Optimization of RocksDB for Redis on Flash

Keren Ouaknine
Hebrew University
Givat Ram Jerusalem
9190401 Israel
ouaknine@cs.huji.ac.il

Oran Agra
Redis Labs
Habarzel 28 Tel-Aviv
6971040 Israel
oran@redislabs.com

Zvika Guz
Samsung Semiconductor
3655 N 1st st. San Jose CA
95134 USA
zvika.guz@samsung.com

ABSTRACT

RocksDB is a popular key-value store, optimized for fast storage. With Solid-State Drives (SSDs) becoming prevalent, RocksDB gained widespread adoption and is now common in production settings. Specifically, various software stacks embed RocksDB as a storage engine to optimize access to block storage. Unfortunately, tuning RocksDB is a complex task, involving many parameters with different degrees of dependencies. As we show in this paper, a highly tuned configuration can improve performance by an order of magnitude over the baseline configuration.

In this paper, we describe our experience optimizing RocksDB for Redis-on-Flash (RoF) – a commercial implementation of the Redis in-memory key-value store that uses SSDs as RAM extension to dramatically increase the effective per-node capacity. RoF stores hot values in RAM, and utilizes RocksDB to store and manage cold data on SSD drives. We describe our methodology for tuning RocksDB parameters and present our experiments and findings (including both positive and negative tuning results) on two clouds: EC2 and GCE. Overall, we show how tuning RocksDB improved the database replication time for RoF by more than 11x. We hope that our experience will help others adopt, configure, and tune RocksDB in order to realize its full performance potential.

CCS Concepts

•**Information systems** → **Key-value stores; Database performance evaluation;**

Keywords

Databases, Benchmark, Redis, Rocksdb, Key-Value Store, SSD, NVMe

1. INTRODUCTION

RocksDB is a persistent key-value (KV) store that was specifically architected for fast storage, mainly flash-based SSDs [1]. Forked from LevelDB [2], RocksDB provides superior performance [3], and was designed to be highly flexible in order to facilitate embedding as a storage engine by higher-level applications. Indeed, many large-scale production applications use RocksDB to manage storage, leveraging its high performance to mitigate the ever-growing pressure on the storage-system [4].

Unfortunately, RocksDB flexibility and superior performance come at a cost: tuning RocksDB is a complex task that involves more than a hundred parameters with varying levels of inter-dependencies. Furthermore, "while recent changes have made RocksDB better, it is much harder to configure than LevelDB"; too often poor results "are caused by misconfiguration" [5].

The main questions raised when operating with RocksDB are: (1) which configuration parameters should be used for which hardware and under what workload? (2) what are the optimal values for these parameters? (3) are parameters interdependent (i.e., tuning parameter a works if and only if parameters b, c and d have certain values)? (4) will the positive optimization from two different tunings cumulate or negate when brought together? Last but not least, (5) what, if any, are the side effects of these optimizations?

This paper seeks to answer these questions by sharing our experience optimizing RocksDB in the context of Redis-on-Flash (RoF) [6, 7] – a commercial extension to the popular Redis in-memory key value store [8]. RoF uses SSDs as a RAM extension to provide competitive performance to the in-memory Redis variant while dramatically increasing the effective dataset capacity that can be stored on a single server. In RoF, hot values are saved in RAM, while cold-values are saved in SSDs and are managed by RocksDB (See Section 2.2). Because RocksDB handles all of RoF accesses to storage, its performance plays a major role in the overall system performance, especially for use cases with low access locality. Since RoF aims to provide competitive performance to the pure-RAM Redis variant, tuning RocksDB proved to be a key challenge.

During the process of tuning RocksDB for the RoF case, we analyzed a large set of parameters and experimented with their impact on the performance for several different workloads – database replication, a write-only workload, and a 50-50 read:write workload. To verify the robustness of our settings across different hardware setups, we run all experiments in both Amazon Elastic Compute Cloud (EC2) and Google Compute Engine (GCE). Overall, our tuning reduced the time needed to replicated a node by more than 11x. The bulk of this paper describes the methodology, tuning process, and specific parameters settings that lead us to this result.

In Section 3, we describe our methodology and explain the experiments process. Then, in Section 4 we detail the parameters tuning that had the largest positive effect on performance. We also specifically list parameters for which we expected performance improvement but instead either reduced performance or had other negative

ICCDA '17 May 19-23, 2017, Lakeland, FL, USA

ACM ISBN 978-1-4503-5241-3/17/05…$15.00
http://dx.doi.org/10.1145/3093241.3093278

side effects, as we believe this information will be useful to others. While our experiments were done in the context of RoF, we expect similar systems to require similar configuration, and hope that our methodology and results will help reduce the tuning time beyond the scope of this specific study.

