Manos: A Benchmarking System for RocksDB

Manuja DeSilva
Boston University

Michael Hendrick
Boston University

ABSTRACT

RocksDB [2] is an in-memory database that is built using the concept of LSM(log-structured-merge) trees. Out of the box, it provides some guarantees, such as that its default parameters are suitable for most workloads. In addition, it also provides many knobs that can be adjusted by the user (the developer) to optimize the database for their particular workloads.

We will build a command line interface tool to simulate workloads on RocksDB. We will include the ability to tune various parameters of RocksDB and run workloads using those parameter configurations. In addition to running these aggregated style experiments, we will also provide the ability to run individual experiments testing the affects of tuning one parameter on a dataset. Lastly, we will run our own benchmarking experiments using the CLI tool we built to test the guarantees cited by RocksDB, and devise our own parameter configurations that we predict are suitable for different types of workloads.

1 INTRODUCTION

1.1 Motivation

Unlike other databases, developers must implement functionality within their applications themselves to utilize RocksDB, due to the fact that RocksDB runs at the application layer itself, rather than being on a separate database layer. Most of these other databases can be accessed and manipulated using well-documented APIs that don't require expertise in computer science to understand. In addition, other databases provide a wealth of well documented APIs to adjust various knobs to improve performance. RocksDB, in addition to providing the ability to perform basic operations such as Gets, Puts, and Deletes, also offers a wealth of knobs for tuning. However, at the time of writing, these knobs require extensive amounts of non-trivial research to implement. So, currently, developers do not have a method of easily simulating their existing workloads to observe how RocksDB performs under certain workload proportions, and ultimately ascertain if RocksDB can provide the performance necessary for their applications. In addition, developers cannot currently easily simulate workloads running on custom parameter configurations. We aim to build a tool that can do both; that is, provide developers with an application to easily tune RocksDB and simulate custom workloads on RocksDB.

1.2 Problem Statement

In this project, we would like to explore how RocksDB performs under various workloads, using different proportions of point and range queries, inserts, updates, and deletes. In addition to performing workloads using the default RocksDB [2] parameters, we would like to perform experiments after tuning various RocksDB parameters to observe the effects of performance after tuning. We aim to test the in-memory store performance of RocksDB, as well as provide a CLI application that developers can utilize to perform benchmarking for their applications.

1.3 Contributions

We built a easy to use command line interface benchmarking tool that allows developers to

- Create custom workloads
- Easily tune RocksDB parameters
- Run aggregated experiments
- Run individual experiments using predefined experiment templates
- Create their own experiments

In addition to writing the benchmarking tool, we ran our own experiments on various RocksDB parameters and realized that tuning some knobs provided negligible results after some point. We also devised our own set of configuration options for read optimized and write optimized workloads using information gleamed from the sum of our experiments.

2 BACKGROUND

To build our benchmarking tool, we first developed a solid understanding of how log structured merge trees function. Then, we learned about the data structures utilized by RocksDB in its LSM tree implementation, and how those data structures affect performance. Finally, we read through the RocksDB documentation [2] and learned about the various tunable parameters it offers, and spent additional time browsing through the RocksDB codebase learning how to adjust those parameters.

3 BENCHMARK

For our project, we wrote a C++ command line interface application that instantiates a temporary database with a size of the user's choosing and then simulates a workload with the following operations, with the proportion of each operation relative to the total number of operations set accordingly by the user:

- Point queries
- Range queries
- Point inserts
- Point updates
- Point deletes
- Range inserts
- Range updates
- Range deletes

Each operation returns the time spent performing the operation, and will also return RocksDB statistics relevant to the operation.

Users can also choose to tune several RocksDB parameters from the command line interface, or go with the defaults, and observe the effects of the tuned parameters on the performance of the workload. Some of these parameters are explained in detail below.

3.0.1 Memtable Size. Writes in RocksDB are initially written to a **Memtable**, a set of in-memory write-buffers. The size of the memtable is important when taking scenarios such as bulk loading data into account. If the size of the dataset is consistently larger than the memtable size, then subsequent writes will be blocked when all memtables are full and waiting to be flushed to disk. So, as the size of the memtable increases, more data can be written to memory without waiting to flush to disk, and thus leads to a decrease in write time.

