Exploring Cache Policies in RocksDB

Alec Gallardo
Brandeis University

Binyamin Friedman
Brandeis University

ABSTRACT

This work explores the space of configurations for the caching policy in LSM-trees, with a particular focus on Facebook's RocksDB. We examine RocksDB's exposed parameters and run a series of benchmarks to measure performance under different loads. Some interesting trends in our results motivate future work and experimentation. Ultimately, we hope to develop precise cost models that enable more workload-aware cache designs.

1 INTRODUCTION

1.1 Background

Log Structured Merge Trees (LSM-trees) are a widely used data structure intended to handle massive quantities of data. LSMs organize data across multiple levels in a log-structured format. They are vital to modern data systems deployed on write-heavy workloads, due to the use of fast, buffered ingestion which allows for competitive write performance. The key to LSM-trees' wide applicability is the use of auxiliary structures, like indexes and bloom filters, which enable efficient point queries. Interestingly, the LSM-tree's organization can be tuned in order to navigate the Pareto frontier of read versus write performance. This ability to scale for large amounts of data while remaining versatile is vitally important in dealing with the rapidly increasing production of data in our modern world. LSM-trees have been adopted in many popular database systems, such as RocksDB [3] at Facebook, Cassandra [1], and BigTable [2] at Google.

1.2 Problem Statement

Commercial LSM engines use a cache for the index, filter, and data blocks. Due to locality, caches are a fundamental technique for making use of the memory hierarchy. Since memory is a limited resource, the decision of which blocks to cache at any given time is crucial to lookup performance, since blocks found in cache can be queried in a negligible amount of time. The space of possible algorithms is enormous, but difficult to reason about.

In practice, commercial systems implement simple, generally effective cache policies like Least Recently Used (LRU) and CLOCK, typically assigning metadata (index and filter) blocks with a higher priority. We envision a caching policy that can generalize to different workload types, either through automated tuning or through configuration done with advanced knowledge of the required workload. In this paper, we explore cache policies available through RocksDB and benchmark the space of parameters. We hope this gives direction to future research on more resilient caching policies.

1.3 Related Works

Previous works on caching policies in LSM-trees have examined a couple key areas of improvement. Some papers focus on making caches adjust their methodologies to service different workloads. For example, AC-Key [5] uses smart heuristics to lazily adjust the sizes of its key-value, key-pointer, and block caches towards an equilibrium. The authors show how each of these components can service different query types, and use a smart cost model to adjust them automatically.

Another subject of research delves into mitigating the negative effects of compaction on block cache performance. Compactions invalidate a large amount of blocks, which typically increases the number of cache misses dramatically until the cache refills. Leaper: A Learned Prefetcher for Cache Invalidation [6] explores the use of sampling and machine learning to enable smart prefetching during compactions. The scheme is quite complex but ultimately mitigates latency spikes very effectively. LSbM-tree [4] also focuses on the latency spikes that arise during LSM compactions, however they introduce a secondary LSM-tree to store hot values.

Across these works, the key takeaway is resilience. The block cache is a fundamental part of modern data system performance, so a resilient data system requires a resilient caching policy.

2 ROCKSDB'S CACHE

RocksDB provides support for two distinct cache implementations.

(1) **The LRU Cache** is the standard option, which implements a basic LRU replacement policy. This implementation allows for multiple priorities, which can be used to distinguish index and filter blocks (and blob blocks in the context of BlobDB).
(2) **The HyperClock Cache** is an experimental implementation which is not yet production tested. This implementation uses aging variant of CLOCK eviction, and is purported to run better under parallel load. However, the HyperClock cache also exposes additional tuning parameters.

Both implementations are sharded according to a number of shard bits, which allows for concurrent access to the cache without mutex contention. For our experiments, we choose the better supported LRU cache, which is also RocksDB's default and currently recommended choice.

