. See 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 provides a backup API, `BackupEngine`. 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 or rate limiter.

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 redundant caching of the same data in OS page cache.

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, enable direct I/O, etc... See IO for more details.

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 skiplist. The skiplist 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 skiplist 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. See Stress-test. You can also add fuzzers to this directory by leveraging the fuzzing infrastructure there, the fuzzers are continuously run on Google's OSS-Fuzz, errors will be reported to RocksDB developers.

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.

- Publication
- Features Not in LevelDB
- How to ask a performance-related question?
- Articles about Rocks

Clone this wiki locally

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