Quantitative Analysis of Compaction Policies in a Key-Value Store

Inho Song
Dept. of Computer Science
Dankook University
Yongin, Korea
inhoinno@dankook.ac.kr

Yejin Han
Dept. of Computer Science
Dankook University
Yongin, Korea
hyj0225@dankook.ac.kr

Hojin Shin
Dept. of Computer Science
Dankook University
Yongin, Korea
hojin03s@dankook.ac.kr

Seehwan Yoo
Dept. of Computer Science
Dankook University
Yongin, Korea
seehwan.yoo@dankook.ac.kr

Jongmoo Choi
Dept. of Computer Science
Dankook University
Yongin, Korea
choijm@dankook.ac.kr

Yoojin Chung
Div. of Computer and ES Engineering
Hankuk University of Foreign Studies
Yongin, Korea
chungyj@hufs.ac.kr

Abstract— **Compaction is an essential ingredient in a LSM (Log-Structured Merge)-tree based key-value store. In this paper, we analyze two representative compaction policies, called *leveled* and *universal*, using RocksDB. Our analysis uncovers that the universal policy has a capability to reduce write amplification by applying compaction in a lazy manner. However, the lazy manner deteriorates space amplification, which leads to adverse effects such as a relatively longer period of low performance for compaction and degraded throughput for range query. We also observe that, for sequential access pattern, the leveled policy can provide better write amplification than the universal policy by employing a technique called *trivial move*. In addition, we find out that the background compaction and index cache give a substantial impact on the performance of point query. Our analysis reveals tradeoffs between two policies based on various aspects including access pattern, query type, and configurations, which can be used effectively for designing new and hybrid compaction policies.**

Keywords— Big Data, Key-Value Store, LSM-tree, Compaction, Evaluation

I. INTRODUCTION

A key-value store is a well-known storage engine, becoming a de-facto standard database for Big data analysis. Many companies have their own key-value store solutions such as Google's LevelDB [1], Facebook's RocksDB [2], Amazon's Dynamo [3], Microsoft's DocumentDB [4], and so on. In addition, several outstanding studies have been proposed in academia including Wisckey [5], PebblesDB [6], SILK [7] and SpanDB [8].

Most key-value stores make use of the LSM (Log-Structured Merge)-tree [9]. It is a write-optimized data structure that converts random writes into sequential ones by updating data in a log-structured manner. It also employs a mechanism, called *compaction*, to reclaim the space occupied by obsolete data and to merge remaining data in a sorted order for fast lookup. Compaction plays a vital role, influencing throughput and latency of a key-value store greatly [2, 6, 7].

Two representative compaction policies are leveled and universal (also known as tiered). The key difference between two policies is the eagerness to compaction. The leveled policy tries to do compaction aggressively whereas the universal policy do compaction in a lazy manner. This difference yields various tradeoffs in the viewpoint of write, space and read amplification.

In this study, we investigate these policies quantitatively using RocksDB under diverse workloads and make the following four observations. First, the universal policy provides better write amplification than the leveled policy, eventually resulting in enhanced throughput. However, at the cost of better write amplification, the universal policy suffers from higher space amplification that causes relatively longer period of low performance for compaction.

The second observation is that the leveled policy is relatively easier to exploit a technique, called *trivial move*, than the universal policy, which gives an opportunity to reduce write amplification, especially for sequential write workload. Third, the higher space amplification of the universal policy leads to higher read amplification, which in turn deteriorates the performance of range query. The fourth observation is that the lookup performance heavily relies on not only the compaction policies but also other configurations such as background compaction and index cache.

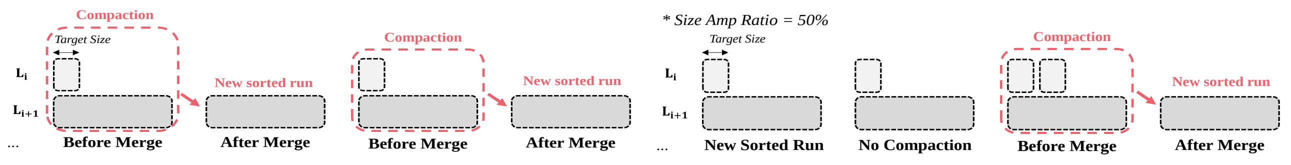
Our observations expose that, in general, the universal policy provides better performance for write-intensive workloads while the leveled policy outperforms for the read-intensive workloads. However, both policies have room for improvement by utilizing timely background compaction, trivial move, and secondary index for range query. This study also reveals the need for an adaptive scheme that integrates both merits of two policies and adjusts itself dynamically according to workload characteristics.