In summary, this paper makes the following contributions:

- We present our RocksDB benchmark results and analysis for several workloads under the two main clouds: EC2 and GCE,
- We describe our tuning process, the parameters that had the largest positive effect on performance, and their optimal settings; and
- We describe negative results for tuning efforts that either did not pan out, reduced performance, or had non-intuitive side effects.

2. BACKGROUND

This section briefly overviews Redis, Redis on Flash, and RocksDB. It describes the high-level architecture of these systems and provides the necessary background for understanding the details brought up in the rest of the paper.

2.1 Redis

Redis (Remote Dictionary Server) [8] is a popular open-source in-memory key-value store that provides advanced Key-Value abstraction. Redis is single-threaded, it handles a command from just one client at a time in the process' main thread. Unlike traditional KV systems where keys are of a simple data type (usually strings), keys in Redis can function as complex data types such as hashes, lists, sets, and sorted sets. Furthermore, Redis enables complex atomic operations on these data types (e.g., enqueuing and dequeuing from a list, inserting a new value with a given score to a sorted set, etc.). Redis abstraction and high ingestion speed have proven to be particularly useful for many latency-sensitive tasks. Consequently, Redis has gained wide-spread adoption, and is used by a growing number of companies in production setting [9].

Redis supports high availability and persistence. High availability is achieved by replicating the data from the master nodes to the slave nodes and syncing them. When a master process fails, its corresponding slave process is ready to take over following a process called failover. Persistence can be configured by either one of the following two options: (1) using a point-in-time snapshot file called RDB (Redis Database), or (2) using a change log file called AOF (Append Only File). Note that these three mechanisms (AOF rewrite, RDB snapshot, and replication) rely on a fork to acquire a point in time snapshot of the process memory and serializing it (while the main process keeps serving client commands).

2.2 Redis on Flash

In-memory databases like Redis store their data in DRAM. This makes them fast yet expensive, because (1) DRAM capacity per node is limited, and (2) DRAM price per GB is relatively high. Redis on Flash (RoF) [6, 7] is a commercial extension to Redis that uses SSDs as RAM extension to dramatically increase the effective dataset capacity on a single server. RoF is fully compatible with the open-source Redis and implements the entire Redis command set and features. RoF uses the same mechanisms as Redis to provide high-availability and persistence rather than relying on the non-volatile property of flash.

RoF keeps hot values in RAM and evicts cold values to the flash drives. It utilizes RocksDB as its storage engine: all drives are managed by RocksDB, and accesses to values on the drives are done via RocksDB interface. When a client requests a cold value, the request is temporally blocked while a designated RoF I/O thread submits the I/O request to RocksDB. During this time, the main Redis thread serves incoming requests from other clients.

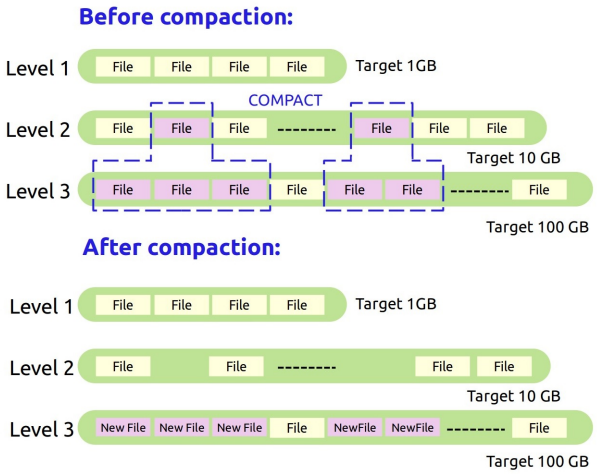

Figure 1: Illustration of the compaction process

2.3 RocksDB

RocksDB [10] is an open source key-value store implemented in C++. It supports operations such as get, put, delete, and scan of key-values. RocksDB can ingest massive amounts of data. It uses SST (sorted static tables) files to store the data on NVMes, SATA SSDs, or spinning disks while aiming to minimize latency. RocksDB uses bloom filters to determine the presence of a key in a SST file. It avoids the cost of random writes by cumulating data to memtables in RAM, and then flushing them to disks in bulks. RocksDB files are immutable: once created they are never overwritten. Records are not updated nor deleted, and instead new files are created. This generates redundant data on disk, and requires regular database compaction. Compaction of files remove duplicate keys and process key deletions to free up space as shown in Figure 1.

