

How RocksDB works

April 19, 2023 dev [\(/tag/dev/\)](#) databases [\(/tag/databases/\)](#)
key-value [\(/tag/key-value/\)](#) lsm [\(/tag/lsm/\)](#) rocksdb [\(/tag/rocksdb/\)](#)

Introduction

Over the past years, the adoption of RocksDB increased dramatically. It became a standard for embeddable key-value stores.

Today RocksDB runs in production at Meta, [Microsoft](#) (<https://blogs.bing.com/Engineering-Blog/october-2021/RocksDB-in-Microsoft-Bing/>), [Netflix](#) (<https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef>), [Uber](#) (<https://eng.uber.com/cherami-message-queue-system/>). At [Meta](#) (<https://engineering.fb.com/2021/07/22/data-infrastructure/mysql/>) RocksDB serves as a storage engine for the MySQL deployment powering the distributed graph database called TAO.

Big tech companies are not the only RocksDB users. Several startups were built around RocksDB - [CockroachDB](#) (<https://www.cockroachlabs.com/>), [Yugabyte](#) (<https://www.yugabyte.com/>), [PingCAP](#) (<https://www.pingcap.com/>), [Rockset](#) (<https://rockset.com/>).

I spent the past 4 years at Datadog building and running services on top of RocksDB in production. In this post, I'll give a high-level overview of how RocksDB works.

What is RocksDB

RocksDB is an embeddable persistent key-value store. It's a type of database designed to store large amounts of unique keys associated with values. The simple key-value data model can be used to build search indexes, document-oriented databases, SQL databases, caching systems and message brokers.

RocksDB was forked off Google's [LevelDB](#) (<https://github.com/google/leveldb>) in 2012 and optimized to run on servers with SSD drives. Currently, RocksDB is [developed](#) (<https://github.com/facebook/rocksdb>) and maintained by Meta.

RocksDB is written in C++, so additionally to C and C++, the C bindings allow embedding the library into applications written in other languages such as Rust (<https://github.com/rust-rocksdb/rust-rocksdb>), Go (<https://github.com/linxGnu/grocksdb>) or Java (<https://github.com/facebook/rocksdb/tree/main/java>).

If you ever used SQLite, then you already know what an embeddable database is! In the context of databases, and particularly in the context of RocksDB, "embeddable" means:

- The database doesn't have a standalone process, it's instead linked with your application.
- It doesn't come with a built-in server that can be accessed over the network.
- It is not distributed, meaning it does not provide fault tolerance, replication, or sharding mechanisms.

It is up to the application to implement these features if necessary.

RocksDB stores data as a collection of key-value pairs. Both keys and values are not typed, they are just arbitrary byte arrays. The database provides a low-level interface with a few functions for modifying the state of the collection:

- `put(key, value)` : stores a new key-value pair or updates an existing one
- `merge(key, value)` : combines the new value with the existing value for a given key
- `delete(key)` : removes a key-value pair from the collection

Values can be retrieved with point lookups:

- `get(key)`

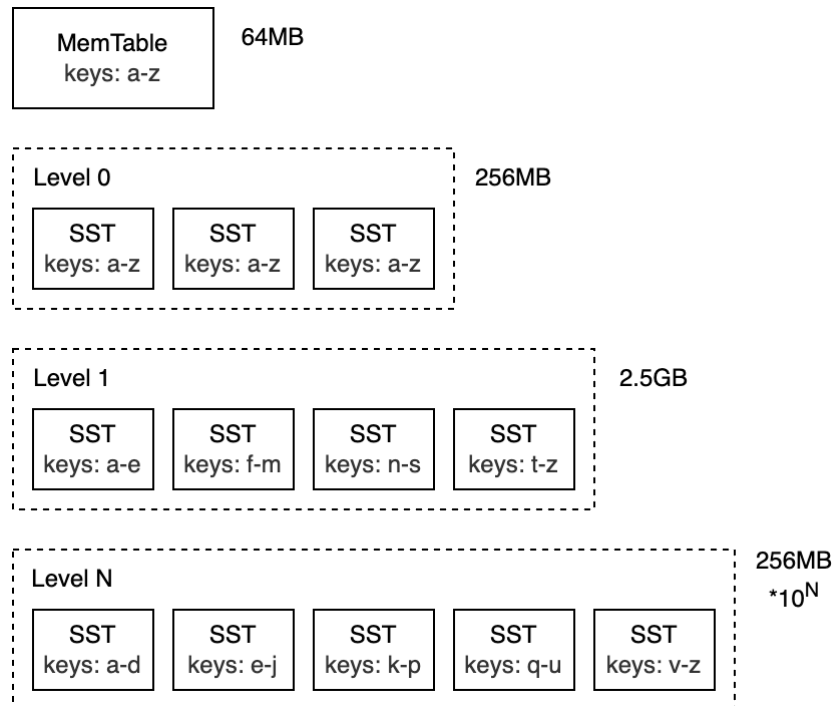
An iterator enables "range scans" - seeking to a specific key and accessing subsequent key-value pairs in order:

- `iterator.seek(key_prefix); iterator.value(); iterator.next()`

Log-structured merge-tree

The core data structure behind RocksDB is called the *Log-structured merge-tree* (LSM-Tree). It's a tree-like structure organized into multiple levels, with data on each level ordered by key. The LSM-tree was primarily designed for write-heavy workloads and was introduced in 1996 in a paper (<http://paperhub.s3.amazonaws.com/18e91eb4db2114a06ea614f0384f2784.pdf>) under the same name.

The top level of the LSM-Tree is kept in memory and contains the most recently inserted data. The lower levels are stored on disk and are numbered from 0 to N. Level 0 (L0) stores data moved from memory to disk, Level 1 and below store older data. When a level becomes too large, it's merged with the next level, which is typically an order of magnitude larger than the previous one.



Note: I'll be talking specifically about RocksDB, but most of the concepts covered apply to many databases that uses LSM-trees under the hood (e.g. Bigtable, HBase, Cassandra, ScyllaDB, LevelDB, MongoDB WiredTiger).

To better understand how LSM-trees work, let's take a closer look at the write and read paths.

Write path

MemTable

The top level of the LSM-tree is known as the *MemTable*. It's an in-memory buffer that holds keys and values before they are written to disk. All inserts and updates always go through the memtable. This is also true for deletes - rather than modifying key-value pairs in-place, RocksDB marks deleted keys by inserting a tombstone record.

The memtable is configured to have a specific size in bytes. When the memtable becomes full, it is swapped with a new memtable, the old memtable becomes immutable.

Note: The default size of the memtable is 64MB.

Let's start by adding a few keys to the database:

```
db.put("chipmunk", "1")
db.put("cat", "2")
db.put("raccoon", "3")
db.put("dog", "4")
```

MemTable

Type	Key	Value
PUT	cat	2
PUT	chipmunk	1
PUT	dog	4
PUT	raccoon	3

As you can see, the key-value pairs in the memtable are ordered by their key. Although *chipmunk* was inserted first, it comes after *cat* in the memtable due to the sorted order. The ordering is a requirement for supporting range scans and it makes some operations, which I will cover later more efficient.

Write-ahead log

In the event of a process crash or a planned application restart, data stored in the process memory is lost. To prevent data loss and ensure that database updates are durable, RocksDB writes all updates to the *Write-ahead log* (WAL) on disk, in addition to the memtable. This way the database can replay the log and restore the original state of the memtable on startup.

The WAL is an append-only file, consisting of a sequence of records. Each record contains a checksum, a key-value pair, and a record type (Put/Merge/Delete). Unlike in the memtable, records in the WAL are not ordered by key. Instead, they are appended in the order in which they arrive.