The rest of this paper is organized as follows. In Section 2, we explain the background of this study including RocksDB internals and two compaction policies. Then, we discuss our methodology in Section 3. Analysis results are given in Section 4. Finally, we present the conclusion and future work in Section 5.

II. BACKGROUND

A. RocksDB internals

RocksDB has two core components, Memtable in main memory and SSTable in storage [10, 11]. Memtable is a kind of a write buffer where a newly inserted or updated key-value pair is managed using the Skiplist data structure. When the



(a) Leveled compaction

(b) Universal compaction

Fig. 1. Conceptual comparison between the leveled and universal policy.

size of a Memtable instance becomes larger than a threshold (*write_buffer_size* in the RocksDB's nomenclature), it is written into storage, becoming a new SSTable file.

As more key-value pairs are put, there exist many SSTables that are managed on multiple levels, called L_0, L_1, .. and L_{max}. When a Memtable instance is written into storage, it goes to L_0, the top-most level. When the number of SSTables in L_0 becomes above a threshold, some or all SSTables are merged into the below level, that is L_1. This operation is called as *compaction*. Note that, in LSM-tree, more recent data is always located in the above level. Hence, the most recent SSTable is located in L_0 while the oldest SSTable is located in L_{max}. Also, all key-value pairs in a level are in a sorted order, enabling to fast key lookup using binary search.

B. *Compaction policies*

RocksDB supports three compaction policies, namely leveled, universal and FIFO [10, 11]. Since the FIFO policy is a special case for a key-value cache usage scenario, this study focuses on the remaining two policies.

Leveled compaction policy. This policy always triggers compaction when the size of a level is larger than a threshold, called *target_size* [10], as illustrated in Fig. 1(a). Assume that new key-value pairs are added in L_i and its size becomes larger than its target_size. Then, compaction is triggered and SSTables in L_i and L_{i+1} that have overlapped key range are merged into L_{i+1}, becoming new SSTables. Compaction is a heavy operation, consisting of reading SSTables from storage, performing merge-sort, and writing new SSTables to storage.

In Fig. 1(a), we refer to all SSTables newly merged in L_{i+1} as a new sorted run since all key-value pairs are managed in a sorted order. Actually in the leveled compaction, each level except L_0 is a sorted run that is partitioned into multiple SSTables. SSTables in L_0 is a special case, written from Memtable, where each SSTable can be considered as a sorted run. Hence, SSTables in L_0 can have overlapping key ranges.

When compaction is triggered L_i and there are multiple SSTables that can be picked as a candidate, the leveled policy utilizes various selection rules such as age or least cost. If it selects a SSTable that does not overlap with other SSTables in L_i and L_{i+1}, that table can be moved without paying actual compaction overhead. This technique is called as *trivial move*.

Universal Compaction. This policy is an RocksDB's implementation of the tiered compaction policy. The tiered policy allows to have multiple sorted runs in a level [12]. It merges SSTables just from L_i and makes a new sorted run in L_{i+1}. Therefore, there exist multiple runs in L_{i+1}. This approach guarantees the per-level write amplification as 1 meaning that the number of writes between adjacent levels is bound to 1.

The universal policy, RocksDB's implementation, is a bit different form the classic tired policy. It makes use of various parameters, including size amplification ratio, individual size ratio, number of sorted runs, and age of data [10]. But, in this paper, we discuss the size amplification ratio only for simplicity, as shown in *Fig. 1*(b). Assume that a new run is added in L_i. If the size amplification ratio is becomes larger than a threshold, all runs both in L_i and L_{i+1} are merged into a new run. Note that this new run can be in L_i or L_{i+1} according to the previous state of levels. In the universal policy, levels are rather logical concept used for managing SSTables together in a sorted run.

The selection rule used by the universal policy is rather simple, choosing the smallest run. Then, it extends runs with the consecutive time ranges for compaction. This rule makes it relatively hard to employ the trivial move technique in the universal policy.