2.3.1 RocksDB Architecture

RocksDB organizes data in different sorted runs called levels. Each level has a target size. The target size of levels increases by the same size multiplier (default x10). Therefore, if the target size of level 1 is 1GB, the target size of level 2, 3, 4 will be 10GB, 100GB, and 1000GB. A key can appear on multiple levels, but the most up-to-date value is located higher in the levels hierarchy as older keys are pushed down during compaction.

RocksDB initially stores new writes in RAM using memtables. When a memtable is filled up, it is converted to an immutable memtable and inserted into the flush pipeline at which point a new memtable is allocated for the following writes. Level 0 is an exact copy of the memtables. When level 0 fill ups, the data is compacted, i.e. pushed down to the deeper levels. The compaction process applies to all levels, and merges files together from level N to level N+1 as shown in Figure 1.

2.3.2 Amplification factors

We measured the impact of the optimizations by monitoring the throughput and duration of the experiments under various work-

loads. Additionally, we monitored side effects based on the amplification factors of RocksDB defined as follows:

- **Read Amplification** is the ratio between the total number of bytes read from the disk (including the ones read to process data compaction) and the bytes read from the KV-store
- **Write amplification** is the ratio between the total bytes written to the disk and the bytes written to KV-storage
- **Space amplification** is the ratio between the size of the database files on disk and the KV-storage size.

2.3.3 Memtier benchmark

Memtier benchmark [11] is the benchmark tool we used to send traffic to Redis. It was developed and open-sourced by Redis Labs [12]. It can send reads and writes at various ratios, and implements several different traffic patterns (e.g., sequential, random, Gaussian, etc.). The tool maintains a pipeline of Redis commands which sends a new request only when a reply returned. The memtier command-line tool provides control over multiple aspects of the request stream, e.g., operations number, read/write ratio, worker threads number, clients number, value size, etc. At the end of each run, it reports back aggregated averages of reads throughput, writes throughput, and their latency.

3. METHODOLOGY

Our initial motivation when optimizing RocksDB, as well as our primary optimization target was to minimize the overall RoF database replication time. The replication process is needed to provide high availability to the master nodes and consists of two steps: (1) reading the entire dataset from the master server and sending it over the network to the slave server, and (2) writing the dataset on the slave server. Once this initial replication process is complete, all additional changes on the master will be sent to the slave to keep it in-sync with the master. When (if) a master fails, the slave becomes the new master and a new slave will be replicated so that the database remains fault-tolerant. Hence, Replication time is important because (1) during the replication process the cluster operates at a lower performance point due to the master being busy reading and sending data to the network, and (2) there is a risk for data loss, because there is no other data replica available in case of a master failure.

Using the default RocksDB settings, the time to replicate a 50GB with a RAM:flash ratio of 10:90 was a whopping 212 minutes. This is prohibitively long for our use-case and had to be trimmed down to less than 30 minutes. In Section 4, we describe the steps we took and the configuration changes we applied to reduce the replication time to 18 minutes. Since real production systems usually run up to 50GB on a single Redis server process, we chose that to be our experiments database size.

While our primary target was to minimize the database replication time, it was imperative to ensure that our settings do not degrade the system steady-state performance. Thus, for every optimization evaluated, we measured performance for four types of workloads:

1. Write-only workload: writing 50M keys of 1KB value for a total of 50GB. This workload represents the population of the database.
2. Read-only workload: reads 10% of the dataset.
3. Benchmark: mixed reads and writes workload, with a 50-50 read:write ratio.
4. Database replication: reading 50GB from the master and writing them on the slave.

Using these workloads, we accept a specific optimization iff it improves the performance of workload 4 *and* does not degrade the performance of workloads 1-3.

The first step of the optimization process consists of identifying the bottleneck. Since the database replication is primarily composed of reads on the master and writes on the slave, we analyzed which of these two operations was longer. We run two experiments: (1) an experiment where all data on the master is stored in RAM, so drive accesses can only happen as a result of writes on the slave (this experiment is called pure-ram master), and (2) an experiment where all the data on the slave is stored in RAM, so drive accesses can only happen as a result of read on the master (this experiment is called pure-ram slave). Comparing the duration time of these two experiments allowed us to work on the optimization of the longest of the two in order to reach a shorter replication time.

