IEIE Transactions on Smart Processing and Computing, vol. 9, no. 4, August 2020
https://doi.org/10.5573/IEIESPC.2020.9.4.317

Deduplication Adapted CaseDB for Edge Computing

Khikmatullo Tulkinbekov, Joo-Hwan Kim, and Deok-Hwan Kim

Department of Electronic Engineering, Inha University Incheon, Korea
{mr.khikmatillo, wnghks3030}@gmail.com, deokhwan@inha.ac.kr

***** Corresponding Author: Deok-Hwan Kim

Received January 13, 2020; Revised February 9, 2020; Accepted March 15, 2020; Published August 30, 2020

** Regular Paper*

Abstract: Log-structured merge-tree (LSM-tree) family key-value stores are becoming the databases most in demand for big data systems. They provide an easy-to-implement interface, and they automatically perform garbage collection by applying a compaction procedure over the multi-level structure. CaseDB offers various advantages by reducing write amplification considerably, using a metadata compaction technique. However, it suffers from a space amplification problem in update-intensive workloads. As an implementation of the LSM-tree structure, CaseDB refuses to instantly perform deletes, but delays them for the compaction process, resulting in an increasing amount of deprecated data. This paper proposes a deduplication extended compaction method for CaseDB. It scans for duplicated keys within the compaction method and removes the old values. Experiment results show that the proposed technique offers various threshold values of deduplication for different balances between space amplification and write amplification.

Keywords: Key-value store, CaseDB, Deduplication, Space amplification, Write amplification

1. Introduction

Key-value (KV) stores are widely used in data-intensive applications for their scalability and flexibility. They are a type of non-relational structured query language (NoSQL) [7] database that provides a simple interface using public methods like Put, Get, and Delete to manage the data, and they do not rely on heavy SQL queries like traditional relational databases. Over the last decade, the log-structured merge-tree (LSM-tree) [1] has become a dominating data structure in the implementation of KV stores. The biggest companies are demanding LSM-tree family KV stores as their primary database in data centers. LevelDB by Google [5], RocksDB from Facebook [6], and DynamoDB from Amazon [8] can be state-of-the-art examples of LSM-tree family KV stores. LSM-tree implements a multi-level structure where each level employs a bigger space than the previous one. New data are first located in the smallest and highest levels and are moved to lower ones using the compaction process as the levels fill up. The compaction process merges and sorts data from two adjacent levels into a lower one. It reads all the data according to a specific key range, and rewrites them after compacting. After rewriting the new, sorted order into a lower level, old data are deleted from each level. Compaction is the core idea for LSM-tree and replaces several essential techniques in non-LSM KV stores. First, it solves data defragmentation issues by repeatedly rewriting all the data for new insertions and deletions. Secondly, it eliminates the need for garbage collectors by automatically deleting the updated data while merging the levels. Finally, it performs in-memory sorting that guarantees the level base–sorted KV items. Thanks to these advantages that the compaction process offers, LSM-tree–based KV stores provide compatible Read and Write performance compared to other types of NoSQL and traditional relational databases, and are becoming the most promising databases for big data systems. On the other hand, LSM-tree cannot handle instant updates. If the value is updated, the old one remains in the database until the compaction process reaches the level holding it. Since the updated item will be stored in the highest level, the Read process never reaches the old value, so this problem might seem to be considered negligible. But in update-intensive applications, it causes a space amplification problem that needs to be avoided for the best use of hardware resources.

As a result of the exponential increase in data in recent years, Write- and Update-dominated workloads [20, 23] have become common in cloud-based applications. For example, observations nowadays show that each person is generating 1.5MB of data each second using different devices they own, and this rate is increasing over time.

Moreover, 90% of all data in human history was created in the last decade [14]. These statistics prove that the amount of data is increasing exponentially each day. Most data are saved in different user-interactive social apps, like Facebook, Instagram, WeChat, or Telegram. Moreover, with help from more variety in new features offered by applications, users are demanding not only to share data interactively but also to edit and delete their recent updates simultaneously. It also increases the proportion of update operations at the database level. As mentioned above, LSM-tree allows each level to keep an old copy of updated data until compaction reaches that level. In big data systems, this causes a space amplification problem if the update proportion is high. Space amplification refers to having several duplications of the new data in the database while it is being written. It can be visualized in the space amplification ratio (SAR), which is the average number of duplicates for each unique key in the KV store.

