Advocating for Key-Value Stores with Workload Pattern Aware Dynamic Compaction

Heejin Yoon, Jin Yang, Juyoung Bang, Sam H. Noh*, Young-ri Choi
UNIST, *Virginia Tech
{heejin5178, yangjin, juyoungbang, ychoi}@unist.ac.kr, *samhnoh@vt.edu

Abstract

In real life, the ratio of write and read operations of key-value (KV) store workloads usually changes over time. In this paper, we present a *Dynamic wOrkload Pattern Aware* LSM-based KV store (DOPA-DB), which supports dynamic compaction strategies depending on the workload pattern. In particular, DOPA-DB is a tiered LSM-based KV store with multiple key ranges, which enables varying compaction sizes. For write-intensive workloads, DOPA-DB can minimize write stalls while minimizing compaction overhead, and for read-intensive workloads, it can aggressively perform compaction to reduce the number of file accesses. Our preliminary experimental results show the potential benefits of dynamic compaction and provide insight into research directions for dynamic compaction strategies.

CCS Concepts: • **Information systems → Database management system engines**.

Keywords: key-value store, log-structured merge-tree, dynamic workload

ACM Reference Format:

Heejin Yoon, Jin Yang, Juyoung Bang, Sam H. Noh*, Young-ri Choi. 2024. Advocating for Key-Value Stores with Workload Pattern Aware Dynamic Compaction. In *16th ACM Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE '24), July 8–9, 2024, Santa Clara, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3655038.3665955

1 Introduction

A Log-Structured Merge-tree based KV store (LSM-based KV store) has been popularly used for various data-intensive applications due to high write performance [4, 5, 10, 16, 18]. An LSM-based KV store is composed of a buffer in memory and multiple levels of files in disks. To attain high write throughput, it buffers key and value (KV) pairs and then writes them sequentially to a disk. Once these KV pairs are stored as a file, they are compacted between neighboring levels multiple times so that all the inserted KV pairs are sorted leading to high read performance. Also, write stalls are used as a means to limit the speed of write operations to improve read performance [1, 4, 5].

In real life, workload patterns regarding the ratio of write and read operations change over time [9, 17]. For example, sensing related data can be inserted and searched with some periodic patterns such as diurnality [7, 15]. Depending on the workload pattern, a different compaction strategy will be effective in improving the performance of an LSM-based KV store. For a write-intensive workload pattern, writing as much data to the disk (i.e., Level 0 (L_0) in the LSM-based KV store) as fast as possible is crucial. Therefore, compaction should be minimal but, at the same time, write stalls should be minimized. On the other hand, for a read-intensive workload pattern, reducing the number of files to be accessed is key. Therefore, compaction should be done aggressively to sort the inserted KV pairs into a single level or as few levels as possible. However, existing LSM-based KV stores typically use the same compaction strategy with a static compaction size without considering different workload patterns.

In this paper, we present a *Dynamic wOrkload Pattern Aware* LSM-based KV store (DOPA-DB) that supports dynamic compaction strategies depending on the workload pattern. In particular, DOPA-DB is a tiered LSM-based KV store with multiple key ranges. It provides efficient compaction, reducing write amplification, as tiered compaction avoids the merge-sort of overlapped KV pairs at two neighboring levels, while using key ranges eliminates merge-sorts between different key ranges. DOPA-DB detects the workload pattern at runtime and dynamically adapts the compaction size to reflect the current workload pattern. Our tiered structure with multiple key ranges enables different compaction sizes in a fine-grained way as the minimum compaction unit for each level is a key range, and multiple of these units can be compacted together as recommended by the Compaction Size Recommender (CSR). CSR keeps monitoring how much data can be further filled in L_0 before L_0 reaches the stall threshold as well as flush and compaction speeds, and based on this information, it dynamically decides the compaction size. This size will be estimated to be the largest compaction size such that a write stall does not occur due to this compaction. Thus, this size will tend to be smaller for

This work is licensed under a Creative Commons Attribution-NoDerivs International 4.0 License.
HOTSTORAGE '24, July 8–9, 2024, Santa Clara, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0630-1/24/07.
https://doi.org/10.1145/3655038.3665955