III. METHODOLOGY

In this section, we describe our methodology regarding how to compare two policies. We start from three amplifications. Then, we discuss workload-aware metrics and our implementation for monitoring in RocksDB.

A. *Three amplfications*

Write amplification. Write amplification is a core metric to measure the performance of a LSM-tree based key-value store. It occurs while compaction is performed multiple times. Write amplification impacts the throughput of a key-value store by consuming storage bandwidth for compaction. It also worsens user latency by interfering lookup requests. In addition, it may hurt the reliability and lifetime of the flash-based storage since flash memory has an endurance issue [13]. We monitor bytes written to storage and bytes put to RocksDB for estimating write amplification.

Read amplification. Read amplification is defined as the ratio between the amount of data read to storage and the amount of data requested by user. This amplification is caused by several sources, 1) for compaction, 2) for reading data in a disk block unit, not a key-value pair alone, and 3) due to the false-positive effect of the Bloom filter mechanism [11]. In addition, it is influenced by block cache, used for buffering recently referenced disk blocks. We again monitor both bytes read from storage and gotten by users.

Space amplification. Space amplification was not a big consideration among three amplifications in past especially when we use HDD (Hard Disk Drive) as storage. However, in SSD (Solid State Drive) era, space amplification becomes serious because the performance of SSD relies on its utilization [13] and becomes primary bottleneck under typical production workloads at Facebook [14]. We monitor the amount of obsolete data that has not been reclaimed.

B. *Write intensive workloads*

For comparison under write intensive workloads, we make use of three metrics, write amplification, throughput and

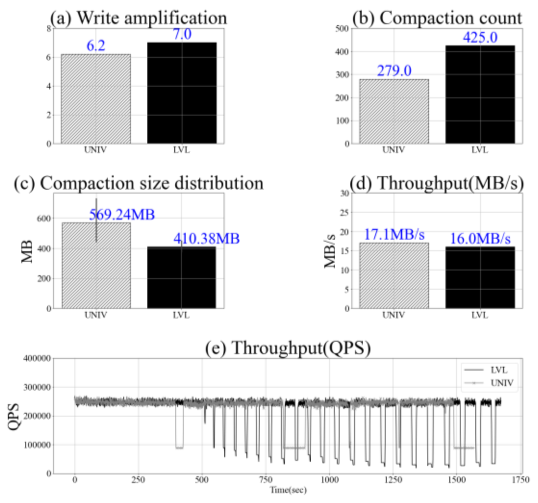


Fig. 2. Write amplification(a), Compaction Overhead(b,c), Throughput (d,e) of the leveled and universal policy for random write workload (*db_bench fillrandom* with 25GB).

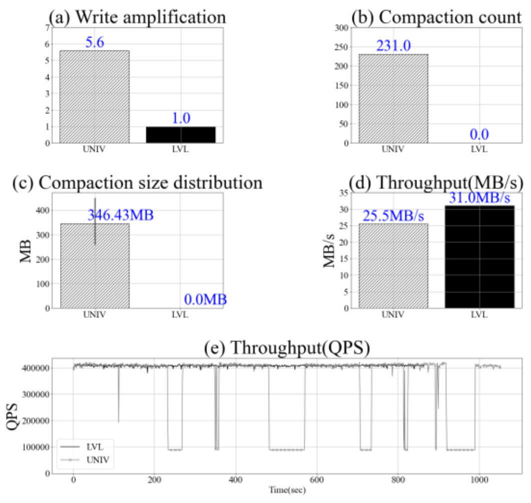


Fig. 3. Write amplification(a), Compaction Overhead(b,c), Throughput (d,e) of the leveled and universal policy for sequential write workload (*db_bench fillseq* with 25GB).

compaction overhead. RocksDB already reports two throughput values, MB/s and QPS (Queries per second).

Compaction overhead. To assess compaction overhead, we measure the total number of compactions while running a workload and the size of each compaction. The number and size shows distinct behaviors between two policies, which will be further discussed in Section 4.

C. *Read intensive workloads*

Unlike write intensive workloads, read intensive workloads are influenced by the block cache. So, not only the read amplification and throughput, but also cache hit ratio and latency are used as metrics for these workloads.

