About “Rocks For Every one”

Learning the RocksDB internals in “right amount of” depth.

Why do I write for “RocksDB for everyone”?

Prior joining Facebook, I read the source code of LevelDB, which was nicely written and very delightful to read. At that time, I managed to read through all the code and understood its high level design ideas—of course, there were still numerous nuances that I failed to pay attention to.

Luckily, at Facebook I got the opportunity to contribute to RocksDB, which was inherited from LevelDB and was improved for production workload. After looking into RocksDB closely and witnessing its evolution, a lot of puzzles about LevelDB/RocksDB were answered one by one.

After we open-sourced it, RocksDB is gaining increasing attention, from both industry and academia. A lot of people, with or without experience in database internals, are interested to learn more about this new power engine.

For me, I joined RocksDB team without much low level knowledge in database itself. However, as a *light weight* and *relative simple* database that absorbs wisdom from a variety of areas, I believe any one who have moderate background in system programming can learn a lot from RocksDB without struggling much.

That’s why I would like to share my thoughts in RocksDB with “everyone” who are interested.

What is this collection about?

The goals of this collection is to:

1. Give you a high-level view of RocksDB internals, which will be helpful when you are about to navigate in RocksDB’s codebase.
2. Highlight the good (and probably nuanced) design in RocksDB. I assume most of readers are not database experts (of course, I’m not expert, either). Thus, for many of you, who may already equipped yourself with some system programming knowledge, it could be very hard to understand some of the design and/or implementation details. I’d try my best to pick out these nuances and illustrate why they’re designed so.
3. Share the lesson I learned from developing RocksDB.

Also, here are the non-goals:

1. Not a code-level introduction. This collection only serves as an assistant during users’ RocksDB adventure.
2. Not a complete guide. I only pick the most interesting aspects about RocksDB in this collection.
3. Not an official documentation. For authoritative and update-to-date references of RocksDB, please check out RocksDB’s wiki page. I use this collection as a means for *personal* experience sharing.

So that’s the story behind this collection. Hope you enjoy RocksDB as much as I did.

How To Sound Like You Know RocksDB (1)

What People Talk About When They Talk About RocksDB

As the kick-off series of this collection, I'd like to give you a gentle bird-view over RocksDB. The **primary goal** of "How To Sound Like You Know RocksDB" series is to lay a good foundation for prospective RocksDB code readers.

Of course, most of us may just curious what the hell is RocksDB and why it's gaining increasing attention from the open source communities these days. Well, congratulations, the **secondary goal** of this series is: after reading these articles, even a lazy guy who knew little RocksDB can talk about it with RocksDB fans or even experts for 15 minutes without being seen through.

Yet-another key-value store?

One of the most frequently asked question about RocksDB was: why do we have one more key-value store? we've already have dynamo, redis, etc., why RocksDB shouldered in the already crowded key-value stores' party.

I'd say, RocksDB has *totally different* focuses compared with those data stores. Dhruba Borthakur from Facebook had excellent slides explaining what is RocksDB. In a nutshell, RocksDB is a data store that is:

1. **Persistent**: you can store you data safely in non-volatile storage.
2. **Designed for fast storage**: RocksDB has been optimized to work on flash devices or as a in-memory database; though it's performance is still pretty good on disk.
3. **Embedded**: RocksDB is a library so you can use it as a building block for your own program.

And here are RocksDB's non-goals:

1. **Distribution**
2. **Failover**
3. **High availability**

No panic, remember RocksDB is just a library and you can implement all the above-mentioned features on top of it. From this perspective, RocksDB can be viewed as a (relatively) low-level data engine that strives for high performance on faster storage.

LSM (Log-structured merge-tree)

Log-structured merge-tree is fundamental idea that fueled RocksDB, LevelDB (RocksDB's predecessor), BigTable, Apache HBase, etc.

We are not going to dig deep into LSM. Let's start with one simple fact: r*andom write (on either disk or flash) is bad.*

How bad? From "What are the numbers that every computer engineer should know", we can find that:

> Disk seek: **10,000,000 ns**
>
> Read 1 MB sequentially from disk: **30,000,000 ns**

That is, in disk we can do around 100 seeks/second, implying that significant overhead will be incurred when we perform a random write on the disk. (Flash devices, though optimized for random reads, still plagued by random writes).

That also explains one questions that puzzled many people when they first take the database class.