Many researchers have put a lot of effort into enhancing data communication performance [10, 21], and into improving the performance of LSM-tree KV stores [3, 9, 11-13]. But most of them focused only on improving the write amplification problem, which refers to the number of data rewrites because of compaction. The abovementioned problem has not been at the center of researchers' attention because it was considered negligible. However, nowadays, space amplification is becoming one of the most important factors to be avoided, because the needs of applications are changing so fast. Recently, the deduplication-triggered compaction method [15] was introduced to reduce the SAR by increasing the frequency of compaction according to the duplication ratio in LSM-tree families. But increasing compaction frequency meanwhile results in an increase in the write amplification ratio (WAR). So, this method has a serious drawback in dealing with the tradeoff between SAR and WAR. In this paper, for one of the most recent LSM-tree–based KV stores (CaseDB [4]), we present a new deduplication method that decreases the SAR on update-intensive workloads. The new approach tries to resolve two main challenges that occur when applying deduplication:

- reducing space amplification
- keeping the balance between space amplification and write amplification

Our experiment results show that the method used is able to eliminate the SAR completely for higher threshold values while maintaining a balance in Write throughput.

The rest of the paper is organized as follows. Section 2 discusses the overall background and the CaseDB architecture, and introduces the space amplification problem in detail. Section 3 presents the proposed method, and in Section 4, the experiment results are evaluated. Finally, Section 5 gives a brief conclusion.

2. Background and Motivations

There are several implementations of LSM-tree as a KV store. LevelDB is the simplest one, which comes with multi-level memory and disk components constructed by fixed-size data units called memtables and string-sorted tables (SSTables). Data are inserted into skip-tree–based memtables in memory first, and are flushed to SSTables in persistent storage. The compaction process moves SSTables to lower levels as all the data reach the lowest levels after several compactions. RocksDB enhances the Read performance of LevelDB by applying multi-thread key lookups in disk levels. On the other hand, CaseDB focuses on improving Write performance by aggressively separating keys from values and implementing a new metadata layer on disk. Apart from Atlas [16] and WiscKey [17], CaseDB avoids small data writes using CBuffer [2] and moves Bloom filters [18] from SSTables to metadata files. It also reduces write amplification while processing most of the compaction on metadata files and reducing unnecessary file I/O on the data layer. Since CaseDB is one of the latest introduced KV stores based on Google's LevelDB, and it offers low-cost key lookup possibilities in the metadata layer, we implemented the proposed method on its architecture.

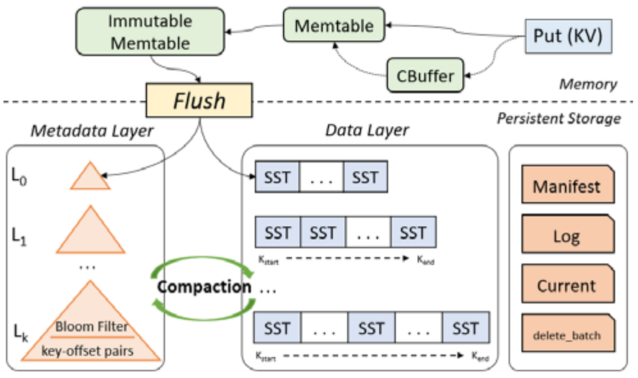
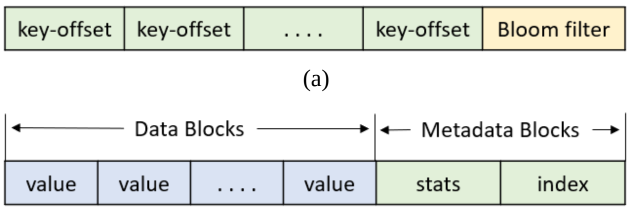


Fig. 1. The CaseDB Architecture.



(a)

(b)

Fig. 2. CaseDB Data Units (a) the Metadata Layout, (b) the SSTable Layout.

2.1 CaseDB Architecture

CaseDB is one of the most recent and advanced implementations of LSM-tree as a KV store. As shown in Fig. 1, it inherits memtables and SSTables from LevelDB, and employs extra data units in both memory and at the disk level. Newly inserted KV items are first inserted into a memtable directly if the size is bigger than a threshold value. Otherwise, they are inserted into CBuffer [2] for merging multiple small values into a larger chunk, and are moved to a memtable after logging metadata about the grouping state. When a memtable is full, it is converted to an immutable memtable, where data will be flushed to disk from memory. All memory components are implemented using a skip list [22] that provides a consistent O(log(n)) complexity for all data operations. In persistent storage, CaseDB employs two layers for metadata and data constructed by metadata files and SSTables, as shown in Figs. 2(a) and (b). Both layers implement the LSM-tree structure with multiple levels, in which each next level is 10 times bigger than the previous one. When data are flushed, keys are aggressively separated from values and inserted into a metadata layer with offset references to the corresponding value. To improve data lookup performance, Bloom filters are saved to metadata files with key-offset pairs. Actual values are then written to SSTable files and are located in the data layer. When *L0* is filled up, compaction occurs, and metadata files from *L0* and *L1* are rewritten to *L1*. This procedure continues until each level reaches the maximum size, and metadata files are moved to lower levels. Once compaction is done with the metadata layer, SSTable files in the data layer are relocated to corresponding levels following the metadata files. CaseDB also employs a *delete_batch* log file to keep track of deprecated KV items in the data layer to provide data consistency. The *delete_batch* file is initialized during the first part of compaction when only the metadata layer is involved, and it plays a crucial role in the next part to remove updated and deleted KV items from the database while SSTables are relocated. Employing metadata compaction and delayed relocation for values reduces the write amplification problem for CaseDB, which is the most vulnerable point of LevelDB and most of the LSM-tree–family KV stores.