RocksDB provides many configuration option dictating how it works with the cache. By default, RocksDB stores index and filter blocks outside of the block cache in main memory, which removes control from the user. We choose to experiment with index and filter blocks in the block cache to allow for fine-grained control over the policy. While RocksDB has support for multilevel "partitioned" indexes, we focus our experiments on unpartitioned indexes. Finally, we leave the cache sharded by default.

In our experiments, we experiment with the choice to store metadata with high priority, whether to pin metadata, the block cache capacity, and the high priority ratio. RocksDB allows for pinning no metadata, pinning metadata tables that may have originated from a flush, and pinning all metadata. The high priority ratio setting controls the size of the high priority LRU list. A summary of these is displayed in Table 1.

Options	Values		
Cache metadata	In block cache	Separately	
Cache high priority	Yes	No	
Pin metadata	None	Flushed and similar	All
Cache size	0-110% of dataset		
High priority ratio	0-100% of cache		
Shard bits	≥0 bits		

Table 1: RocksDB Cache Configuration

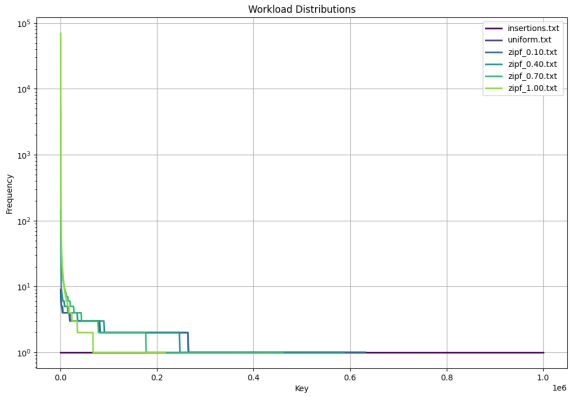

Figure 1: A higher α corresponds to a higher skew.

3 EXPERIMENTS

3.1 Workloads

We use SSD-Brandeis' KV-WorkloadGenerator to generate synthetic workloads for different key distributions. Our insertion workload consists of 1,000,000 random key-value pairs. An entry is 128 bytes and the key size is 8 bytes. We generate the following workloads for testing point-lookup query performance,

(1) **Uniform:** 1,000,000 random uniform point queries.
(2) **Zipfian:** We query from a Zipfian distribution with $\alpha \in [0.1, 0.2, 0.3, ..., 0.9, 1.0]$.

See Figure 1 for a visualization of our workload distributions.

3.2 The Experimental Setup

We forked and refactored SSD-Brandeis' RocksDB-Wrapper which includes RocksDB, a workload generator, and some utilities as submodules. View our repository on GitHub.

Our testing framework runs RocksDB through a Python wrapper. Our wrapper automatically generates our desired workloads and runs them through the C++ RocksDB wrapper. We pipe in RocksDB's console output in order to track workload progress on a fine level, and parse the output statistics into an easily readable JSON format. This allows for more expressive control over our experiments and analytics, while leaving all performance critical tasks to the compiled C++ executables.

3.3 Experiments

We conduct four experiments to capture the relationship between block cache size, workload skew, and metadata priority.

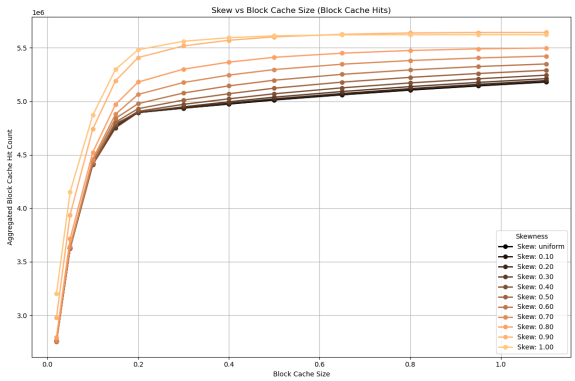

Figure 2: As expected, both a larger skew and a larger cache size improve the efficacy of the cache.