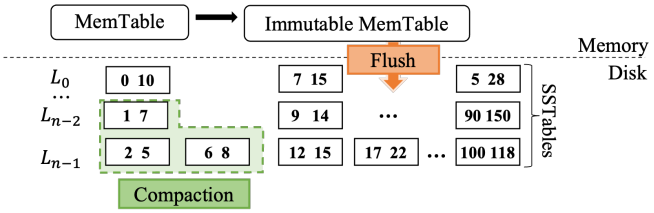


Figure 1. LSM-based KV store

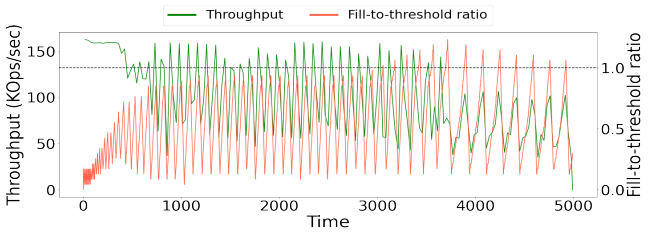


Figure 2. Throughput and fill-to-threshold ratio

more write-intensive workloads while it will be larger for more read-intensive ones.

We evaluate DOPA-DB using mixed workloads with different ratios of write and read operations. Our preliminary experimental results show the potential benefits of dynamic compaction and provide insight into research directions for dynamic compaction strategies. Instead of using static compaction as in existing LSM-based KV stores, we argue for compaction strategies that adapt the compaction size quickly as the workload pattern changes as in real-world datasets and for an LSM-based KV store with a new structure that efficiently supports fine-grained dynamic compaction sizes. We argue that this envisioned dynamic approach that considers the workload pattern can further drive performance improvements for both reads and writes in LSM-based KV stores.

2 Background and Motivation

LSM-based KV store Figure 1 shows a Log-Structured Merge-tree based KV store that consists of memory and disk components. For the memory component, the *MemTable* and *Immutable MemTable(s)* are located in DRAM, while for the disk component, *Sorted String Tables* (SSTables) are located in disks. A new key is first added to the MemTable. If the MemTable is full, it becomes an Immutable MemTable, and a new MemTable is created. For the Immutable MemTable, a background thread *flushes* it to the disk, storing KV pairs in SSTable format. The disk component has multiple levels, from L_0 to L_{n-1}, where n is the number of levels, and the size of each level is limited but the size limit increases for higher levels. To maintain the size of each level, *compaction* is leveraged. When the size of level L_i is larger than its limit, a set of SSTables in L_i are merge-sorted with SSTables in L_{i+1}, whose key ranges are overlapped with those of SSTables in L_i. This not only garbage collects old KV pairs but also improves search performance by reducing the number of files as well as non-empty levels. In the multiple levels, for the same key, a newer value is stored in a lower level.

Stalls in LSM-based KV stores An LSM-based KV store suffers from space amplification due to out-of-place updates and read amplification caused by the multi-level structure. To alleviate these issues, existing LSM-based KV stores leverage stalls to limit the speed of write operations. As discussed in ADOC [25], for RocksDB [5], there are three types of stalls, MemTable, L_0, and Pending. Among them, the L_0 stall strongly affects write performance. L_0 is a special level in LSM-based KV stores because L_0 is the only level that allows overlapped SSTable files. RocksDB incurs an L_0 stall when the size of L_0 is larger than a specific threshold, as the large L_0 leads to serious read performance degradation and space amplification if there are many update queries. In our experiment of loading a dataset composed of 500 million KV pairs, the L_0 stall contributes to 71.9% of the total write stall time. Reducing the L_0 stall time is critical for attaining high write performance.