We also analyze the activity on the servers: we examine the runtime, throughput, and latency following each tuning. We also monitor multiple system metrics: Redis and RocksDB thread loads, the I/O statistics, the RocksDB level statistics, the amplifications factors, the slowdowns, and the write stalls. These metrics (described throughout the paper) help us measure side effects of the optimizations evaluated, and opt-out tunings that caused performance degradation (see Section 4.2).

Hardware: We run our experiments on Amazon (EC2) and Google (GCE) clouds. EC2 is the most widely used cloud [13]. We use powerful instances of type I2.8xLarge with 32 vCPUs, 244GB of RAM, 8 x 800GB SSD, and a 10GB Network with Enhanced Network Interface (ENI) enabled by default. Additionally, we ran experiments on GCE to take advantage of the NVMe drives which are currently not available on EC2. Similarly, we use powerful instances with 32 vCPUs of type Intel Xeon 2.20GHz, 208GB of RAM, and 8 x 375GB NVMe.

4. EXPERIMENTS AND RESULTS

This section describes our experiments and findings during the optimization process. First, in Section 4.1, we walk through our experiments and detail both the parameters that significantly improved performance as well as tunings that seemed promising but had a negative effect. These experiments were run on Amazon EC2. Then, in Section 4.3, we repeat the tuning process on Google GCE using NVMe drives.

Throughout this section, words in **bold font** highlight specific RocksDB tuning experiments, and words in `typewriter font` specify a RocksDB knob name. A summarizing list of all tuning experiments, knob names, and their corresponding performance impact is given in Table 1.

4.1 Experiments on EC2 with SATA SSD

Our RocksDB baseline configuration uses the stock RocksDB version 4.9 with several non-default parameter settings listed in Table 2. These were changed during the initial setting stage, and are considered part of the baseline. Since Redis on Flash uses RAM for caching we disable the RocksDB cache (there is no need to cache a value twice). We also disable OS buffers to make sure writes go to the drives rather than stay in memory buffers. A more-significant change is the disabling of Write-Ahead-Logging (WAL) with `rocksdb_writeoptions_disable_WAL` : WAL has a cost and no use in our scenario, because Redis on Flash rely on other mechanisms to provide durability.

Table 1: Parameters and their impact

Parameter	Name	Original Value	New Value	Performance Impact
Compaction threads	max_background_compactions	8	64	24%
Slowdowns	level0_slowdown_writes_trigger	12	24	10%
Stops	level0_stop_writes_trigger	20	40	10%
Compaction readahead	compaction_readahead_size	0	2MB	300%
Redis IO threads	Redis IO threads	8	64	500%
RAID chunks	chunk	512k	16k	68%
Filter for hits	optimize_filters_for_hits	0	1	7%
Bulk mode	prepare_for_bulk_mode	0	1	500%
Block size	block_size	4k	16k	60%
Synchronization	bytes_per_sec	0MB	2MB	0%

Table 2: Initial configuration

Parameter	Value
Memtable budget	128MB
Level 0 slowdown	12
Max open files	-1
WAL	Disabled
OS buffer	Disabled
Cache	Disabled

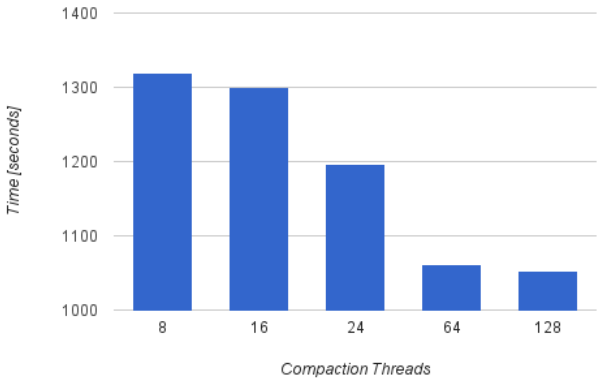

Figure 2: Increase parallelism for the compaction threads decreases the replication time

4.1.1 *Maximizing Parallelism to Remove Bottlenecks*

We started our optimization efforts by reducing the load on threads with high CPU utilization. We characterized the CPU occupancy of RocksDB background threads as well as Redis' I/O threads, and increased their parallelism when necessary to prevent them from being the bottleneck.