Latency and Hit Ratio. We observe the read latency distribution and measure the 99^{th} percentile long-tail latency. This metric measures the time required to complete a *get()* request. For cache hit ratio, we measure the hit/miss counts of the block cache that manages recently accessed key-value pairs using the LRU policy. In this study, we intentionally disable the index cache to analyze the effect of the block cache more clearly.

IV. Analysis

TABLE I. Experiment environment

Category	Specification
CPU	Intel(R) Core(TM) i7-10700K, 3.80GHz
Memory	32GB
Storage	Samsung 860 PRO 1TB SSD (SATA Interface)
Kernel, File System	Linux 5.7.7, Ext4
LSM-tree	RocksDB 6.21ver, Options are uploaded to github.com/DKU-StarLab/QACPs

This section discusses the comparison results between the leveled and universal policy. We discuss write intensive workloads under two access patterns, sequential and random and read intensive workload results under two query types, range and point query. Table 1 summarizes our experimental environment.

A. *Write intensive workloads*

Fig. 2 shows the write amplification, compaction overhead (count and size distribution), and throughput (MB/s and OPS) under the random access pattern. Fig. 3 are results under the sequential access pattern. From these figures, we can find that,

Observation #1 : ***The universal policy shows better write amplification than the leveled policy under random write workloads.*** In Fig. 2(a), we identify that the universal policy reduce write amplification, compared with the leveled policy, which results in better throughput in Fig. 2(d). Fig. 2(b) displays that the compaction count of the universal policy is less than the leveled policy since it can delay compaction. However, the average compaction size is larger (Fig. 2(c)) since it merges more sorted runs than the leveled policy. This larger size causes a relatively longer period of performance drop of the universal policy as shown in Fig. 2(e). The size amplification presented in Fig. 4 also demonstrates that the universal policy can delay compaction while consumes larger space, which incurs larger compaction size.

Observation #2 : ***The leveled policy can reduce write amplification under sequential write workloads.*** In general the universal policy shows better write amplification. However, for the sequential access pattern shown in Fig. 3, the leveled policy supports better write amplification by exploiting the trivial move technique. In current implementation, the universal policy does not enable this technique. We think this is due to the philosophy of the universal policy that it merges runs with the consecutive time ranges. The trivial move technique may violate this philosophy. This result also uncovers that there is room for improvement of the universal policy, especially for the sequential or non-overlapping key ranges access pattern. If we turn on the technique, the universal policy shows comparable write amplifications.

B. *Read intensive workloads*

Fig. 5 and 6 show the results under range query and point query, respectively. From these figures, we can find that,

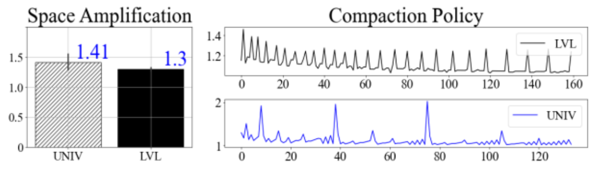


Fig. 4. Space amplification between leveled and universal policy.

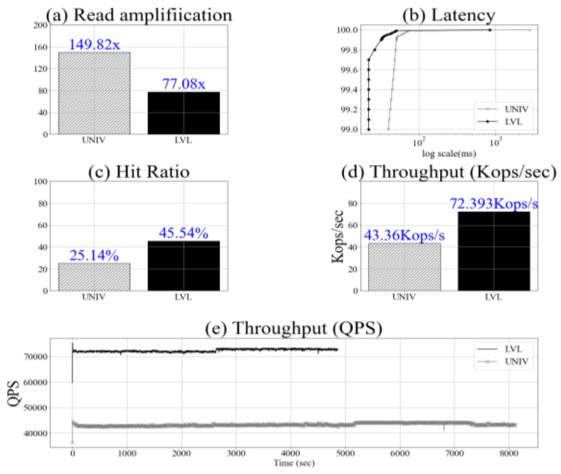


Fig. 5. Read amplification(a), Latency(b), Hit Ratio(c), Throughput(d,e) of the leveled and universal policy for range query workload (*db_bench seekrandom* with 10GB and *–keys_per_prefix=8).*