> If we enabled transaction log (or commit log, redo log, etc.) in a database, for each insertion or update operation, we will:
>
> 1. Write the data to the log file.
> 2. Write the data to the database itself.
>
> The data written to the log file is almost of the same size as that written to database. Will the write time be 2x longer?

The answer is no. This is because writing to log file requires only sequential writes, whereas updating the database may involve multiple random seeks.

So can we be more aggressive and design a database that only write to the log file (since the log file already contains all the truth about the data)?

That's exactly what motivates LSM.

This is an article about RocksDB. To be pragmatic, I'm not going to talk about the original LSM. Instead, I'll present a (overly) simplified architecture of RocksDB and illustrate how RocksDB employs that idea to minimize "random writes".

Again, please keep in mind that the examples below are just the handy simulations. As a first step, let's just focus on the basic idea and bear with the sub-optimum.

In the simplified RocksDB, there are 2 major components:

1. **Memtable**: a buffer that can temporarily host the incoming writes; memtable are normally sorted (in RocksDB, the default memtable implementation is SkipList), but that is not required.
2. **SSTable**: once a memtable reached a certain size, it will flush to the storage (to make it simple, let's assume we are talking disk) and generates a SSTable (sorted static table), which is immutable in its life time.

Optionally, to make sure data written to memtable is safe from process or server crash, we often enable the transaction log, which is essentially a redo log that can help RocksDB to rebuild the memtable after such events.

The figure below shows the basic components within RocksDB.

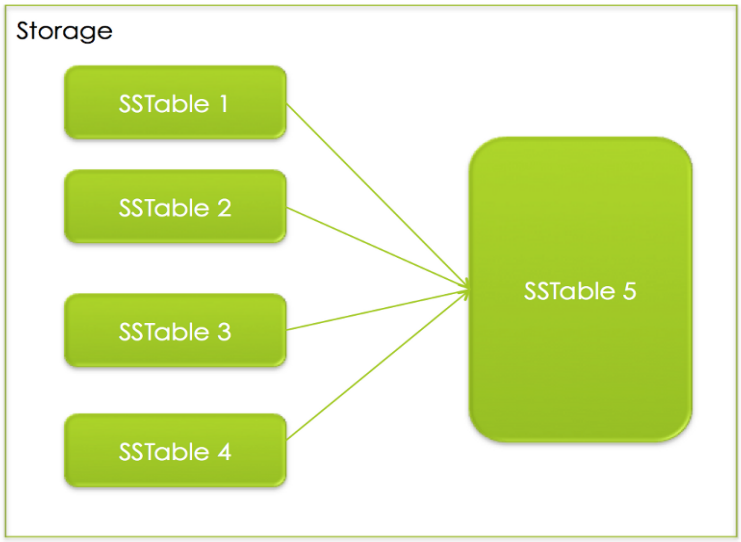
But! What if user writes the same key with different values? Simply speaking, by default RocksDB keeps all these copies, as a result:

- If we perform multiple Put() on the same key, RocksDB won't do any check but simply put them in the memtable first. Old version of the key-value pairs may be flushed to SSTables.
- When user do the Get(), we'll search from memtable to SSTable one by one because key may exist in any of them. So for a query key, it will first search memtable; if not found, search the newest SSTable; if again not found, search the second latest SSTable… Until we find the given key or pass the last file (Please note that this workflow is roughly similar to that of real RocksDB's, which involves lot's of complicated, delicate, carefully optimized details).

What! So if we keep writing same keys to the database and it will grow infinitely?

That's a valid concern! That's why in RocksDB, we have a background thread that periodically do the "compaction": merging these SSTables to a bigger SSTable.

As shown in the figure below, SSTable 1 ~ SSTable 4 are being compacted to a bigger SSTable. During the process we'll remove duplicate keys and even, for advanced usage, we may define our own "compaction filter" to filter out the unwanted key/value pairs as we wish (for example, based on it we can have time-to-live feature for each key-value pair).



That's pretty much it! We'll cover more on the memtable, SSTables, compaction algorithms in other article of this collections.

How To Sound Like You Know RocksDB (2)

A tale of two databases

LevelDB: RocksDB's predecessor

RocksDB is evolved from LevelDB. Much of LevelDB's code and architecture still plays significant role in RocksDB. From wikipedia:

> LevelDB is an open source on-disk key-value store written by Google Fellows Jeffrey Dean and Sanjay Ghemawat,who built parts of Google's platform. Inspired by BigTable, […] under the New BSD License and has been ported to a variety of Unix-based systems, […] Android.