Issues of static compactions in LSM-based KV stores In real-world workloads, the ratio of write and read operations of workloads change over time [9, 17]. Thus, naturally, depending on the workload pattern, a different compaction strategy is preferred [13, 14]. For a write-intensive workload, L_0 should take in as much writes as possible. To do so, write stalls should be avoided as write performance is severely degraded when a write stall occurs. Therefore, the compaction strategy should focus on minimizing the write stall time while also minimizing compaction overhead. Figure 2 shows the fill-to-threshold ratio, which shows how much L_0 is filled until the L_0 stall threshold, and write throughput over time, of RocksDB, for a write-intensive workload. We can see that when the fill-to-threshold ratio becomes larger than one, write throughput is degraded as a write stall occurs. These cases occur when compaction in an upper level takes so long such that L_0 - L_1 compaction, which relieves L_0, is delayed. For a read-intensive workload, the number of levels as well as the number of SSTable files that should be accessed for a read operation should be minimized as access of every level and SSTable incurs overhead. Therefore, the compaction strategy should focus on aggressively performing compaction so that inserted KV pairs are located in a single or a few upper levels to minimize file accesses. Existing LSM-based KV stores such as RocksDB, however, do not consider the dynamic workload pattern and use the same compaction strategy with a static compaction size (64MB in RocksDB by default), leading to sub-optimal write and read performance.

3 DOPA-DB

In this section, we describe DOPA-DB, a KV store that adapts the compaction size according to the workload patterns such that performance is optimized. More specifically, as shown in Figure 3, DOPA-DB is a *tiered LSM-tree based KV store* with *multiple key ranges* denoted R_i^j for level i. With the

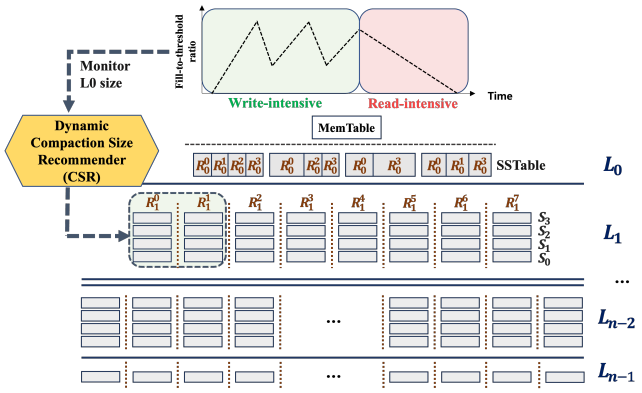


Figure 3. System Overview

tiered compaction design choice, DOPA-DB merge-sorts the KV pairs in the same level and then places them in the next level, avoiding the merge-sort of overlapped KV pairs at two neighboring levels except for the last level, while with the multiple key range design choice, compaction overhead across key ranges is eliminated as merge-sorts between key ranges do not happen.

As shown in Figure 3, DOPA-DB consists of a MemTable in DRAM and multiple levels, L_0 to L_{n-1}, in disk, where n is the number of levels. While L_0 and L_{n-1} have no sub-levels, each level L_i where $1 \leq i < n-1$ has p sub-levels, S_0 to S_{p-1}, to support tiered compaction. In each level L_i, the total key space is divided into r_i ranges such that in each sub-level, KV pairs within the same range are sorted and stored as an SSTable file or multiple SSTable files, separately from KV pairs in other ranges. Thus, SSTable files in the same sub-level never overlap with each other. (How sub-levels within a level are generated and compacted to the next level will be discussed below when DOPA-DB's tiered compaction is described.)

L_0 is set to have r_0 ranges and as level L_i gets higher, the number of ranges r_i increases based on the range amplification ratio R_{ratio}. For example, if L_0 has 4 ranges and R_{ratio} is four, then r_1, the number of ranges for L_1, is 16. As i of L_i increases, typically, r_i will increase as the level capacity also increases as it will contain more KV pairs. The values of p, r_0, and R_{ratio} need to be configured as they affect the performance of DOPA-DB. We will discuss how to configure these parameters in Section 6. Note that the key space of each key range also needs to be determined. In this work, we take a naive approach where they are set based on the keys stored in the Immutable MemTable that is initially flushed to the disk such that each key range in each level has the same number of keys, and L_i where $0 < i \leq n-1$ always inherits r_{i-1} ranges of L_{i-1}.