The Read process in CaseDB follows the top–down data lookups for a specific key. The searched key is first checked from memory components in CBuffer and memtables. If it cannot be found in memory, the metadata layer will be searched. Starting from the highest level of the metadata layer, only Bloom filter checkout is performed first. If the Bloom filter returns positive, then the metadata file is searched for the key. Since the Bloom filter might return a positive result even if the data does not exist in the set, the key might not be found in the metadata file, and the process continues with the next level. Once the key is found in the metadata file, it references the SSTable holding the value of the key. In this scenario, CaseDB guarantees that only one SSTable will be read during the Read process.

2.2 Bloom Filters

The Bloom filter [18] is a data structure used for checking the existence of a particular item in the set without reading the set. It is a lightweight and adaptable filter, since its implementation requires less space allocation on disk, compared to other data structures. If the feedback it returns is negative, the item is not in the set. For these advantages, Bloom filters are widely used in many applications.

The Bloom filter is a combination of *n* bits and *k* hash functions ($H_i(k)$). An example of a Bloom filter with 16 bits and five hash functions is shown in Fig. 3. Values for all bits are initialized to 0 in the beginning, as shown in Fig.

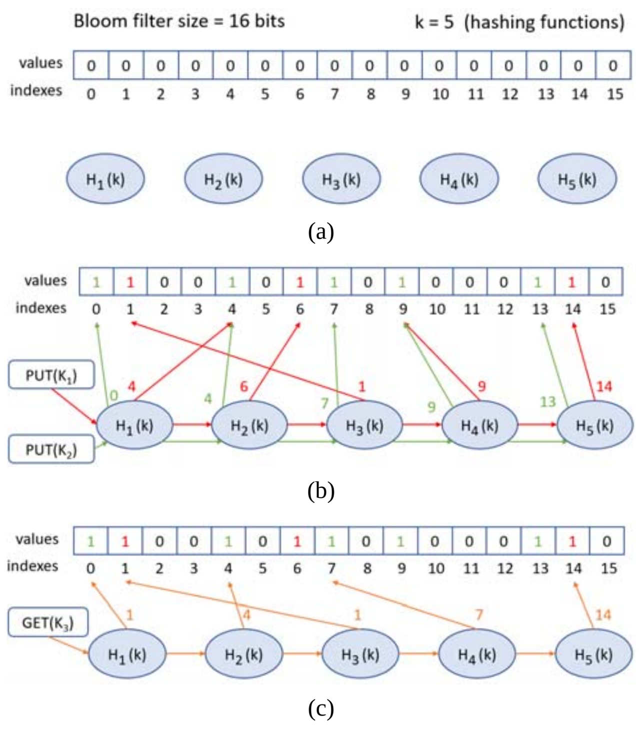


Fig. 3. Bloom Filter Example (a) the Initial State, (b) the State after Initialization, (c) a False Positive.

3(a). In CaseDB, when the data are flushed from memory to disk, keys are separated from values and written to metadata files with the corresponding offset. In this process, each key participates in Bloom filter initialization by checking every $H_i(k)$. Fig. 3(b) shows an example of inserting two keys into a Bloom filter using hash functions. Each $H_i(k)$ returns a unique number for each key that is smaller than the maximum number of bits in the Bloom filter. The generated number from $H_i(k)$ is considered an index of the filter, and the corresponding value will be initialized to 1. There is the possibility that $H_i(k)$ generates a number that was already initialized by the previous key. Even if the value is already initialized, it will not alter the value to 0 again, but reassigns 1. In the example, $H_2(k)$ returns 4 for K_2, and $H_1(k)$ returns 4 for K_1, and both initialize the same bit in the set. Following the same scenario, all keys are checked through all hash functions and initialize the Bloom filter.