WAL

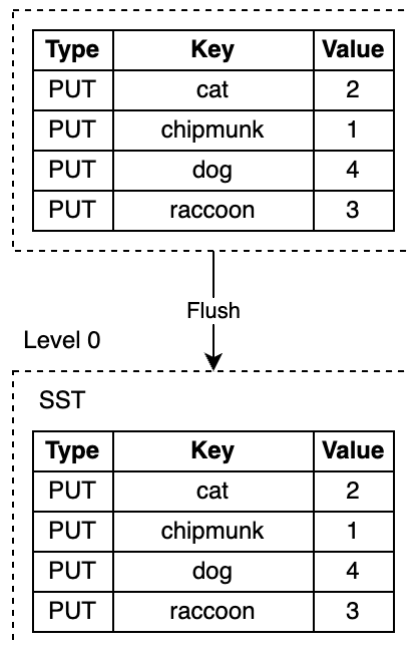
Type	Checksum	Key	Value
PUT	7301	chipmunk	1
PUT	4197	cat	2
PUT	3824	raccoon	3
PUT	1832	dog	4

Flush

RocksDB runs a dedicated background thread that persists immutable memtables to disk. As soon as the flush process is complete, the immutable memtable and the corresponding WAL are discarded. RocksDB starts writing to a new WAL and a new memtable. Each flush produces a single SST file on L0. The produced files are immutable - they are never modified once written to disk.

The default memtable implementation in RocksDB is based on a [skip list](https://en.wikipedia.org/wiki/Skip_list) (https://en.wikipedia.org/wiki/Skip_list). The data structure is a linked list with additional layers of links that allow fast search and insertion in sorted order. The ordering makes the flush process efficient, allowing the memtable content to be written to disk sequentially by iterating the key-value pairs. Turning random inserts into sequential writes is one of the key ideas behind the LSM-tree design.

Immutable MemTable



Note: RocksDB is highly configurable. Like many other components, the memtable implementation can be swapped with an alternative. It's not uncommon to see self-balancing binary search trees used to implement memtables in other LSM-based databases.

SST

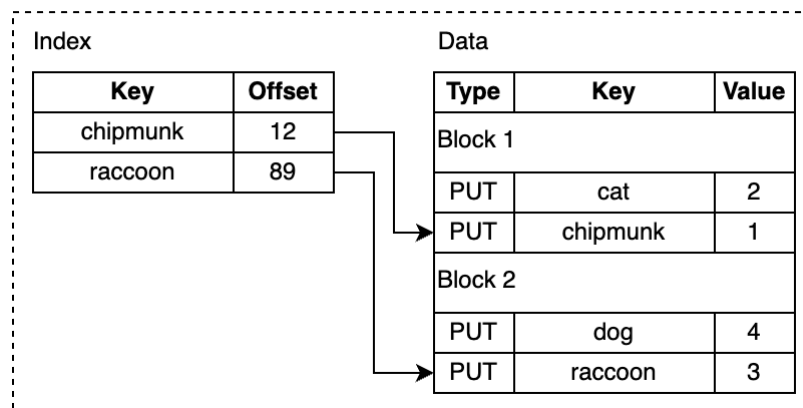
SST stands for Static Sorted Table (or Sorted String Table in some other databases). This is a block-based file format that organizes data into fixed-size blocks (4KB by default). Individual blocks can be compressed with various compression algorithms supported by RocksDB, such as Zlib, BZ2, Snappy, LZ4, or ZSTD. Similar to records in the WAL, blocks contain checksums to detect data corruptions. RocksDB verifies these checksums every time it reads from the disk.

Blocks in an SST file are divided into sections. The first section, the *data* section, contains an ordered sequence of key-value pairs. This ordering allows delta-encoding of keys, meaning that instead of storing full keys, we can store only the difference between adjacent keys.

To find a specific key, we could use binary search on the SST file blocks. RocksDB optimizes lookups by adding an index, which is stored in a separate section right after the data section. The index maps the last key in each data block to its

corresponding offset on disk. Again, the keys in the index are ordered, allowing us to find a key quickly by performing a binary search. For example, if we are searching for *lynx*, the index tells us the key might be in the block 2 because *lynx* comes after *chipmunk*, but before *raccoon*.

SST



In reality, there is no *lynx* in the SST file above, but we had to read the block from disk and search it. RocksDB supports enabling a [bloom filter](https://en.wikipedia.org/wiki/Bloom_filter) (https://en.wikipedia.org/wiki/Bloom_filter) - a space-efficient probabilistic data structure used to test whether an element is in a set. It's stored in an optional bloom filter section and makes searching for keys that don't exist faster.