As we discussed, LSM was the one of the fundamental ideas behind LevelDB. Be be more precise, LevelDB is based on BigTable's tablet implementation. BigTable wasn't open-sourced because of its dependency on multiple Google's internal libraries. Thus LevelDB authors, Jeff Dean and Sanjay Ghemawat, decided to build a light-weight system that resembles the BigTable's tablet—but simpler. At first, LevelDB was famously known to be used in Chrome.

LevelDB provides a list of features that are still being supported (and enhanced) by RocksDB:

Keys and values are arbitrary byte arrays.

1. Data is stored sorted by key.
2. The basic operations are **Put**(key,value), **Get**(key), **Delete**(key).
3. Multiple changes can be made in one **atomic batch**.
4. Users can create a **transient snapshot** to get a consistent view of data.
5. Forward and backward **iteration** is supported over the data.

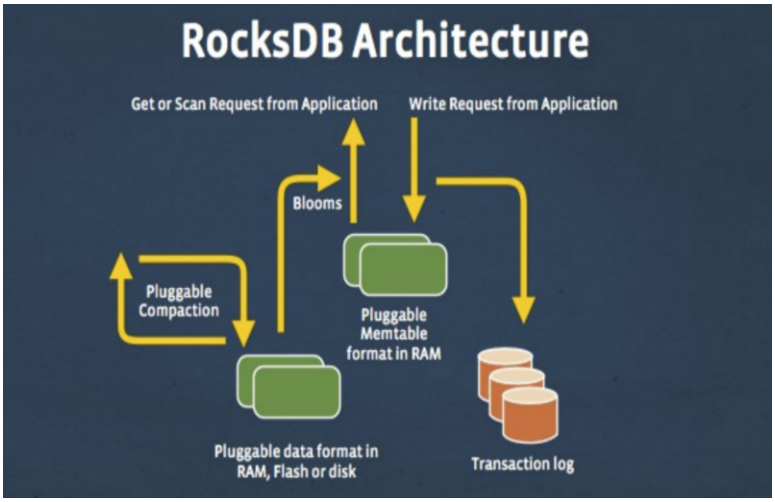
(*Excerpted from LevelDB's official site*)

RocksDB: for fast storage environments

RocksDB's story begins when Facebook was looking for an embedded database that could be a good candidate for databases that run on memory or flash devices. Eventually LevelDB stood out since it has best overall random read/write performance.

LevelDB was good, but not perfect, especially it wasn't designed for production workload. These slides has well explained the evolution of RocksDB. Especially, one of its slide highlights the similarities and differences between these two databases.

Also, a lot of components in RocksDB were made pluggable, ie. memtable, SSTable, even the compaction algorithms etc. Another slide described the new architecture:



Next…

In the next posts, we'll continue our tour to explore the architecture and dig deeper (well, not too deep, just right amount of depth) into the important components.

Speeding up Block-based Table Lookup

Prerequisites

Before reading this post, you may need to know:

- RocksDB stores its persistent data in **SST** (**S**tatic **S**orted **T**able).
- There are several types of SST, and in this post we want to talk about the optimization of block-based table—the default table format that works in disks and flash devices.
- RocksDB supports prefix seek, which has been heavily optimized by RocksDB team.

Problem

> One team uses RocksDB as its storage layer, which works perfectly until one day they significantly increased the data size, ensued by soaring CPU usage

From the profiling tool, we identify that the heavy CPU consumption was caused by the binary search in the memtable and block-based table (each ate approximately **30%** of the total CPU). As we already had the support for hash-based memtable, the binary search was easily eliminated and CPU usage from memtable had soon dropped to under **1%**.

As for block-based table, hmm, there was not a readily available solution. In block-based table, entries are compactly stored in an ascending order. Every *4KB* entries will be grouped as a "block", which is the finest granularity that we write/read data from filesystem.

For every block, we'll extract its last entry to the *index*, which will be loaded to the memory as we open the table. When RocksDB performs a lookup for a key ***k*** in a table, it will

1. do a binary search in the in-memory index.
2. locate the block, do a internal binary search to check if ***k*** exists (the internal search is complicated and well optimized, I may discuss this topic in another post).

And the heavy CPU consumption occurs in step 1.

Another interesting characteristic of this use case is: it only perform prefix lookup and retrieve a group of entry of the same prefix. The concept of prefix may sound too abstract to you. To help you understand this feature, I'd like to give a concrete example.