When reading the keys from CaseDB, only Bloom filters from metadata are checked first. The searched key is given to the same hash functions, and they will return indexes according to the key. Every $H_i(k)$ returns the same number for the same key. So, if the key is already inserted, indexes from every $H_i(k)$ should already be 1. If any bit value is 0, the Bloom filter comes back negative, which guarantees the key does not exist in the metadata. If all bits are already 1, the Bloom filter returns a positive result, and then, the metadata file can be searched for the specific key. In this way, the Bloom filter provides the possibility of getting a prediction about data existence in the set without reading the whole file. However, the Bloom filter suffers from false positives. In Fig. 3(c), K_3 is searched from the same Bloom filter initialized above. We know that K_1 and K_2 were inserted into the filter, but K_3 does not exist in the set. In some cases, all hash functions return indexes where those values are already assigned 1. Because all values are 1, the Bloom filter returns positive results even though the data do not exist in the set. Therefore, a Bloom filter cannot give trusted information about the existence of the item in the set, even if it comes back positive, but it guarantees data are not in the set in the case of negative output. In order to reduce the false positive rate, the Bloom filter size and number of hash functions should be calculated carefully according to the number of items in the set. Employing fixed-size metadata files limits the number of keys in the metadata, and CaseDB always keeps the false positive rate at a low percentage.

2.3 Space Amplification

The compaction process performs repeated file Read and Write I/O causing write amplification, which is the weakest part of LSM-tree KV stores. CaseDB solves this issue by separating keys from values and delaying value compaction. But that is the only problem that should be considered. As well as offering advantages by guaranteeing sorted data, and performing automatic garbage collection with the compaction process, LSM-tree allows keeping the deprecated data in the database until compaction reaches them and removes them. Before going deeper into the problem, we should know about the update scenario in CaseDB. Since it inherits the main architecture from LevelDB, there is no separate method for updating the data. If the actual KV item needs to be updated, the same key should be inserted with the new value. The compaction process identifies two identical keys at different levels and removes the old one. For this reason, the old value is not removed from the database instantly but is allowed to allocate physical memory until compaction reaches the holding level and removes it. In general, this looks like a negligible problem. Even though deprecated data is not abolished instantly, compaction finds it and cleans the disk. However, not all workloads are Put- or Read-oriented. Some applications need to perform update operations intensively. In this kind of system, delaying delete operations causes the severe space-amplification problem, which simply refers to situations where the actual database size gets considerably bigger than the actual data size. Accurately, in CaseDB, all levels can keep the old values until compaction reaches them. Considering that the size of each next level is 10 times bigger than the previous one (and compaction occurs only if the size of the level reaches the threshold), the frequency of compaction gets 10 times slower for each lower level. In other words, lowest levels keep the highest proportion of data, and because compaction frequency is low at those levels, most of the data become deprecated in update-intensive workloads.

Fig. 4 shows a comparison of WAR and SAR in CaseDB for different update shares of a 500 GB workload. As can be seen from the graph, write amplification is not affected by the update share. Because the update is another view of a Write operation, CaseDB achieves the WAR at values between 2 and 3, as promised for all cases. But the SAR is highly affected by update share, because it was

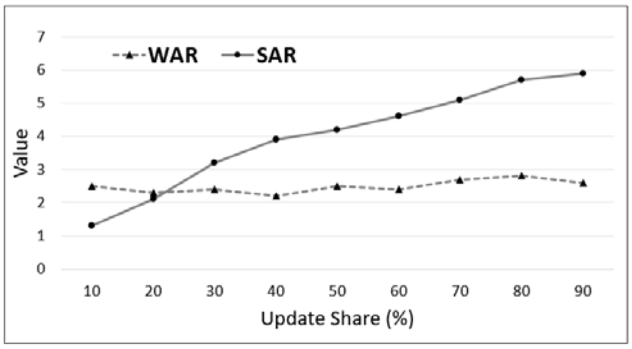


Fig. 4. Write and Space Amplification Ratio in CaseDB for Different Update Proportions.

negligible for low updates and reaches 6 in update-intensive cases. This value means the database size gets six times bigger than the workload size. We also need to consider that this factor highly depends on the number of levels in the database. For bigger workloads, the SAR gets even bigger. In our experiments, we found the database reaches 3TB for a 500GB workload when performing updates. Even though it is a temporal case, and size will decrease after completing compaction for all levels, as the workload size increases, it causes a severe space leakage problem. Moreover, it needs to provide a hard disk six times bigger than the actual data stored on it. Some applications operate with several terabytes of data, and space amplification becomes a crucial problem to be avoided for better performance. One easy solution to the problem could be increasing the compaction frequency at lower levels. But we should also consider increasing the number of compactions, which results in a higher WAR that reduces performance considerably.

Pointing out the problem mentioned above, we propose a deduplication extended compaction method for lower levels in CaseDB. The proposed technique is focused on solving two main challenges: one is reducing the SAR, and the other is keeping a balance between SAR and WAR. There is always a tradeoff between SAR and WAR in LSM-tree KV stores; it is impossible to reduce both, and the balance should be considered carefully.