RocksDB backgrounds threads have two main functions – running compaction jobs, and running flush jobs (see Section 2.3). As can be seen in Figure 2, increasing the number of **compaction thread** using `max_background_compaction` from 8 to 64 improved performance by 24%. We observed that more parallelism on the compaction threads shortened the compaction cycles and provided a higher throughput on all the write workloads. We also experimented tuning the number of flush threads (which copy data from memtables to disk), but increasing their parallelism did not improve overall performance.

4.1.2 *Stabilizing Throughput and Latency*

Next, we tried to stabilize the system performance, namely reduce the server performance variance in terms of both throughput and latency. As can be seen in Figure 3, we initially observed a stop-start phenomenon of the server throughput, with a peak of 50k ops/sec lowering to less than 1k ops/sec in cycles of 10 seconds. That behavior also resulted in long tail latencies, as a small percentage of the request experienced very long response times.

As expected, the compaction cycles impact the traffic. They are triggered when the number of files in level 0 reaches its limit, i.e., `level0_slowdown_writes_trigger` (default of 12 files). When this limit is low, compaction happens too frequently, causing a disturbance to traffic as we observe in Figure 3. On the other hand, when the limit is too high, we cumulate a large compaction debt that eventually causes RocksDB to block all traffic for several seconds when it exceeds the predefined limit (`level0_stop_writes_trigger`). For RoF, it is preferable to have a slower but stable throughput for the entire duration to prevent both stop-start as well as write stalls (stops). As an example, Figure 4 shows a rather stable throughput where the ops/sec vary by only 1-2k ops/sec.

We experimented with different values for the slowdown and stop tunings. We observed a higher throughput as we increased the values, and kept on increasing them until they caused a degradation to other workloads due to the cumulated compaction debt. Results are shown in Figure 5. The optimal values were a slowdown of 24 and a stop of 40, providing us with an increased performance of 10%. Compaction triggered later (e.g., for values 40, 60, etc.) provided a faster replication but had a negative impact on the throughput of read-only workloads. With delayed compaction it takes longer to access data, as bloom filters are larger and not yet compacted.

4.1.3 *Generating More Throughput*

The **compaction readahead** knob (`compaction_readahead_size`) enables reading large chunks of compaction inputs during compaction jobs. By default RocksDB does not use readahead (0MB). We ran experiments with a 2MB compaction readahead and obtained a 3x shorter replication time. As previously mentioned, the compaction efficiency has a significant impact on RocksDB throughput.

Additionally, we obtained a better throughput by tuning the number of **Redis IO threads**. Since RocksDB API is synchronous, we use multiple Redis I/O threads to increase the I/O parallelism. We found that increasing the number of Redis I/O improves the latency of read requests, but that the increased contention on the memtables causes significant reduction in the performance of write request.

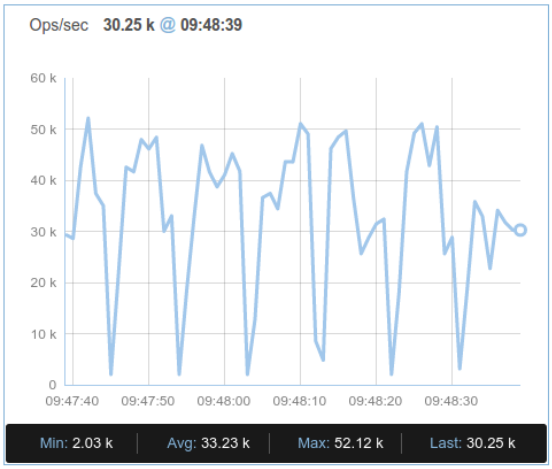

Figure 3: Slowdowns and stops limits configured too low generate a stop-start effect.