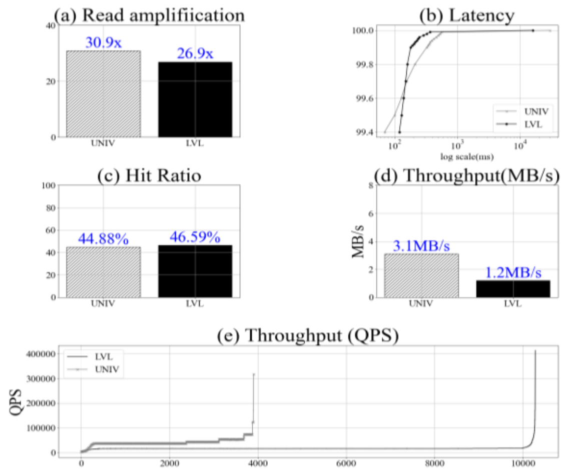


Fig. 6. Read amplification(a), Latency(b), Hit Ratio(c), Throughput(d,e) of the leveled and universal policy for point lookup query workload (*db_bench readrandom* with 10GB and *–keys_per_prefix=8).*

Observation #3 : ***The universal policy shows worse read amplification than the leveled policy, both for range and point query.*** As shown in Fig. 5(a) and Fig 6(a) the universal policy shows higher read amplification because it delays compaction, leading to many uncompacted SSTables to search. Range query especially shows terrible performance. We carefully argue that, for range query intensive workloads, we need to choose the leveled policy or at least have a method that converts from the universal to leveled dynamically according to workloads characteristics.

Observation #4 : ***The leveled policy sometimes shows poor performance especially when there are many SSTables in L_0.*** In general, better read amplification leads to better throughput, as demonstrated in Fig. 5(a) and Fig. 5(d). However, there are some cases for point query that the leveled policy performs worse even though it shows better read amplification, as shown in Fig 6(a) and Fig 6(d). Our sensitivity analysis reveals that these cases happen when there are many SSTables in L_0 since point query has to look up all these SSTables, which yielding long tail latency (Fig. 6(b)). Many SSTables in L_0 can occur when the compaction between L_0 and L_1 does not work smoothly (partly due to the shortage of compaction threads or untimely scheduling). Note that this case hardly occur in the universal policy because it always limits the total number of runs below a threshold. These results expose that the leveled policy can be enhanced by applying timely background compaction.

V. Conclusion

This paper investigates two compaction policies. Generally, the universal policy can reduce write amplification, resulting in better throughput for write intensive workloads. In contrast, the leveled policy outperforms for the read intensive, especially range query workloads. Our study also uncovers various optimization methods such as trivial move for the universal policy, timely background compaction for the leveled policy, and hybrid scheme that embraces the merits of both policies adaptively. We are currently implementing these methods and comparing them with existing hybrid proposals such as Leveled-N [10] and Fluid LSM [15].

Acknowledgement

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01475, (SW StarLab) Development of Novel Key-Value DB for Unstructured Bigdata) and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (NRF-2020R1A2C1006331)

References

[1] S. Ghemawat and J. Dean, “A fast key-value storage library written at Google”, https://github.com/google/leveldb.

[2] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du. “Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook”, In USENIX FAST, 2020.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon’s highly available key-value store", In ACM SOSP, 2007.

[4] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, R. Ferreira, M. Nassar, M. Koltachev, J. Huang, S. Sengupta, J. Levandoski, and D. Lomet, "Schema-agnostic indexing with Azure DocumentDB", In VLDB, 2015.

[5] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage", In USENIX FAST, 2016.

[6] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees", In ACM SOSP, 2017.

[7] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing latency spikes in log-structured merge key-value stores", In USENIX ATC, 2019.

[8] H. Chen, C. Ruan, C. Li, X. Ma and Y. Xu, "SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage", In USENIX FAST, 2021.

[9] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree”, Acta Informatica, 33, 1996.

[10] RocksDB Wiki, https://github.com/facebook/rocksdb/wiki

[11] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, “Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience”, In USENIX FAST, 2021.

[12] N. Dayan, M. Athanassoulis, and S. Idreos, “Monkey: Optimal Navigable Key-Value Store,” In ACM SIGMOD, 2017.

[13] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage." In USENIX FAST, 2015.

[14] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strumm. "Optimizing Space Amplification in RocksDB." In CIDR, 2017.

[15] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging.", ACM SIGMOD, 2018.