**Intro to RocksDB**

RocksDB is a modified version of LevelDB and was developed at Facebook.

At facebook, the I/O and CPU were efficient and meeting the SLA. However, they found storage capacity to be the bottleneck since storage on SSD is expensive. Hence, they developed RocksDB with increase storage resource efficiency in mind.

Results show that for MyRocks (MySQL on RocksDb) consumes only 50% of storage size compared to MySQL on InnoDB.

With space amplification in mind, we study the following aspects:

* Comparative study of compression algorithms
* Standard I/O vs Direct I/O
* Fadvise
* Compaction

 **Comparative study of compression algorithms**

RocksDB uses compression to further reduce space amplification. It provides the following methods for compression -

1. *Prefix Encoding –*

It is applied on keys by not writing repeated prefix of previous keys. It reduces space requirements by 3%-17% depending on the data workload.

1. *Sequence ID garbage collection –*

The sequence ID of the key older than the oldest snapshot taken is removed. The users can take snapshot of the current state of the database to be able to access older versions of data. Removing snapshot ID helps with compression because the it is 7 bytes long and doesn’t compress well. This optimization reduces space requirements from between 0.03% (e.g., for a database storing social graph vertexes that will have large values) and 23% (e.g., for a database storing social graph edges that will have empty values).

1. *Dictionary based compression –*

A data dictionary can be used to further improve compression. Especially, in case of small data blocks, it gives good compression. It reduces space requirement further by 3%.

1. *Data compression –*

RocksDB supports a number of compression algorithms. Each level can be configured to use a different compression algorithm or no compression at all.

Compression is applied on per block basis.

Weaker compression algorithms can reduce the space requirement to 40% and strong compression algorithms can reduce the space requirement to 25%.

In our study, we found that Snappy algorithm gives a reasonable throughput and strong compression which makes it the most popular choice. Zlib was found to be the strongest compression algorithm amongst the available options.

For reasonable performance and compression, zlib compression algorithm is used for the last level since the last level contains 90% of the data and receives least number of reads and writes.

For other levels, either snappy or no compression can be used.

**Direct IO and Fadvise**

1. Setting fadvise isn't a good setting for mixed workload.
2. Why do we need Direct I/O: 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 implements 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.
3. 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.

**Configure minimum compaction speed adequate for workload**

During compaction, small SST files are read into memory, sorted and written as large SST file. This process consumes considerable amount of SSD bandwidth. If configured compaction speed is lesser than optimal value for a given workload, higher compaction speed results in higher foreground write throughput. Because inadequate compaction speed causes foreground write to stall frequently and hence lesser write throughput. But, as soon as we start increasing compaction speed above its optimal value for a given workload, foreground write starts to slow down due to wastage of SSD bandwidth by compaction process.

When compaction speed is higher than optimal, data is pushed faster to lower level. This

makes RocksDB apt to handle spikes in workload but results in higher write amplification, wastage of SSD bandwidth and SSD wear-out. Hence compaction speed should be configured considering workload and nature of application. Spikes in workload may cause foreground write to stall. If application can’t tolerate any latency caused by write stalls due to spikes in the workload, compaction speed should be adjusted to handle such spikes in the workload.

RocksDB supports three styles of compaction :-

 (a)Levelled compaction

 (b)Universal compaction

 (c) FIFO compaction

Levelled compaction and Universal compaction is generally used because FIFO style compaction is considered unreliable. Universal style compaction generally results in lower write amplification than Levelled compaction. But, it has higher space and read amplification.



**Other Observations***:*

* Writing complete pages is better than writing data smaller than the page size, since the smallest SSD unit storage is a page. Because updating page will effectively allocate a new page and invalidate the previous one, updates may result into Garbage Collection.
* It’s better to keep the write operations page-aligned in order to avoid additional write multiplication.
* Keeping the data with similar lifecycle together will be beneficial for performance.
* Most of these points are points speak favour of immutable LSM-like Storage, rather than systems that allows in-place updates: writes are batched and SSTables are written sequentially, files are immutable and, when deleted, the whole file is invalidated at once. Unlike Write-Ahead Log , Memory Tables pre-sort the data before it reaches disk in order to facilitate sequential read access. Records that are more likely to be read together, are written together.
* In this way, RocksDb satisfies the unwritten contracts mentioned in the SSD paper (http://pages.cs.wisc.edu/~jhe/eurosys17-he.pdf)

 **Conclusions**

* Current SSD technology suffers from the performance degradation caused by write amplification. Minimal read unit on SSD is page. Reads and writes are performed in pages. Deleting a page worth of data does not immediately remove data physically. Instead, a page is marked as stale and will wait for Garbage Collection to reclaim free space.
* Because writes are performed in pages, even if a single byte has to be updated, the whole page will be written anyway.
* At the same time, because of the specifics of NAND storage, pages can not be updated in place, so writes can be performed only into the empty pages.
* These two properties attribute for the write amplification on SSDs.
* So keeping that in mind, we had focused our work on: compression, direct io , fadvise & compaction. Below are the final conclusions for each of the above aspects.
1. Compression: Snappy gives the best throughput and proves to be a reasonably strong compression algorithm and is most popular while Zlib is used for strong compression.
2. direct io & fadvise: Using O\_DIRECT will most likely require us to write a buffer cache our-self (taking care of alignment). Until Rocksdb supports DirectIO for WAL, it is better not to use this option. Instead, keep using Kernel Buffer Cache with fadvise based on workload
3. Compaction: Use minimum compaction speed adequate for workload.