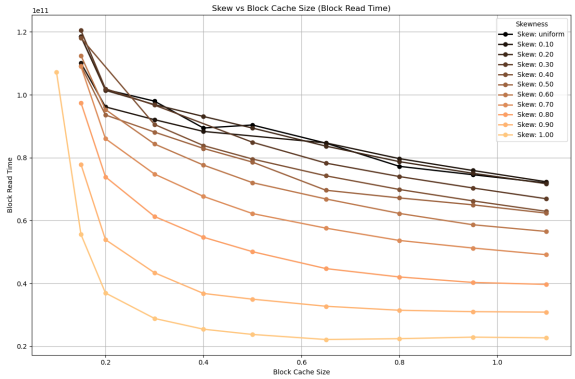

Figure 3: Total read time functions as an inverse of the number of cache hits.

(1) **Cache Size against Skew:** This experiment varies cache size against different levels of skew. We test sizes in $[0.02, 0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.65, 0.8, 0.95, 1.1]$. Note that RocksDB does not allow for a dynamically sizing cache. Instead we set the sizes as a percentage of the total size of our ten million entries. We test a size larger than 1 because the index and filters add to the number of pages. For all configurations, we set pinning to none, priority ratio to 0.5, and cache metadata with high priority. See Figures 2 and 3. The overall trend of our results is predictable; a higher workload skew increases the number of block cache hits which lowers the overall read time. Interestingly, the benefits of the block cache largely round off after a size of 0.2. This is confusing, because one would expect the benefit to continue rising dramatically until all pages can fit into memory. These results suggest the effect of another performance factor.
(2) **Pinning Policy against Cache Size:** This experiment varies the pinning policy against a subset of our list of cache sizes. For all configurations, we pin metadata with high priority and set the priority ratio to 0.5. We use a moderately skewed workload with $\alpha = 0.30$. See Figure 4. Note that data points are missing on the pin all policy for smaller cache sizes

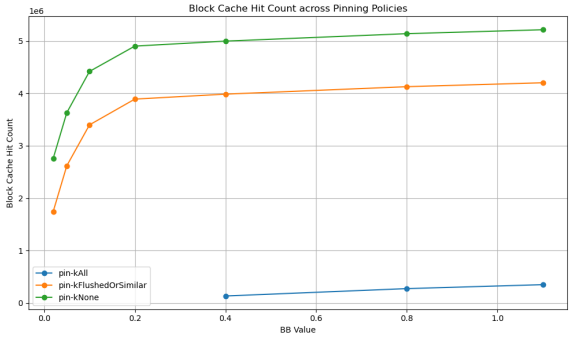

Figure 4: Pinning less results in more hits.

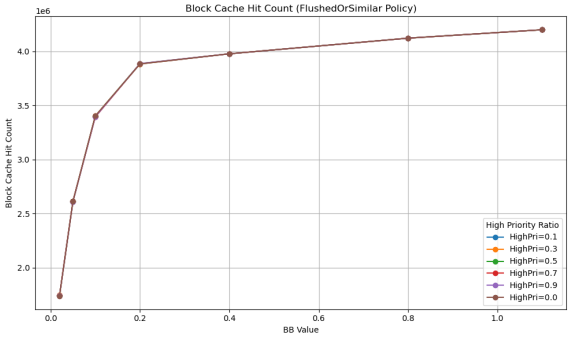

Figure 6: Varying priority ratio has no effect on performance.