3.0.2 LRU Cache size. After a memtable is flushed to disk, reads are served from disk if the key is no longer present in the memtable. To improve performance, RocksDB utilizes a block cache to cache data served from disk so that subsequent reads can derive the data from the cache rather than going to disk. By default, RocksDB utilizes a Least Recently Used(LRU) cache with a 8MB capacity. By increasing the size of the LRU cache, we can increase the number of cache hits, and decrease read times (read amplification)

3.0.3 Number of Memtables. By default, RocksDB uses 2 memtables. However, when writing large amounts of data, if the memtable size is too small and all allocated memtables are full, then writes are stalled until at least one memtable is flushed to disk. To improve write times, the number of memtables can be increased to account for larger, potentially unbounded datasets, such as when utilizing RocksDB as the backend data store for streaming applications.

3.0.4 Bloom filters. Once memtables are full they are flushed to disk and maintained across **Static Sorted Table (sst)** files. When serving reads from disk, e.g when the data item does not exist in the Memtable or the block cache, reads may have to perform several disk IOs to get the data item from disk. Bloom filters can prevent these extra lookups by overlaying bit arrays over each file. Bloom filters can tell us whether the data item might exist in an object, or if it definitely does not exist within an object. Bloom filters are especially useful for point queries. By increasing the size of the bloom filter, we can decrease the amount of false positives when performing reads, thus leading to improved read amplification.

3.0.5 Allocation of Threads. In most LSM based architectures, there are two main processes, flushing and compaction. Flushing moves data from memory to disk, and compaction merges files, processes the deletions of keys, and removes multiple copies of the same key if it has been overwritten. Write rates can increase if compaction requests are issued concurrently using multiple threads.

3.0.6 Compaction style. Out of the box, RocksDB supports two types of compaction. **Level-style** takes up less space, optimizing space amplification by minimizing the number files in each compaction step. One compaction step will merge one file in level n with all of its overlapping files in level $n+1$. Another type of compaction, **Universal style**, requires more temporary space by potentially merging many files and levels at once, but leads to lower write amplification.

4 RESULTS

All experiments were performed on a dataset of 10 million tuples.

4.1 Individual Experiments

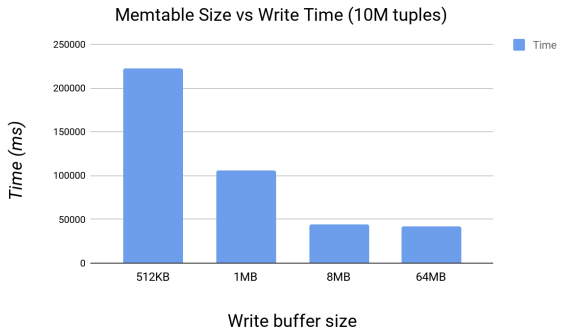

Figure 1: We observed that as we increased the size of the memtable, the write time decreased tremendously, by several magnitudes.

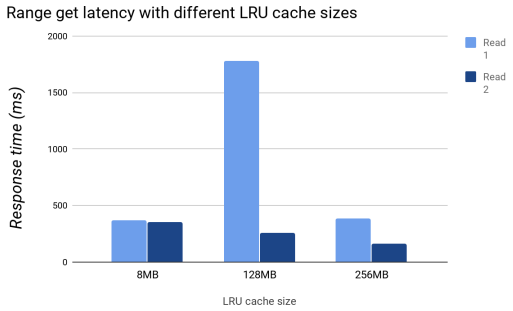

Figure 2: After increasing the size of the LRU cache, we observed that subsequent reads served after a primary read took less time to be delivered when performing Range Gets.

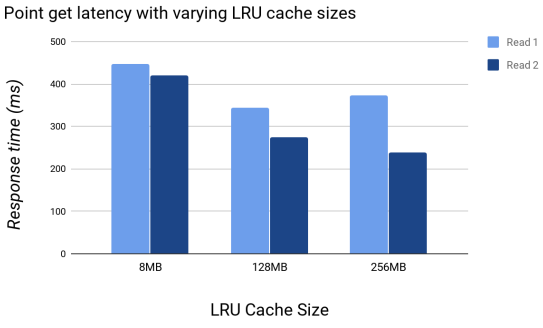

Figure 3: As with range gets, we observed that increasing the LRU cache size resulted in quicker reads.