> RocksDB is a key/value database—keys and values are essentially an array of arbitrary length of bytes. Although there is great flexibility to construct your key, one of the common pattern is to use prefix as identifier of the entries of similar properties. For example, we may use <user-id>.<tiemstamp>.<activity-type> as the key for a data store the keeps the user activities. With this scheme, we can perform prefix seek to get the database iterator that points to the first activity of a user in a given time range, and then efficiently perform Iterator::Next() to retrieval subsequent entries.

Solution: *V1*

After analyzing the use case, we realized the essential problem was how to quickly find the mapping *prefix => block id*.

Naturally, we considered to add a hash table to remember such mapping. The challenges is, the index data structure does more than random lookup; it also supports range lookup with *Prev()/Next()*.

Instead of providing a radically new index format for block-based table, we decided to add an additional/optional hash layer on top of the existing data structure.

Here's how it works:

- When we construct the table, we'll detect the unique prefixes (this is very easy to do because the entries are already sorted) and store the prefixes and their mapped block ids to a metablock (an extensible block that makes future modification on table format easy) inside of this table.
- When opening a table, if we enabled hash-prefix lookup, we'll scan through the materialized *prefix=>hash id* mapping and construct an in-memory hash map (for v1, we happily use std::unordered_map).
- When the in-memory index calls *Seek()*, It will consult the hash table to find the mapped block for a given prefix.

This solution works. It also reduced the expensive binary search from **30%** to almost nothing. But we all know the size of hash index is determined by the number of unique prefixes.

For this particular use case, the memory usage had increased by 20%, but that's fine, we still have much capacity left. In this solution, we care mostly about the cleanness of the newly added semantic and a pivot solution that proves hash solution is the right way to go—and if this solution happens to work, we'd happy to stick to it (however, in retrospection, I think in the API layer, we'd at least warn users of the potential high memory usage).

Little optimization (except the optimization that avoids frequent memory allocation on the heap) is done since we treated it as an experimental feature and want to optimize only after we gain more experience for similar use cases.

Solution: v2

Everything changes. Soon, we discover that in some server there are significantly more unique prefixes, resulting in out-of-memory. So more efficient memory is mandatory right now.

From the profiling result, the size of prefixes (in memory) accounts for 98% of the whole index. There seems to be many ways to optimize the hash table because from the profiling results, for example, std::unordered_map is not most space efficient, we can reduce memory usage by 10%~20% by writing a highly specialized hash table. Also, we may compress the prefixes to further cut the memory usage, etc.

Those solution sounds feasible but even 30% reduction seems still not enough. How can we squeeze even more from the memory?

As a matter of fact, back to our initial observation, essentially we only want to speed up the lookup for the mapping *prefix => block id.* So, do really we need to stored the prefixes at all? The whole purpose of a prefix in our solution are only to be

1. Hashed
2. Compared with the query prefix to see if they match.

Can we skip (2) and live without prefix? Yes, we can!

In our improved solution, instead of using a hash map for the *prefix => block id* mappings, we only maintain an array (essential a hash set) to store mapped block id (without any prefix being stored).

Workflow for constructing memory friendly hash index:

- When opening the table, scan through the materialized *prefix => block id* mappings from the on-disk file.
- For each prefix, hash it and map it to a hash bucket. We'll store only the associated block id for that prefix.

For example: if prefix "abc" (maps to block 3), "def" (maps to block 9) are both hashed to "1", then the hash bucket 1 will be a linked list of "3" and "9".

The tradeoff is, with the new hash index, when we search prefix "abc", it may take two searches inside both block block 3 and block 9. And for non-existing prefix "xyz" that happens to be hashed to 1, we don't have a quick way to determine that "xyz" doesn't exist; instead, we need to jump into both block 3 and block 9 to see if "xyz" exists.

However, if we hash the prefixes deliberately, there are normally not many block ids inside a bucket (most of them should have 1~2 entries). As a result, we traded off a little lookup performance with a 97% reduction for the hash index.

Lessons learned

- **Start with a simplest solution and don't do premature optimization:** it's tempting to start with a big upfront designed solution and try to nuke all problem with it. But too often we failed to solve the real problem. Before V1 we don't even know how much hash table can alleviate the problem. Looking back it's rewarding to come up with a quick solution to verify our hypothesis, and then based on the improvement, we can come up a follow-up solution with well-directed optimizations.
- **The solution was often simpler than you thought.** Sometimes as we are busy solving the problem, we may focus too much on the immediate problems. By taking as step and reviewing the original problem that we want to solve, we can often avoid ourselves getting trapped by the local maximums.