3. The Proposed Technique

Implementation of the deduplication extended compaction method focuses on eliminating the space amplification problem in CaseDB, and provides a reliable and flexible KV store for mixed workload applications. The technique resolves the issue by performing compaction in the lower level if the deduplication ratio reaches the threshold. Moreover, this method keeps a reliable balance between space amplification and write amplification. Running compaction in lower levels does not increase the frequency, but extends latency. In other words, the proposed method does not run compaction based on deduplication, but runs deduplication during each compaction occurrence. Delaying the compaction with the deduplication checking process incurs extra I/O, but this

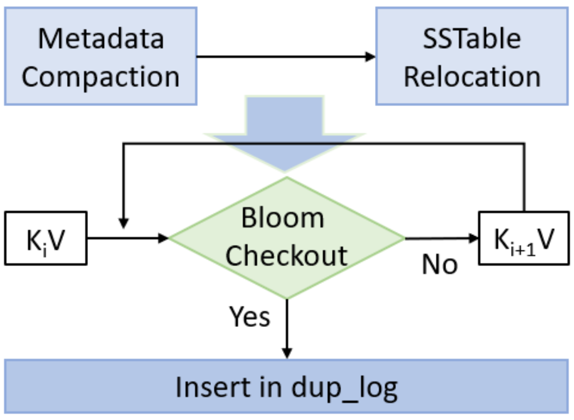


Fig. 5. CaseDB Deduplication-Oriented Compaction Flowchart.

process only affects the duplicated data and costs considerably less than running compaction from the beginning.

Figs. 5 and 6 show the flowchart and the procedure for extended compaction with the data deduplication checkout for the CaseDB architecture. As discussed above, the CaseDB compaction process has two parts. In part 1, metadata compaction is performed, and in part 2, SSTables are relocated according to the updated metadata. Part 1 lasts for all levels that reach the size threshold. After finishing both parts, the data duplication checkout process starts. The *DedupCheckout()* method only involves all lower levels that Part 1 did not reach, and its internal procedure is shown in the lower part of Fig. 5. In the *DedupCheckout()* method, Bloom filters play a crucial role in checking for the existence of KV items without reading the metadata. In this step, each updated key in upper levels is checked and marked as duplicated data or not. As discussed above, the Bloom filter returns either a positive or a negative result, and it is guaranteed that the data is not available in the set if the feedback is negative. So, the key will just be skipped after a negative result and the process repeats with the next one. Also, the probability of false positives is reduced as small as possible in CaseDB, so *DedupCheckout()* assumes the key exists in the metadata file if the Bloom filter's feedback is positive and inserted into the *dup_log* file. Even if it does not exist in the set, extended compaction finds it in the next steps, and this does not affect data consistency. Therefore, the duplication checkout process runs surprisingly fast by only involving Bloom filters.

$$DUP\_RATIO = TOTAL\_KEYS / DUP\_KEYS \qquad (1)$$

After checking all lower levels for duplicated keys, the duplication ratio is calculated using (1). This equation depends on two variables to calculate the duplication ratio. One is the total number of keys that can easily be extracted from metadata files, and the other is the number of duplicated keys, which is available in the *dup_log* file initialized in the previous step. The calculated value is used in the next step to check the threshold for deduplication. From the equation, it is easy to see that a higher value for *DUP_RATIO* increases the frequency of deduplication and guarantees better results in terms of space amplification. Also, *TOTAL_KEYS* is always higher than *DUP_KEYS* since the total number of keys also includes the count of keys in upper levels where compaction was already performed, and the number of duplicated keys includes only lower levels. Even if all keys are duplicated at lower levels, the value of *DUP_RATIO* never reaches 1. So, it is recommended to use bigger threshold values for better performance.

Algorithm 1 Deduplication Extended Compaction

```
function COMPACT()

procedure
    if $NeedsCompaction()$ then
        while $NextLevelSize >= THRESHOLD$ do
            $SelectKeyRange()$
            $MetadataCompaction()$
        $SSTableRelocation()$

        $//Deduplication\ starts\ here$
        $DedupCheckout()$
        $TOTAL\_KEYS \leftarrow GetTotalKeys()$
        $DUP\_KEYS \leftarrow GetDupKeys()$
        $DUP\_RATIO \leftarrow TOTAL\_KEYS\ /\ DUP\_KEYS$
        if $DUP\_RATIO <= DEDUP\_THRESHOLD()$ then
            while $NextLevel \neq NULL$ do
                $SelectKeyRange()$
                $MetadataCompaction()$
            $SSTableRelocation()$
        $//Deduplication\ ends\ here$
```

Fig. 6. The Deduplication Extended Compaction Algorithm for CaseDB.