To effectively flush the MemTable to L_0, each SSTable file simply maintains its sorted order resulting in multiple key ranges within an SSTable, as shown in Figure 3. This flush operation is done similarly to existing LSM-based KV

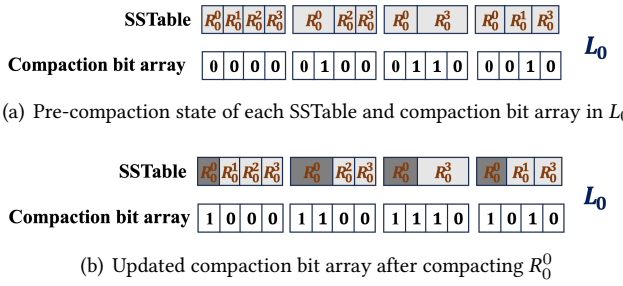


Figure 4. Compaction bit array for $L_0 - L_1$ compaction

stores. However, to deal with the key ranges, DOPA-DB maintains for each SSTable file an additional compaction bit array, which represents whether the range has been compacted with L_1 or not. More specifically, it performs L_0-L_1 compaction in key range units in a round-robin fashion one by one. For each range R_0^j, starting from $j = 0$, the compaction bit for range j of each SSTable file is checked. If the compaction bit is 0, this means the KV pairs have not been compacted to L_1 yet. Thus, compaction is performed on these KV pairs. Then, the corresponding compaction bit is set to 1 so that they will no longer be considered for compaction. An SSTable file can be deleted only when all the compaction bits are set to 1. Figure 4 shows how the compaction bit array is used for $L_0 - L_1$ compaction. Figure 4(a) shows a pre-compaction state of each SSTable and its corresponding compaction bit array, while Figure 4(b) shows the updated status after performing $L_0 - L_1$ compaction for R_0^0. After the compaction, the compaction bit for R_0^0 for each SSTable at L_0 is set to 1.

For compaction between L_i and L_{i+1} where $1 \leq i < n-1$, DOPA-DB performs compaction for key ranges in a round-robin fashion similarly to L_0 - L_1 compaction, but this time, the minimum compaction unit being all the sub-levels for a key range in a level. Multiple of these units may also be compacted together as recommended by the Compaction Size Recommender (CSR). How CSR decides on the best number of ranges to compact will be discussed below.

For performing compaction between L_i and L_{i+1} where $0 \leq i < n-2$, DOPA-DB employs tiered compaction. It partitions all the KV pairs chosen for compaction in L_i based on the key ranges of L_{i+1}, generates SSTable files accordingly, and then places each of the files in a sub-level of L_{i+1}. The exact sub-level is the one just above the sub-level that was generated during the last compaction for its key range. Figure 5 shows an example of compaction between L_1 and L_2, where two key ranges, R_1^0 and R_1^1, are chosen for compaction. The KV pairs in these ranges will be merge-sorted, and an SSTable file (or multiple SSTable files) will be generated based on the key ranges of L_2 (①). For each range R_2^j from R_2^0 to R_2^3, the corresponding KV pairs may be stored in a different sub-level (②). For instance, the KV pairs for R_2^0 are placed at S_2, while those for R_2^2 are placed at S_0.

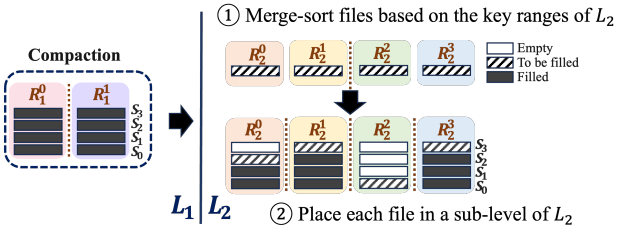


Figure 5. Upper level compaction process of DOPA-DB

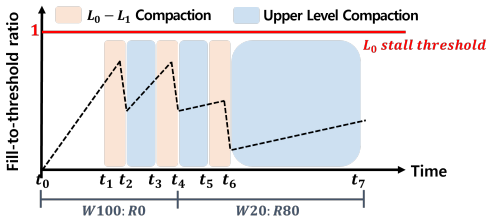


Figure 6. Ideal compaction scenario

Note that before starting compaction for the selected, possibly multiple, ranges in L_i, if any of the sub-levels in L_{i+1} for the corresponding ranges are filled, then DOPA-DB first performs compaction of those ranges of L_{i+1} to L_{i+2} before compacting L_i. For compaction between L_{n-2} and L_{n-1}, leveled compaction, where for a given range, KV pairs in L_{n-2} are merge-sorted with KV pairs in L_{n-1}, will be performed.