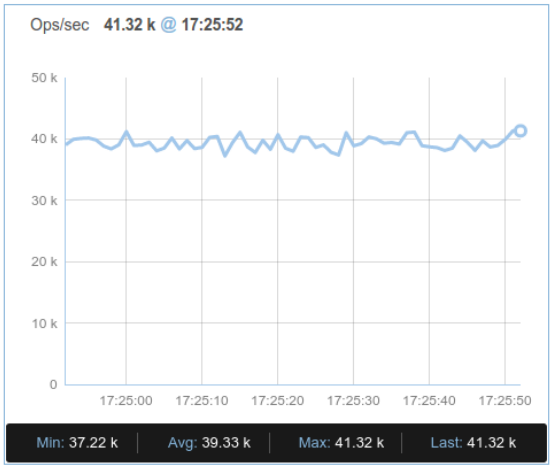

Figure 4: Stable throughput

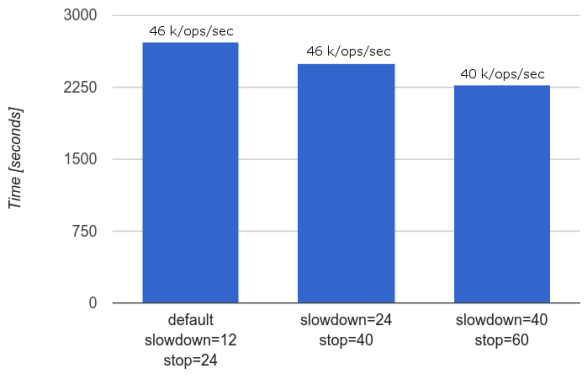

Figure 5: Experiments on slowdowns and stops tunings

Therefore, we dedicate a single I/O thread for handling writes into RocksDB, and use the rest of the I/O threads to handle the reads concurrently (a.k.a "one writer"). The improvement obtained was a factor of 5.

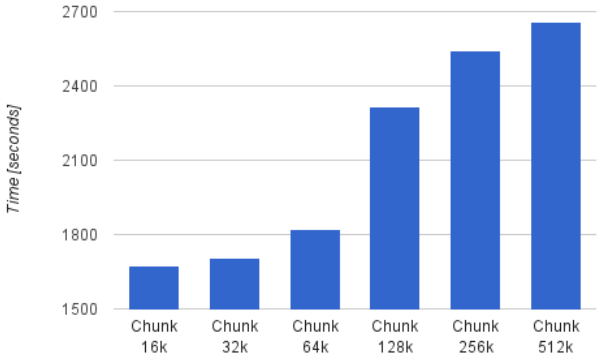

Figure 6: Runtime as a variance of RAID configuration

Since RoF deployments commonly use multiple storage drives to increase the available storage bandwidth, we also experimented with **RAID**-specific setting. RoF (and RocksDB) employ software RAID-0 to create a striped volume, and we experimented with the effect of different `chunk` sizes (stripe sizes) on overall performance. As can be seen in Figure 6, smaller RAID chunks increase the disk access parallelism and thus performance. Therefore, we changed the default from 512KB to 16KB, further improving performance by 68%. Note that while RocksDB tuning guide suggests using larger chunk sizes [14], our experience was the opposite: smaller chunk sizes gave better performance.

Lastly, since RoF keeps a map of all keys and their location in RAM, requests to RocksDB never miss: requests to unavailable keys are handled by RoF, and only requests to data that is known to be present on disk are forwarded to the RocksDB subsystem. Consequently, we enabled a designated RocksDB **bloom filter** that is specifically optimized for this use case: `optimize_filters_for_hits`. RocksDB implements bloom filters for every SST file in order to quickly check whether a given key has a copy in the file. The specific optimization removes the filters from the bottommost levels, because once a request reaches that level, the value is guaranteed to be there. This optimization had a positive impact of 7% on the replication experiments.

4.1.4 Read Speed

When replicating a database, the master sends a copy of the dataset to the slave. The master needs to read all values from either its RAM or its drives depending on where specific values are located. There are two approaches for reading the data from the drives. The first method uses RocksDB iterator to read all of RocksDB database sequentially. The second method reads only values that do not have a copy in RAM, thus reducing the amount of total data read but issuing random reads to the drive. The exact tradeoff between these two methods is mainly a functions of (1) the performance of RocksDB under these two scenarios, and (2) the disk speed for sequential reads and random reads.

Figure 7 compares these two approaches by plotting the database replication time as a function of the percentage of keys residing in flash (out of all keys in the database). A higher x-axis value means that a larger portion of the values reside in flash rather than in RAM. Since the sequential read approach always reads the entire RocksDB dataset, its replication time stays roughly constant across the experiment range (the blue line). Conversely, since the random read approach access only the necessary data on the drive, its replication time grows linearly with the percentage of values

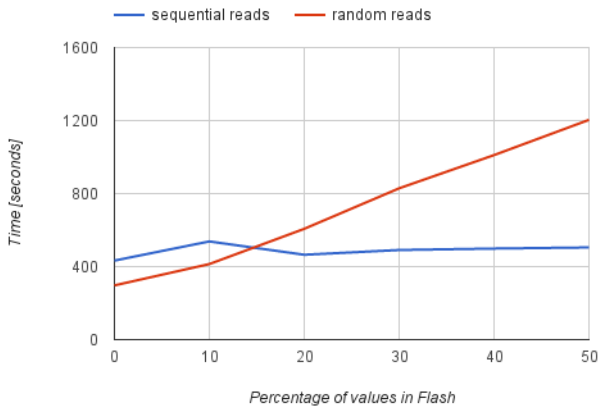

Figure 7: Comparison of replication time with random and sequential reads as a variation of the percentage of values in flash.