In Line 15, the threshold is checked to determine whether deduplication is required or not. If the value of *DUP_RATIO* is smaller, then compaction will be extended to all lower levels, and it automatically removes duplicated data. The balance between WAR and SAR can be managed by altering the deduplication threshold according to system needs and workloads.

4. Performance Evaluation

CaseDB is designed for embedded devices as a lightweight KV store. Embedded devices are applicable in many hardware-limited systems like edge servers. In edge computing, the response time highly depends on data management performance, so low-cost databases are crucial in improving overall performance. We also use one of the latest embedded devices applicable to edge computing, called the Jetson AGX Xavier (NVIDIA, Santa Clara, CA), with hardware parameters shown in Table 1. The board comes with an ARMv8 processor and 16GB of RAM. Also, the default operating system is Ubuntu, which makes it easier to compile and run the applied methods on the hardware. We attached a 500GB SSD drive using an external wire connector to the eSATA port.

We used DB_BENCH [19], the KV store benchmarking tool introduced by Google. Experiments were done with an average 300GB workload, which is

Table 1. Experimental Hardware Parameters.

Name	*Jetson AGX Xavier*
CPU	4 * ARMv8 Processor
Memory	16GB
OS	Ubuntu 18.04 LTS
SSD	500GB Samsung 860 EVO

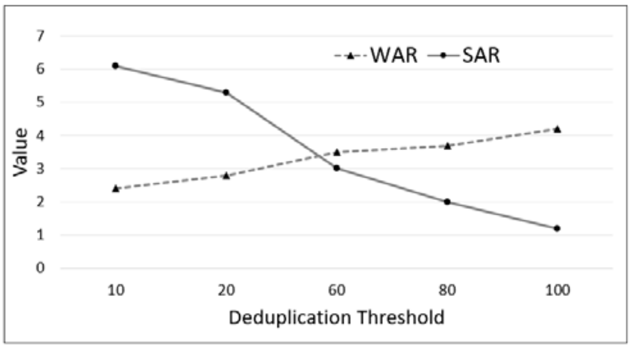


Fig. 7. CaseDB SAR and WAR with Deduplication Extended Compaction.

enough to create up to six levels in the CaseDB architecture. Since the SSD space is limited to 500GB, the workload size was altered for different thresholds. The experiments were performed based on three different criteria. First, we evaluated the balance of WAR and SAR using the applied deduplication method. Also, we evaluated the effect of deduplication on the Write performance of CaseDB. Finally, the WAR effect of the existing method and the proposed method were evaluated according to different SAR values.

4.1 WAR and SAR

As discussed in previous sections, when reducing space amplification by applying a deduplication algorithm, the most important thing to consider is keeping a balance between WAR and SAR. Fig. 7 shows the values of both factors for different thresholds of the duplication ratio. We have already mentioned that it is highly recommended to limit the lower bound of the threshold because the duplication ratio never reaches 1. Moreover, very high values for the deduplication threshold might result in unacceptable results in terms of write amplification. In our experiments, we selected boundaries between 10 and 100 and recorded the values of WAR and SAR in the figure. When we achieved a five times improvement in terms of SAR, we ended up with a 4.1 WAR, which is almost two times bigger than the initial value. This result shows that the applied method keeps the balance acceptable. When it is about managing big data, achieving a five times improvement over space amplification by sacrificing the Write throughput by half can be considered an acceptable improvement for providing reliability. Moreover, we must consider that for bigger workloads of several terabytes, the

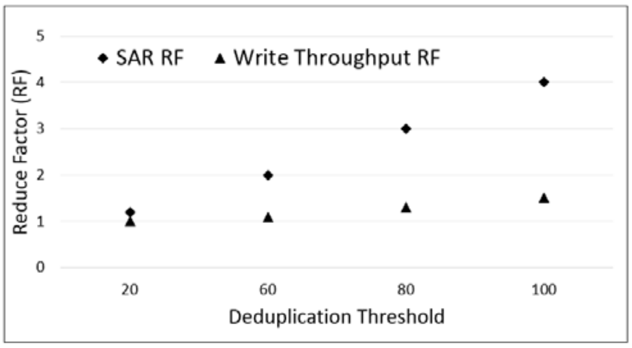


Fig. 8. CaseDB SAR and Write Throughput with Deduplication Extended Compaction.

SAR would be even bigger according to the number of levels. Depending on the workload, the threshold value varies.

4.2 SAR and Write Throughput

Another critical factor is the effect on Write throughput when the SAR is reduced. In some cases, Write throughput might look similar to write amplification since most latency for throughput is affected by the WAR. But they have different meanings in evaluations. Write throughput indicates the overall latency of CaseDB during the Put operation, including compactions that occur in the middle. On the other hand, WAR shows the number of file rewrites because of compaction. As we know now, only the Write operation awakens compaction, deletion, and update, and in some cases, Read can also invoke compaction affecting WAR.