Additionally, there are several other less interesting sections, like the metadata section.

Compaction

What I described so far is already a functional key-value store. But there are a few challenges that would prevent using it in a production system: space and read amplification. *Space amplification* measures the ratio of storage space to the size of the logical data stored. Let's say, if a database needs 2MB of disk space to store key-value pairs that take 1MB, the space amplification is 2. Similarly, *read amplification* measures the number of IO operations to perform a logical read operation. I'll let you figure out what *write amplification* is as a little exercise.

Now, let's add more keys to the database and remove a few:

```
db.delete("chipmunk")
db.put("cat", "5")
db.put("raccoon", "6")
db.put("zebra", "7")
// Flush triggers
db.delete("raccoon")
db.put("cat", "8")
db.put("zebra", "9")
db.put("duck", "10")
```

Level 0

SST 1			SST 2		
Type	Key	Value	Type	Key	Value
PUT	cat	2	PUT	cat	5
PUT	chipmunk	1	DEL	chipmunk	
PUT	dog	4	PUT	raccoon	6
PUT	raccoon	3	PUT	zebra	7

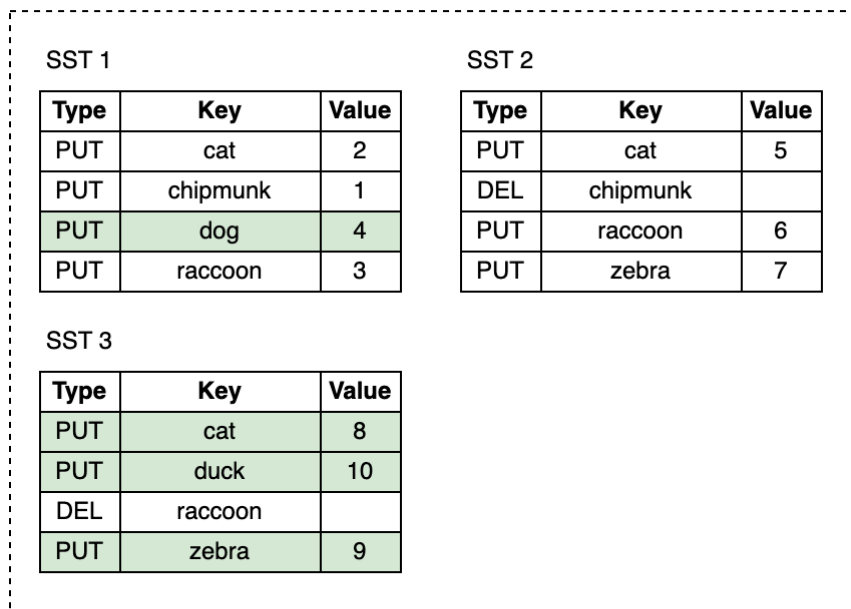
SST 3		
Type	Key	Value
PUT	cat	8
PUT	duck	10
DEL	raccoon	
PUT	zebra	9

As we keep writing, the memtables get flushed and the number of SST files on L0 keeps growing:

- The space taken by deleted or updated keys is never reclaimed. For example, the *cat* key has three copies, *chipmunk* and *raccoon* still take up space on the disk even though they're no longer needed.
- Reads get slower as their cost grows with the number of SST files on L0. Each key lookup requires inspecting every SST file to find the needed key.

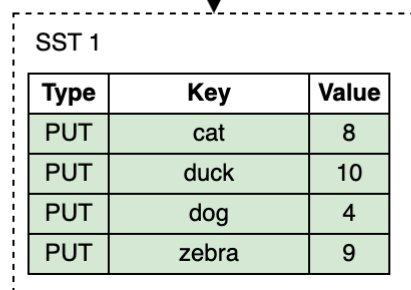
A background process called *compaction* helps to reduce space and read amplification in exchange for increased write amplification. Compaction selects SST files on one level and merges them with SST files on a level below, discarding deleted and overwritten keys.