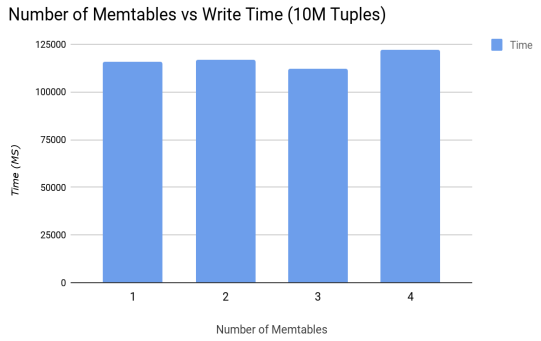

Figure 4: Increasing the number of memtables had negligible effects on write time, in contrast to our hypothesis.

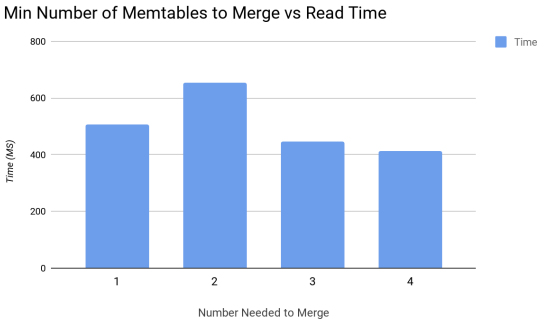

Figure 5: Just as we theorized, increasing the minimum number of tables to merge resulted in quicker reads as more reads were served from memory rather than disk. This is a useful parameter to tune for read and write optimized workloads where greater space amplification is not considered overhead.

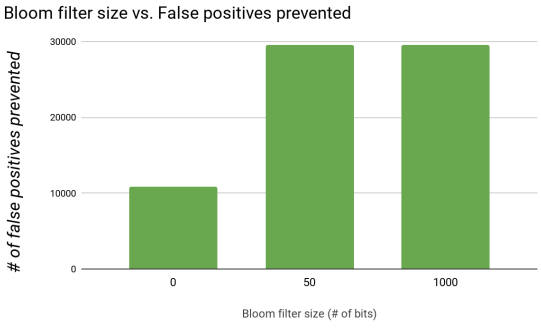

Figure 6: Increasing the number of bloom filter bits improved read amplification dramatically; but we found that it provided neglibible results after around 50 bits.

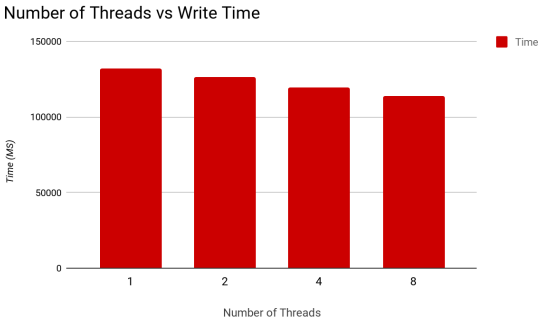

Figure 7: We observed that as we increased the number of threads in the shared thread pool for flushes and compactions, the write time decreased.

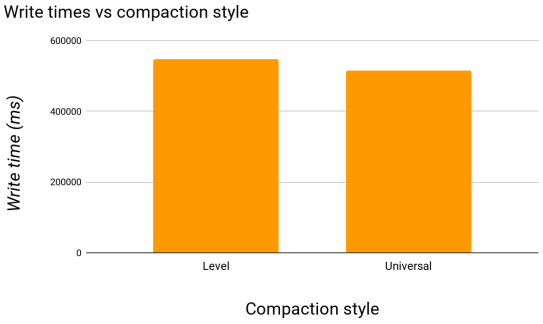

Figure 8: As we predicted, universal style compaction allowed for quicker writes.