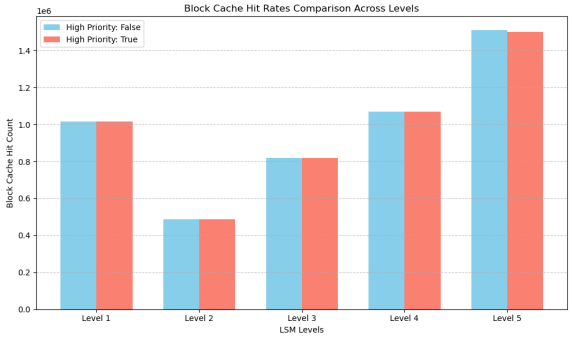

Figure 5: Metadata priority has no effect.

because RocksDB cannot fit all the metadata into the block cache. Counter to our intuition, pinning metadata blocks seems to harm performance, resulting in less cache hits. We also test this for uniform and $\alpha = 1.00$ workloads and find the same result.

(3) **Caching Metadata with High Priority:** This experiment attempts to measure the effects of caching metadata with a higher policy, which takes advantage of RocksDB's high priority LRU list. We set pinning to none, keep the priority ratio at 0.5, and set the cache size to 0.2. We use a Zipfian workload with $\alpha = 0.30$. Figure 5 shows that this option does not affect performance in a measurable way. We continue to experiment with the priority ratio below.

(4) **Adjusting the Priority Ratio:** We pin metadata according to the "flushed and similar" policy, keep the parameters as listed above, but vary the LRU priority ratio. Figure 6 shows that this parameter also does not affect performance in a measurable way. We strongly expected the priority settings to have an effect, so these results are very surprising. We hypothesize this could be due to an implementation issue in RocksDB.

4 CONCLUSION

Our main takeaway is simple; access patterns are extremely important to the block cache's performance. Higher levels of skew greatly increased the performance of the system, no matter the size of the block cache. However, our research mostly leaves unanswered questions about the behavior of RocksDB's cache. Contrary to our expectations, pinning appeared to strictly harm performance, even though it seems to be a smarter solution than plain LRU. In addition, the use of the high priority LRU list had zero effect on performance. We posit this is due to an implementation mistake on RocksDB's end, but any definite conclusion requires a more detailed examination. Ultimately, RocksDB does not expose its space of cache configurations in a transparent way. We believe data systems ought to make clear the precise effects of their various parameters.

4.1 Next Steps

We'd like to more satisfactorily answer our questions about RocksDB, both regarding the lack of effect of the high priority LRU list and the diminishing returns of cache size.

As shown in the results earlier, configuring the high priority list did not have a discernible effect on block cache hits or block read time. However, we can see in Table 2 that caching metadata blocks with high priority has a slight impact on the number of metadata reads. However, this change does not lead to a measurable difference in the total number of cache hits.

We also would like to further investigate the diminishing returns of cache size on hits. Intuitively, one would think that at least in a uniform workload, the number of hits would scale linearly with the size of the block cache. We do not see this relation in our data, so we suspect that another factor is affecting performance. More investigation is needed.

Other avenues of research that to explore caching policies are exploring the HyperClock Cache; within this paper, we chose to focus on the LRU Cache as it is the standard option. However, HyperClock provides additional tuning parameters that could be tested for performance.

Additionally, we would like to test additional workloads, we focused largely on point queries do to the limitations of our computational hardware and time constraints, but believe it would provide valuable information to test more diverse workloads going forward.

REFERENCES

[1] Apache. [n. d.]. Cassandra. https://cassandra.apache.org
[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008.

Metric	Regular Priority	High Priority
Index Hit Count	1,026,500	1,021,094
Index Block Read Count	703	6,109
Filter Hit Count	3,834,346	3,831,024
Filter Block Read Count	453	3,774

Table 2: Comparison of metrics between caching metadata blocks with high priority and regular priority.

Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[3] Facebook. [n. d.]. RocksDB. https://github.com/facebook/rocksdb

[4] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 68–79. https://doi.org/10.1109/ICDCS.2017.70

[5] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David Hung-Chang Du. 2020. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:220836392

[6] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proc. VLDB Endow.* 13, 12 (July 2020), 1976–1989. https://doi.org/10.14778/3407790.3407803