Level 0

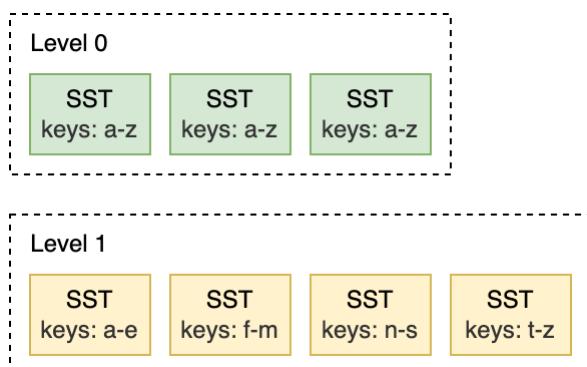


Compaction

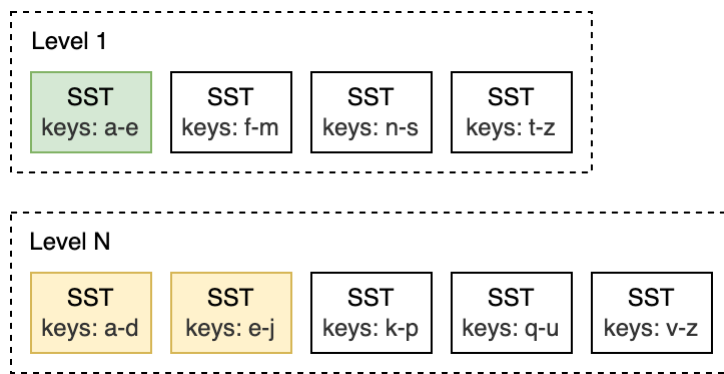
Level 1



Leveled Compaction is the default compaction strategy in RocksDB. With Leveled Compaction, key ranges of SST files on L0 overlap. Levels 1 and below are organized to contain a single sorted key range partitioned into multiple SST files, ensuring that there is no overlap in key ranges within a level. Compaction picks files on a level and merges them with the overlapping range of files on the level below. For example, during an L0 to L1 compaction, if the input files on L0 span the entire key range, the compaction has to pick all files from L0 and all files from L1.



For this L1 to L2 compaction below, the input file on L1 overlaps with two files on L2, so the compaction is limited only to a subset of files.



Compaction is triggered when the number of SST files on L0 reaches a certain threshold (4 by default). For L1 and below, compaction is triggered when the size of the entire level exceeds the configured *target size*. When this happens, it may trigger an L1 to L2 compaction. This way, an L0 to L1 compaction may cascade all the way to the bottommost level. After the compaction ends, RocksDB updates its metadata and removes compacted files from disk.

Note: RocksDB provides other compaction strategies offering different tradeoffs between space, read and write amplification.

Remember that keys in SST files are ordered? The ordering guarantee allows merging multiple SST files incrementally with the help of the [k-way merge algorithm](https://en.wikipedia.org/wiki/K-way_merge_algorithm) (https://en.wikipedia.org/wiki/K-way_merge_algorithm). *K-way merge* is a generalized version of the *two-way merge* that works similarly to the merge phase of the [merge sort](https://en.wikipedia.org/wiki/Merge_sort) (https://en.wikipedia.org/wiki/Merge_sort).

Read path

With immutable SST files persisted on disk, the read path is less sophisticated than the write path. A key lookup traverses the LSM-tree from the top to the bottom. It starts with the active memtable, descends to L0, and continues to lower levels until it finds the key or runs out of SST files to check.

Here are the lookup steps:

1. Search the active memtable.
2. Search immutable memtables.
3. Search all SST files on L0 starting from the most recently flushed.
4. For L1 and below, find a single SST file that may contain the key and search the file.

Searching an SST file involves:

1. (optional) Probe the bloom filter.
2. Search the index to find the block the key may belong to.
3. Read the block and try to find the key there.

That's it!