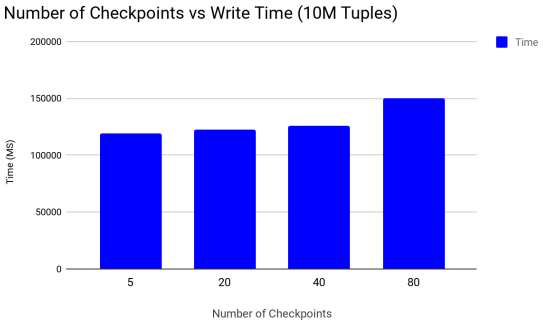

Figure 9: The streaming data processing software, Apache Flink [1], uses RocksDB to store snapshots of streaming data at regular 'checkpoints'. We observed that as we increase the number of checkpoints, write latency increases dramatically.

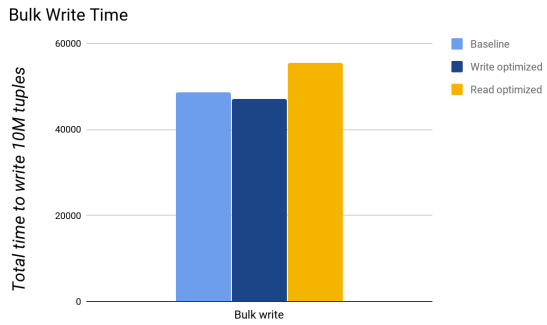

Figure 10

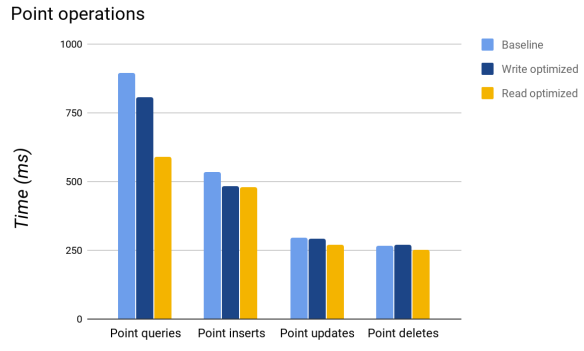

Figure 11

4.2 Aggregated Experiments

Based on the observations of the effects of tuning individual parameters on simulated workloads, we created our own custom parameter profiles for both write optimized and read optimized workloads.

4.2.1 *Write Optimized.*

- LRU Cache Size: Default
- Memtable Size: Default
- Bloom Filter Size: Default
- # of Memtables: 5
- Compaction Style: Universal

4.2.2 *Read Optimized.*

- LRU Cache Size: 512MB
- Memtable Size: 256MB
- Bloom Filter Size: 100 bits
- # of Memtables: Default
- Compaction Style: Level

As shown in Figures 10, 11, and 12, the write optimized parameter profile that we formulated indeed allowed for faster inserts and updates, although only slightly better than the default RocksDB parameters. For both range and point queries, the read optimized parameter profile allowed for much faster reads than the default RocksDB parameters, although with the cost of higher write times when bulk loading data.

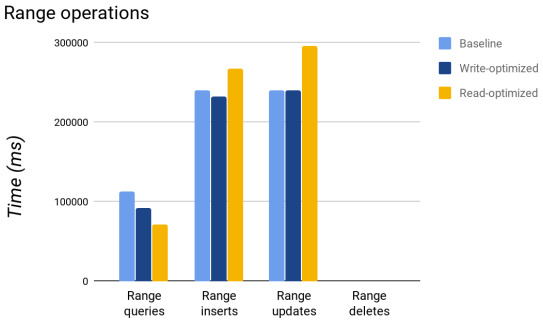

Figure 12

5 CONCLUSION

Using the CLI tool that we built, we were able quickly and efficiently simulate workloads with a wide variety of tunable parameter configurations. This enabled us to test the many promises of RocksDB, put its through its limits, and measure the benefits of an in-memory data store, whose underlying performance promise that was made even more profound after tuning various parameters. Through our observations, we were able to build our own parameter profiles for write-optimized and read optimized workloads that we can apply to real world datasets. In the future, we hope to dynamically add more statistics to our benchmarks, add the ability to tune more RocksDB parameters, and also execute more fine grained experiments on each RocksDB parameter.

REFERENCES

[1] Apache. 2011. Flink. https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/state_backends.html
[2] Facebok. 2012. RocksDB. https://github.com/facebook/rocksdb/wiki