Fig. 8 shows the reduce factor (RF) for the SAR and for Write throughput after applying the deduplication method for different threshold values in CaseDB. The reduce factor represents the ratio of the decrease of the SAR and the Write throughput for a given threshold. The higher RF is for the SAR, the lower space amplification becomes, whereas Write speed diminishes as RF-to-Write-throughput increases. From the figure, the low threshold does not have a considerable effect on Write throughput, but its effect is considerable for higher threshold values. When four-times improvement is achieved in terms of SAR, CaseDB's Write performance is reduced by 1.5 times. But still, when we consider the drawbacks of space amplification in big data systems, a sacrifice ratio can be acceptable for the application. Also, CaseDB achieves a considerably high Write throughput, even in an embedded-board environment, such that a 1.5 times loss does not affect the overall application performance at a critical level when we win a lot in terms of space usage and reliability.

4.3 Proposed and Existing Techniques

The existing method used for comparison [15] is quite simple, as discussed in the above sections. In the deduplication-triggered compaction method, the duplication ratio is checked regularly at the same frequency for levels of all sizes, and it can invoke compaction before the level reaches the maximum size

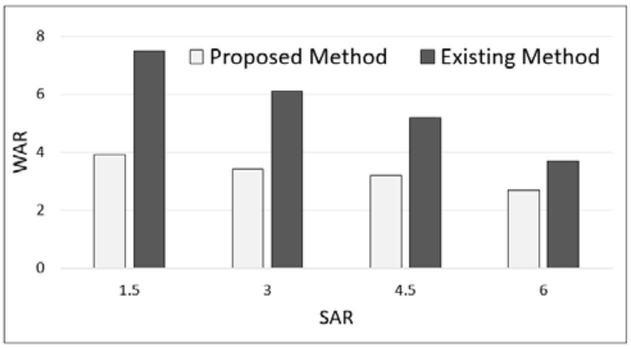


Fig. 9. Existing Deduplication Method and Proposed Method Comparison in CaseDB.

threshold. Even though it is easy to implement the technique, regularly invoking compaction results in an increase in file rewrites, and slows down the performance. On the other hand, our proposed method does not invoke compaction, but invokes deduplication during compaction, which results in far fewer I/O operations, compared to the existing method.

Evaluation of both methods in terms of WAR for different SAR values is shown in Fig. 9. In general, we can see that, with our proposed method, CaseDB achieves about two times lower WAR compared to the existing method. In the worst case scenario, when we want to reduce the SAR to 1.5, the proposed and existing methods show about 3.9 and 7.7 for WAR, respectively. As the SAR value increases, WAR decreases linearly for both, and achieves the default value for CaseDB, which is between 2 and 3. From the graph, we can easily conclude that the proposed method achieves a considerably better balance between SAR and WAR, compared to the existing method, and provides better performance in terms of Write throughput.

5. Conclusion

In this paper, a novel method for performing deduplication on LSM-tree family KV stores is introduced. First, the vulnerabilities of LSM-tree and some of the previous work was well examined. The proposed method focuses on the space amplification problem that has not been researched deeply so far. It is applied to the CaseDB architecture, which is one of the most recent and advanced KV stores. The deduplication extended compaction method focuses on eliminating space amplification as well as keeping the write-amplification balance. For LSM-tree, there is always a tradeoff between SAR and WAR, since it is almost impossible to keep both at low rates. In order to improve performance in terms of space amplification, WAR must be sacrificed, and vice versa. The introduced technique offers a novel solution to keep both in balance, reducing SAR without a considerable sacrifice in WAR. Moreover, it provides a flexible implementation using a threshold that can be changed according to system needs. Experiments prove that the proposed method eliminates the SAR for higher threshold values by keeping an acceptable balance with the WAR. This technique is applicable to applications with update-intensive workloads in big data.

Acknowledgement

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2018R1D1A1B07042602) and in part by Institute for Information & communications Technology Promotion (IITP) grants funded by the Korea government (MSIT): (No.2019-0-00064, Intelligent Mobile Edge Cloud Solution for Connected Cars, and No.2019-0-00240, Deep Partition-and-Merge: Merging and Splitting Deep Neural Networks on Smart Embedded Devices for Real-Time Inference).