Note that DOPA-DB is built on top of RocksDB. Thus, DOPA-DB basically adopts the triggering mechanisms of RocksDB for flush and compaction operations as well as the level-selection mechanism for compaction operations. One modification made by DOPA-DB is that when no level is chosen by the level-selection mechanism employed by RocksDB (because the size of each level is less than its capacity), DOPA-DB selects the lowest level that contains any SSTable file for compaction so that the inserted keys are sorted and stored in the last level or a few upper levels.

Dynamic compaction recommendation A key component of DOPA-DB is its ability to determine the ideal compaction size that should be performed by the upper levels (L_i, where $i > 0$) that we refer to as Upper Level Compaction (ULC) size, which is determined by CSR. To see the usefulness of CSR, consider the extremes of write-intensive and read-intensive workloads. For the former, ULC should occur at the minimum such that L_0 can absorb as much writes as possible, but not so much to incur write stalls. Thus, a lazy form of compaction would be adequate. For the latter, aggressive compaction would benefit as this will reduce the number of files as well as the non-empty levels (meaning some levels may have zero data), which will benefit read performance. Moreover, as overlapped SSTable files are allowed in L_0, which makes a read operation very expensive, it is critical to aggressively perform $L_0 - L_1$ compaction.

Consider a more general case as shown in Figure 6 where the y-axis is the fill-to-threshold ratio, the x-axis is time, and Wa:Rb denotes the ratio of write and read operations to be a to b. Recall that as writes happen, they start to fill up the MemTable, and when full, it is flushed to L_0. Thus, the flush rate reflects the write intensity. Furthermore, as discussed in Section 2, earlier studies have observed that L_0 stalls have significant influence on performance and that it is triggered when the L_0 size reaches some threshold, which we refer to as the L_0 stall threshold [5] (also denoted in Figure 6). The dashed line in Figure 6 represents the rate at which L_0 fills up. The faster L_0 fills up the steeper it will be as depicted by t_0 to t_1, and vice versa as depicted by t_6 to t_7. Then, to avoid this stall, $L_0 - L_1$ compaction must occur and finish before the threshold is reached, which is represented by the yellow boxes (t_1-t_2, t_3-t_4, etc.). When this compaction is done, the fill-to-threshold ratio, that is, the size of L_0, is finally reduced, represented by the dropping dashed line. It is important to note that while performing $L_0 - L_1$ compaction, flushes (of MemTables) can occur concurrently, increasing the fill-to-threshold ratio as shown in the figure.

While this periodic filling and emptying of L_0 space is occurring, note where ULC is occurring, represented by the blue boxes (t_2-t_3, t_4-t_5, etc.) and the time span (x-axis) of the boxes. That is, the time spent for ULC is dictated by how fast the writes fill up L_0 and L_0-L_1 compaction time. When writes are intense as in t_0-t_4, the dashed line will be steep, and ULC should be short, that is, the ULC size must be small, while when writes are less so (or read intense) as in t_4-t_7, ULC size can be large as there is much more slack before the stall threshold is reached. Note that the workload pattern changes at t_4, and the ULC size starting at t_4, which is computed based on the flush speed during the last time window, is the same as before, but the ULC size starting at t_6 becomes much larger as the flush speed decreases from t_4.

The above notion is formalized as Equation 1 where T is the L_0 stall threshold, M_0 is the current size of L_0, F is the current flush speed, which is how much data is flushed per time unit, C is the accumulated compaction speed, which is computed as the total size of compaction divided by the total compaction time so far, and RS_0 is the estimated $L_0 - L_1$ compaction size, which is computed as T divided by r_0 (where r_0 is the number of ranges in L_0) as in this work, DOPA-DB performs $L_0 - L_1$ compaction for each key range in round-robin fashion. CSR uses this equation to compute the maximum compaction size that will not cause a write stall. The first parameter T we take from RocksDB, while M_0, F, and C are collected by monitoring the runtime states.