residing on the drives. As can be seen in the figure, the random read method is preferable only when a major portion of the data reside in RAM – 85% and up in our experiments. Sequential reads benefits from the data readahead configuration which reads larger chunks of input data and speeds up sequential reads. These results were consistent across different database sizes as well as different object sizes. Consequently, we use the sequential read approach for replication.

4.2 Negative Results

Many of the parameters we experimented with were not adopted, either because they did not improve the replication time or because they had negative side effects. In this Section we list three of these tuning efforts that either proved to be counter-intuitive, or caused surprising side effects.

One of our initial efforts involved the **bulk load mode** (`prepareforbulkload`). Enabling this mode improved replication time by 5x. However, bulk load mode has side effects that make reads prohibitively slow, because it also disables compaction. When the database is not compacted, there are many more SST files and reads take much longer to access. We tried to fix the problem by disabling the bulk mode at the end of the replication and launching a manual compaction to restore the database to a useful state, but since manual compaction is a single-threaded process, the process was taking a long time to complete: longer than the replication time. As an alternative, we launched an automatic compaction which is multithreaded, but subsequent reads were still too slow.

Surprisingly, manual compaction and automatic compaction do not have the same compaction result nor the same duration. The first one takes too long to complete, and the second one produces slow read access as it does not run a full compaction of all levels. Note that we encountered a similar compaction debt in Section 4.1.2 as compaction was delayed by the slowdown configuration. Bulk mode is composed of many different tuning parameters (13), which we evaluated separately to see if we could get the benefits while reducing the tolls. However, since disabling compaction proved to be the main reason behind the performance improvement we ended up not using this option.

Full synchronization is not mandatory in RoF because we use flash as a RAM extension (volatile). Therefore, we aimed to save the **synchronization costs** of fsync calls. We configured synchronization every 2MB (non-frequent) using `bytes_per_sync`. Surprisingly, we observed no improvement. We investigated this using the *dd* tool which writes data to disks: we compared the running time writing 50GB of generated data with and without synchronization and observed a faster performance of over 2x **without synchronization**. We were puzzled not to see a similar trend in the replication experiments. We assumed that the writes were cached and therefore we did not see the synchronization savings, but looking at *meminfo* we observed that the data was not cached. We also looked at the individual devices that compose the RAID and their IOPS, and saw they did not decrease either. Last, we confirmed that all calls to the various syncs were not taking place using strace. This counter-intuitive observation remains somewhat a mystery.

Lastly, we tuned the `block_size` parameter. Each SST file contains an index with all its blocks. Increasing the block size reduces the number of index entries for each file, but requires reading more bytes to access a value as the blocks are larger (see Section 2.3.2 on read amplification). Increasing the block size also decreases memory usage because the indices stored in RAM are larger. The default block size is 4KB. When we increased the block size to 16KB, we obtained a 60% time reduction. Unlike many other knobs which can be changed during runtime, the block size cannot be reverted and since we observed a negative impact on the reads performance (as explained above), we ended up not including this tuning.

4.3 Experiments on GCE with NVMe SSD

The experience gained tuning RocksDB on EC2 helped us plan our experiments on GCE. First, we worked on RAID optimization. We ran a sweep of experiments on the chunk size and concluded that 16KB was the optimal value with a faster replication time of 41%. The same chunk size was used for the SSD drives on EC2.

The default level 0 slowdown (12) and stop (20) delayed writes unnecessarily. As soon as we reached 12 files on level 0, a slowdown reduced the throughput significantly: from 60k to 600 operations per second, up until the compaction process completed and files moved from level N to level N+1. In order to avoid the phenomenon of stop-start on the system, we increased the slowdown to 20 and stop to 24 and observed a performance improvement of 15% (similar to EC2).

Additionally, we integrated tunings such as optimize filters for hits. We also added more RocksDB compaction threads and Redis IO threads, configured one writer to the memtables, and increased the data readahead. With all these changes cumulated, we obtained a replication time of 12 minutes and 50 seconds on GCE. Note that the same database replication size took 18 minutes on EC2 with the same RocksDB settings. However, since these are different clouds with different hardware and different storage type – SATA-SSD on EC2 and NVMe on GCE – we were expecting a faster replication time on GCE.