Consider this LSM-tree:

Active MemTable

Type	Key	Value
PUT	cat	4
DEL	chipmunk	

Immutable MemTable

Type	Key	Value
PUT	cat	3
PUT	catopuma	1
PUT	dog	2
PUT	zebra	1

Level 0

SST 1		
Type	Key	Value
PUT	cat	2
PUT	kangaroo	1
PUT	tiger	1
PUT	wombat	1

Level 1

SST 1		
Type	Key	Value
PUT	cat	1
PUT	chipmunk	1
PUT	dog	1
PUT	raccoon	1

Depending on the key, a lookup may end early at any step. For example, looking up *cat* or *chipmunk* ends after searching the active memtable. Searching for *raccoon*, which exists only on Level 1 or *manul*, which doesn't exist in the LSM-tree at all requires searching the entire tree.

Merge

RocksDB provides another feature that touches both read and write paths: the *Merge* operation. Imagine you store a list of integers in a database. Occasionally you need to extend the list. To modify the list, you read the existing value from the database, update it in memory and then write back the updated value. This is called "Read-Modify-Write" loop:

```
db = open_db(path)

// Read
old_val = db.get(key) // RocksDB stores keys and values as byte arrays.
old_list = deserialize_list(old_val) // old_list: [1, 2, 3]

// Modify
new_list = old_list.extend([4, 5, 6]) // new_list: [1, 2, 3, 4, 5, 6]
new_val = serialize_list(new_list)

// Write
db.put(key, new_val)

db.get(key) // deserialized value: [1, 2, 3, 4, 5, 6]
```

The approach works, but has some flaws:

- It's not thread-safe - two different threads may try to update the same key overwriting each other's updates.
- Write amplification - the cost of the update increases as the value gets larger. E.g., appending a single integer to a list of 100 requires reading 100 and writing back 101 integers.

In addition to the *Put* and *Delete* write operations, RocksDB supports a third write operation, *Merge*, which aims to solve these problems. The Merge operation requires providing a *Merge Operator* - a user-defined function responsible for combining incremental updates into a single value:

```
func merge_operator(existing_val, updates) {
    combined_list = deserialize_list(existing_val)
    for op in updates {
        combined_list.extend(op)
    }
    return serialize_list(combined_list)
}

db = open_db(path, {merge_operator: merge_operator})
// key's value is [1, 2, 3]

list_update = serialize_list([4, 5, 6])
db.merge(key, list_update)

db.get(key) // deserialized value: [1, 2, 3, 4, 5, 6]
```

The merge operator above combines incremental updates passed to the *Merge* calls into a single value. When *Merge* is called, RocksDB inserts only incremental updates into the memtable and the WAL. Later, during flush and compaction, RocksDB calls the merge operator function to combine the updates into a single large update or a single value whenever it's possible. On a *Get* call or an iteration, if there are any pending not-compacted updates, the same function is called to return a single combined value to the caller.

Merge is a good fit for write-heavy streaming applications that constantly need to make small updates to the existing values. So, where is the catch? Reads become more expensive - the work done on reads is not saved. Repetitive queries fetching the same keys have to do the same work over and over again until a flush and compaction are triggered. Like almost everything else in RocksDB, the behavior can be tuned by limiting the number of merge operands in the memtable or by reducing the number of SST files in L0.

Challenges

If the performance is critical for your application, the most challenging aspect of using RocksDB is configuring it appropriately for a specific workload. RocksDB offers numerous configuration options, and tuning them often requires understanding the database internals and diving deep into the RocksDB source code:

"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."

— [Official RocksDB Tuning Guide](#)

(<https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide>)

Final thoughts

Writing a production-grade key-value store from scratch is hard:

- Hardware and OS can betray you at any moment dropping or corrupting data.
- Performance optimizations require a large time investment.

RocksDB solves this allowing you to focus on the business logic instead. This makes RocksDB an excellent building block for databases.

I'm not a native English speaker, and I'm trying to improve my language skills. Feel free to correct me if you spot any spelling or grammatical errors!

← [Let's build a Full-Text Search engine \(/blog/2020/07/28/lets-build-a-full-text-search-engine/\)](#)

G

Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Share

Best Newest Oldest


Be the first to comment.

Subscribe

Privacy

Do Not Sell My Data

2013-2023.

 [_ \(https://github.com/akrylysov\)](https://github.com/akrylysov)

 [_ \(https://twitter.com/akrylysov\)](https://twitter.com/akrylysov)