$$C_{idl} = \left(\frac{T - M_0}{F} - \frac{RS_0}{C}\right) \times C \quad (1)$$

As an example, consider that M_0 is 800MB, F is 80MB/s, C is 100MB/s, RS_0 is 200MB, and T is 1.2GB. In this case, the remaining L_0 size until a stall occurs is 400MB (=1.2GB-800MB). Then the time until a stall occurs can be estimated as 5 seconds (= $\frac{400MB}{80MB/s}$) and L_0-L_1 compaction requires 2

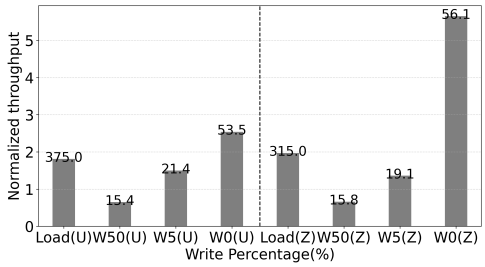


Figure 7. Throughput of DOPA-DB normalized to RocksDB under mixed workloads

seconds (= $\frac{200MB}{100MB/s}$). Thus, the CSR computes the maximum compaction size that can be done within 3 seconds (5-2 seconds) as 300MB (= 100MB/s×3 seconds). This size is provided as a hint. As the minimum unit of compaction is a key range in DOPA-DB, it can compute the number of key ranges for compaction for a given hint such that the total size of the selected key ranges is at most the hint, or one range is selected if the size of the range is larger than the hint.

4 Experiment

For our experiments, we use a machine with a 32-core Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz, 32GB of DRAM, and a Samsung 870 EVO SSD 1TB disk. We run the Ubuntu 20.04.4 LTS operating system. DOPA-DB is implemented based on RocksDB [5]. We compare our DOPA-DB with RocksDB with default parameter value settings, where the sizes of MemTable and L_0 are set to 64MB and 256MB, respectively, which DOPA-DB also uses. For both RocksDB and DOPA-DB, we make use of Direct I/O [3] so that the read operations are not affected by the page cache, which is similarly done in earlier studies [14, 19, 23]. For DOPA-DB, the values of p, R_{ratio}, and r_0 are set to 4, 4, and 4, respectively. A single compaction thread is used for both DOPA-DB and RocksDB.

We use mixed workloads with a dataset generated by the Load operation of YCSB [12] with keys and values of 16 bytes and 100 bytes, respectively, and use the Uniform and Zipfian query distributions. For each distribution, we first initialize the database by loading 100 million KV pairs (with 16 threads) and then execute three workloads, W50:R50, W5:R95, and W0:R100 (where Wa:Rb denotes the a to b ratio of write and read operations), which are similar to workloads A, B, and C, respectively, in YCSB [12], but write operations are invoked instead of update operations. Each of the individual workloads consists of 300 million operations for a total of 900 million operations.

Figure 7 shows the throughputs of our mixed workloads with DOPA-DB, normalized to those of RocksDB, where Wa(U) and Wa(Z) denote the workload with a% writes for the Uniform and Zipfian distributions, respectively. The number above each bar indicates the absolute throughput value in KOps/s. Also, the total amount of writes for DOPA-DB normalized to that of RocksDB is given in Table 1. For Load, DOPA-DB provides 88.1% higher throughput than RocksDB, while the total amount of write is 44.7% lower on average. Also, we analyze that for Load, the total stall time with DOPA-DB is 69% lower on average. Except for Load, the total amount of writes with DOPA-DB is much higher than RocksDB as DOPA-DB aggressively performs compaction for read operations. For W50(U) and W50(Z), DOPA-DB shows lower throughput compared to RocksDB.

Table 1. Total write amount of DOPA-DB normalized to RocksDB

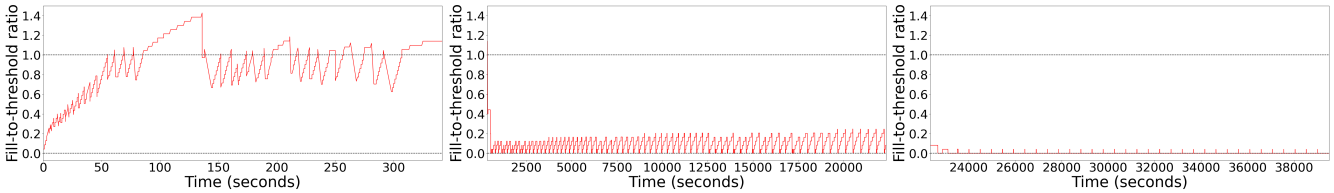
	Uniform				Zipfian			
Write %	100	50	5	0	100	50	5	0
Norm. total write amount	0.57	5.82	8.54	8.63	0.54	4.39	7.33	7.42