5. RELATED WORK

While other storage engines are available [15, 16, 17], RocksDB has a better performance and is widely used. RocskDB tuning guide [18] provides a basic configuration direction for RocksDB [10], which we used for our baseline configuration. As we show in the paper, further tuning can considerably improve performance (by 11x in our example).

Krishnamoorthy and Choi [19] experimented the tuning of RocksDB for NVMe SSDs. For the best of our knowledge, this is the closest work to the evaluation presented in this paper. However, that analysis was done in the context of stress testing RocksDB, while we optimize a full stack real-world use-case with RocksDB

being the underlining engine. Consequently, that work differs both in its tuning process and its conclusions.

Similar to Redis on Flash, several other Redis extensions use flash drives to extend the capabilities of Redis. An Intel project [20], forked from Redis, rely on the non-volatility property of flash drives to provide persistence. The motivation is to eliminate the synchronization costs of disk backups by writing to the non-volatile flash drives instead of writing to the append-only-file (AOF). Upon a node failure, the machine is repaired and brought back on with its non-volatile flash data. This approach does not work on the cloud as machines cannot be reused/repaired. Redis Naive-Disk-Store(NDS) [21] uses LMDB [17] to manage values on local drives. The data is periodically flushed, so in case of a crash, users loose *only* updates that occur since the last flush. It does not keep the key names in RAM and does not support persistence, replication, eviction, and scan.

6. CONCLUSIONS

As the number of applications using RocksDB to manage storage continues to grow, RocksDB is quickly becoming the go-to storage-engine choice. However, its flexibility and superior performance come at a cost: tuning RocksDB is a complex task, and as shown in this paper, a highly tuned configuration can outperform a basic setup by an order of magnitude.

In this paper we used Redis on Flash – a commercial extension of the popular Redis key-value store – as a practical case-study for tuning RocksDB. We described the methodology, tuning process, and specific changes to the RocksDB settings that improved RoF overall performance by more than 11x, as well as the experiments that lead to negative results or to unexpected side effects. While different use-cases will differ in the exact optimal settings, we hope that our experience will help others optimize RocksDB, both in terms of improving overall results, as well as in reducing the required development time. The performance boost from a highly-tuned RocksDB engine is definitely worth the effort.

7. REFERENCES

[1] Siying Dong. RocksDB: Key-Value Store Optimized for Flash-Based SSD. https://www.youtube.com/watch?v=xbR0epinnqo.

[2] RocksDB and LevelDB. http://rocksdb.org/blog/2016/01/29/compaction_pri.html.

[3] Paul Dix. Benchmarking LevelDB vs RocksDB. https://www.influxdata.com/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/.

[4] RocksDB users. https://github.com/facebook/rocksdb/blob/master/USERS.md.

[5] Mark Callaghan blog. http://smalldatum.blogspot.co.il/2014/07/benchmarking-leveldb-family.html, July 7, 2014.

[6] Redis on Flash documentation. https://redislabs.com/redis-enterprise-documentation/rlecflash.

[7] Redis on Flash. https://redislabs.com/rlec-flash.

[8] Redis. http://redis.io/.

[9] Redis Users. http://techstacks.io/tech/redis.

[10] RocksDB website. http://rocksdb.org/.

[11] Memtier benchmark. https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached.

[12] Redis Labs. https://redislabs.com/.

[13] Gartner. Magic Quadrant. https://www.gartner.com/doc/reprints?id=1-2G2O5FC&ct=150519.

[14] RocksDB FAQ. https://github.com/facebook/rocksdb/wiki/RocksDB-FAQ.

[15] Kyoto Cabinet. http://fallabs.com/kyotocabinet/.

[16] LevelDB. http://leveldb.org/.

[17] Lightning Memory Mapped Database. https://lmdb.readthedocs.io.

[18] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[19] Praveen Krishnamoorthy and Choi Changho. Fine-tuning RocksDB for NVMe SSD. https://www.percona.com/live/data-performance-conference-2016/sites/default/files/slides/Percona_RocksDB_v1.3.pdf.

[20] Intel. Redis with persistent memory. https://github.com/pmem/redis#redis-with-persistent-memory.

[21] Redis Naive Disk Storage. https://github.com/mpalmer/redis/blob/nds-2.6.