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-Tree), " *Acta Informatica*, vol. 33, pp. 351-385, 1996. Article (CrossRef Link)

[2] K. Tulkinbekov, M. Pirahandeh and D. Kim, "CLeveldb: Coalesced Leveldb for Small Data, " *2019 Eleventh International Conference on Ubiquituos and Future Networks (ICUFN)*, Zagreb, Croatia, pp. 567-569, 2019. Article (CrossRef Link)

[3] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *Proc. ACM SIGMOND Int. Conf. Manage. Data*, pp. 217-228, 2012. Article (CrossRef Link)

[4] Casedb main page. [Online]. Available: https://github. com/iesl-lab-inha/CaseDB Source (GitHub Link)

[5] Leveldb main page. [Online]. Available: https://code. google. com/p/leveldb/, 2016. Source (GitHub Link)

[6] Rocksdb main page. [Online]. Available: http:// rocksdb.org/, 2016. Source (GitHub Link)

[7] NoSQL Wikipedia. [Online]. Available: https://en. wikipedia.org /wiki/NoSQL, 2019. Article (Website Link)

[8] G. DeCandia, et al., "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Prinsiples*, pp. 205-220, 2007. Article (CrossRef Link)

[9] W. Zhang, Y. Xu, Y. Li, Y. Zhang, and D. Li, "FlameDB: A Key-Value Store With Grouped Level Structure and Heteregeneous Bloom Filter," in *IEEE Access*, vol. 6, pp. 24962-24972, 2018. Article (CrossRef Link)

[10] K. Zang, J. Kim, and G. Cho, "An Efficient and Energy-saving Data Dissemination Mechanism for Low-power and Lossy Networks," J. IEIE Trans. on Smart Processing and Computing, vol. 7, no. 4, pp. 271-278, August 2018. Article (CrossRef Link)

[11] Y. Ting, W. Jiguang, H. Ping, H. Xubin, W. Fei and X. Changsheng, "Building Efficient Key-Value Stores via a Lightweight Compaction Tree," in *ACM*

Transactions on Storage, 13. 1-28. 10.1145/3139922, 2017. Article (CrossRef Link)

[12] X. Wu, Y. Xu, Z. Shao and S. Jiang, "LSM-Trie: An LSM-Tree based Ultra-Large Key-Value Store for Small Data Items,", in *USENIX Annual Technical Conference (USENIC ATC '15)*, Santa Clara, CA, USA, July 2015. Article (USENIX Link)

[13] O. Kaiyrakhmet, S. Lee, B. Nam, S. Noh, Y. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory" in *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, USA, February 2019. Article (USENIX Link)

[14] Big Data. [Online]. Available: https://www.techjury.net/stats-about/big-data-statistics/, 2019. Source (Website Link)

[15] W. Zang, and Y. Xu, "Deduplication Triggered Compaction for LSM-tree Based Key-Value Store," in *IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 1-4, Beijing, China, 2018. Article (CrossRef Link)

[16] C. Lai, et al., "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, pp. 1-14, 2015. Article (CrossRef Link)

[17] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Wisckey: Seperating keys from values in SSD-conscious storage," in *Proc. FAST*, pp. 133-148, 2016. Article (CrossRef Link)

[18] Bloom Filter. [Online]. Available: https://antognini.ch/papers/BloomFilters20080620.pdf/. 2008. Article (Website Link)

[19] db_bench. [Online]. Available: https://github.com/google/leveldb/blob/master/benchmarks/db_bench.cc, 2019. Source (GitHub Link)

[20] Apollo Dataset. [Online]. Available: https://data.apollo.auto/? locale=en-us&lang=en, 2019. Source (Apollo Link)

[21] J. H. Cha, and D. S Kim, "Design and Implementation of a Real-time Monitoring Tool for Data Distribution Service," J. IEIE Trans. on Smart Processing and Computing, vol. 7, no. 4, pp. 264-270, August 2018. Article (CrossRef Link)

[22] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Commun. ACM 33*, 6 June, 1990. Article (CrossRef Link)

[23] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 53-64, 2012. Article (CrossRef Link)

Khikmatullo Tulkinbekov received his bachelor's degree in information and communication engineering from Inha University in Tashkent, Uzbekistan, in 2018. He is currently pursuing his master's degree and is a research assistant in the Intelligent Embedded Systems Lab at Inha University in South Korea. His research interests include storage systems and intelligent cloud storage.

Joo-Hwan Kim received his bachelor's degree in electronic engineering from Inha University in Incheon, Korea, in 2019. He is currently pursuing his master's degree and is a research assistant in the Intelligent Embedded Systems Lab at Inha University in South Korea. His research interests include IoT cloud and edge computing.

Deok-Hwan Kim received a PhD in computer science from the Korean Advanced Institute of Science and Technology (KAIST) in 2003. He has been a professor at Inha University in South Korea since 2006. His research interests include storage systems, intelligent cloud storage, embedded and real-time systems, and biomedical systems. He has had more than 50 papers published in major journals and for international conferences.