We observe that in DOPA-DB, compactions with different sizes are performed as patterns change. Figures 8(a), (b), and (c) show, for example, the fill-to-threshold ratios over time with DOPA-DB for Load(U), W50(U), and W5(U), respectively. The results in the figure show how DOPA-DB, as a consequence of using different ULC sizes as workload patterns change over time, avoids reaching the L_0 stall threshold.

Figure 9 shows the performance of DOPA-DB for two more workloads, one where we execute W10:R90 followed by W90:R10, and the other where we execute W90:R10 followed by W10:R90, after initializing the database. For Figure 9(b), the workload changes from write-intensive to read-intensive as in Figure 7. However, the performance trend of Figure 9(b) is different from Figure 7. We attribute this to our current version that uses an adaptation method that is too naive to accurately and quickly identify workload pattern changes. These issues and more are discussed further in Section 6.

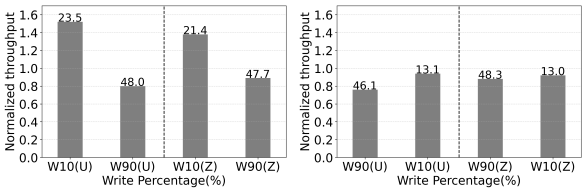
5 Related work

There have been many attempts to reduce the write amplification of LSM-based KV stores. Tiered compaction is one approach leveraged by existing studies [6, 21, 27]. PebblesDB presents Fragmented Log-Structured Merge Trees (FLSM) that divides the key space for each level into disjoint units [21]. FLSM allows the system to avoid rewriting data in the same level. WipDB partitions the key space into N buckets where N is adjusted based on the key distribution [27]. Each bucket has its own LSM structure within. These techniques similarly use key ranges but only one key range is compacted per compaction schedule of a level, possibly aggravating the compaction cost. Additionally, there have been prior studies leveraging machine learning to provide appropriate compaction policies considering dynamic workloads [11, 20] such that tiered compaction is employed when a workload is write-intensive while leveled compaction is employed when it is read-intensive. There have also been efforts to adjust I/O to reduce query latency or minimize stalls [2, 8]. In particular, SILK builds an I/O scheduler for LSM-based KV stores to handle high tail latency and prevent stalls [8]. However, it may delay compaction on upper levels,



(a) Load(U) (b) W50(U) (c) W5(U)

Figure 8. Fill-to-threshold ratios over time



(a) W10:R90 followed by W90:R10 (b) W90:R10 followed by W10:R90

Figure 9. Normalized performance under shifted workloads

increasing compaction overhead on the next turn. Dostoevsky [14] and Monkey [13] aim to find the optimal trade-off of the write and read cost. Dostoevsky introduces a technique to model the most efficient LSM structure based on the write and read ratio [14]. Monkey minimizes worst-case read cost by allocating memory carefully to Bloom Filters [13].

6 Discussions and Future work

As seen, we observe potential benefits of dynamic compaction. Yet, we also observe that there are many aspects that still need in-depth investigation for it to be deployable, a few of which we now discuss.

Nimble and accurate adaptation For non-write intensive workloads, nimble adaptation to a workload pattern is needed. DOPA-DB currently uses a simple window-based method to identify the workload pattern based on the flush speed and only considers the state of L_0 for computing the ULC size. For nimble adaptation, accurate workload pattern recognition is needed, even possibly with forecasting capabilities. The ULC size should also consider read patterns such that smaller ULCs could be considered, instead of uniformly large ULCs, even for read-intensive workloads. In addition, other factors such as disk bandwidth usage need to be considered for accurate estimation as the performance of read operations may be affected by compaction.

Using multiple compaction threads With two compaction threads, the performance of RocksDB increases as one thread can handle L_0-L_1 compaction while the other performs ULC. The performance of DOPA-DB can also be improved with multiple compaction threads. Dynamic compaction strategies with multiple threads need to be studied.

Parameters of DOPA-DB DOPA-DB has a set of parameters to configure; p, which is the number of sub-levels for each level i (where $0 < i < n - 1$), R_{ratio}, which is the range amplification ratio, and r_0, which is the number of key ranges in L_0. These parameters will affect the size of the key range and the compaction unit. That is, the number of key ranges for level i is computed based on r_0, and R_{ratio}, and the compaction unit for level i is computed based on the key range size and p. When choosing the values for these parameters, there are two considerations: (1) the size of any key range should not be too small lest the SSTable files become too tiny and (2) the size of the compaction unit for any level should not be too large so that fine-grained dynamic compaction sizes can be supported. In this work, we have chosen the values of the above parameters empirically, but a detailed study about the effects of these parameters is needed.

Skewed datasets While we only considered the uniform dataset in this study, real-world datasets have been shown to have key distributions that are not uniform and that may change over time [9, 22, 24, 26]. In such dynamic datasets, the initial key ranges of levels may suffer from imbalance such that keys are mainly inserted to certain ranges. That is, each range in a level may contain a varying number of keys from each other, which may prevent CSR from computing different compaction sizes in a fine-grained manner. Dynamic readjustment of key ranges is anticipated to have a strong effect on the performance for real-world datasets.

7 Conclusion

In this paper, we argued for a dynamic workload pattern aware LSM-based KV store, which enables fine-grained control over compaction sizes. Our preliminary results showed possible performance improvements with dynamic compaction sizes.

Acknowledgement

We would like to thank the anonymous reviewers for their invaluable comments. This research was partly supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2021-0-01817) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), National Research Foundation of Korea (NRF-2023R1A2C2006432), NSF grant 2312785, and Samsung Electronics Co., Ltd. Young-ri Choi is the corresponding author.

References

[1] 2016. HyperLevelDB. https://github.com/rescrv/HyperLevelDB.

[2] 2017. RocksDB Rate Limiter. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html.

[3] 2020. RocksDB Direct IO. https://github.com/facebook/rocksdb/wiki/Direct-IO.

[4] 2022. LevelDB. https://github.com/google/leveldb.

[5] 2022. RocksDB. https://rocksdb.org/.

[6] 2023. RocksDB Universal compaction. https://github.com/facebook/rocksdb/wiki/Universal-Compaction.

[7] D. F. Andrews and A. M. Herzberg. 1985. *Monthly Mean Sunspot Numbers*. Springer New York, New York, NY, 67–74. https://doi.org/10.1007/978-1-4612-5098-2_12

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*. 753–766.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)* (2008), 1–26.

[11] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. 2023. Workload-Aware Log-Structured Merge Key-Value Store for NVM-SSD Hybrid Storage. In *Proceedings of 2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2207–2219.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*. 143–154.

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. 79–94.

[14] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 505–520.

[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 1189–1206.

[16] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS Under HBase: A Facebook Messages Case Study. In *Proceedings of 12th USENIX Conference on File and Storage Technologies (FAST)*. 199–212.

[17] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards workload-aware self-management: Predicting significant workload shifts. In *Proceedings of 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*. 111–116.

[18] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review (SIGOPS)* (2010), 35–40.

[19] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. 2019. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*. 739–752.

[20] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proceedings of the ACM on Management of Data (PACMMOD)* (2023), 1–25.

[21] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the ACM SIGOPS 26th Symposium on Operating Systems Principles (SOSP)*. 497–514.

[22] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 308–320.

[23] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. AC-Key: Adaptive caching for LSM-based Key-Value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC)*. 603–615.

[24] Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. 2023. DyTIS: A Dynamic Dataset Targeted Index Structure Simultaneously Efficient for Search, Insert, and Scan. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. 800–816.

[25] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *Proceedings of 21st USENIX Conference on File and Storage Technologies (FAST)*. 65–80.

[26] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. 147–162.

[27] Xingsheng Zhao, Song Jiang, and Xingbo Wu. 2021. WipDB: A Write-in-place Key-value Store that Mimics Bucket Sort. In *Proceedings of 2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1404–1415.