IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 31, 2020, accepted August 6, 2020, date of publication August 14, 2020, date of current version August 25, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3016680

CaseDB: Lightweight Key-Value Store for Edge Computing Environment

KHIKMATULLO TULKINBEKOV AND DEOK-HWAN KIM, (Member, IEEE)
Department of Electronic Engineering, Inha University, Incheon 22211, South Korea

Corresponding author: Deok-Hwan Kim (deokhwan@inha.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant 2018R1D1A1B07042602, and in part by the Institute for Information and Communications Technology Promotion (IITP) grant funded by the Korea Government (MSIT) (Intelligent Mobile Edge Cloud Solution for Connected Car, Deep Partition-and-Merge: Merging and Splitting Deep Neural Networks on Smart Embedded Devices for Real-Time Inference) under Grant 2019-0-00064 and Grant 2019-0-00240.

ABSTRACT Key-value stores based on a log-structured merge (LSM) tree have emerged in big data systems because of their scalability and reliability. An LSM-tree offers a multilevel data structure with a simple interface. However, it performs file rewrites at the disk level, which causes write amplification. This study is concerned with this problem in relation to an embedded board environment, which can be used in edge computing. Addressing the major problems associated with an LSM-tree, we propose a new key-value store named CaseDB, which aggressively separates keys and bloom filters on the non-volatile memory express (NVMe) drive and stores the values on the SSD. Our solution reduces the I/O cost and enhances the overall performance in a cost-efficient manner. CaseDB employs a memory component, CBuffer, to avoid small write operations, and a delayed value compaction technique that guarantees the sorted order for both keys and values. CaseDB also utilizes deduction-based data deduplication to prevent space amplification in the values layer. The experiments show that CaseDB outperforms LevelDB and WiscKey 5.7 and 1.8 times, respectively, with respect to data writes, and additionally improves the read performance by 1.5 times. CaseDB also avoids the space amplification of WiscKey.

INDEX TERMS Key-value store, LSM-tree, NoSQL, write and space amplification, edge computing.

I. INTRODUCTION

In recent years, the rise in technological developments has resulted in a dramatic increase in the number of Internet-of-Things (IoT)-based devices. According to IBM reports, 90% of all the data generated thus far was created in only the last two years. Currently, one person generates approximately 1.5 MB of data every second [13], [14]. The growth in big data is already inevitable, and thus managing the data is becoming the most crucial problem. To date, data have been handled and shared through central data centers by providing cloud storage to users. However, continued exponential growth means the amount of data has become too large to be stored in these kinds of centralized servers, which are no longer able to fulfill current end-users' needs in terms of transmission speed and data consistency.

In this new era of big data, a new computing paradigm known as edge computing has been emerging as the solution that offers decentralized computing. This new paradigm brings data computation and storage closer to the location at which they are needed, to improve response times and save bandwidth. Apart from these benefits, edge computing has certain limitations in terms of the use of hardware resources in an embedded board environment. Moreover, developments in non-volatile memory (NVM) technologies have changed the idea of storage management over the decades. Particularly, new-generation SSDs, termed NVM express (NVMe) drives, have improved the data management speed dramatically. The throughput of an NVMe drive is six times higher than that of an SSD. However, this new technology comes with a higher cost such that it is quite challenging for big data management applications to afford NVMe drives. This means that, providing a high-performance database at an affordable price is one of the most challenging aspects of big data application development.

Thus far, key-value (KV) stores have been the most promising in big data systems as the representative of NoSQL [6] databases. These stores are mostly implemented using log-structured merge (LSM) trees [1] as the data structure. An LSM-tree uses a multilevel structure, and data are written

The associate editor coordinating the review of this manuscript and approving it for publication was Genoveffa Tortora.

in sorted order at each level. LevelDB [4], Dynamo [7], RocksDB [5], Cassandra [15] are the most well-known implementations of LSM-trees as KV stores. Because of its simplicity and availability, LevelDB has been the main focus of studies. This database keeps data sorted at each level and improves the read throughput using an LSM-tree. However, it performs a considerable amount of file rewrites in persistent storage and causes write amplification that is measured by its ratio (WAR). This problem considerably lowers the performance of embedded boards that lack connectivity with external hard drives. Several studies have been conducted to improve the performance of LevelDB [2], [8], [9], [11]. For example, WiscKey [19] separates the keys from their values and implements the LSM-tree only on the keys. This reduces the write amplification by additionally applying compaction to the keys. However, this method is not applicable to small write operations where the value size becomes smaller than the key size. Furthermore, WiscKey depends on a garbage collector (GC) to maintain the values in the sorted order and to remove deprecated data from the database. This causes space amplification in update-intensive workloads because the deprecated data are retained until garbage collection occurs.

In this article, we introduce CaseDB, which aggressively separates keys and bloom filters [21] from their values in the form of metadata and stores the metadata and value layers on NVMe drives and SSDs, respectively. CaseDB employs a delayed value compaction (DVC) process that performs two-step data merging to avoid redundant I/O on values. It first executes merging and sorting on metadata and merges string sorted tables (SSTables) only once in the value relocation step. Moreover, DVC continuously checks for data duplication on values and performs unscheduled compaction (if necessary) to avoid space amplification. Using a memory component coalescing buffer (CBuffer) [2], CaseDB avoids small writes and guarantees a low percentage of metadata share for all workloads. By introducing CaseDB, we claim four major contributions as mentioned below:

- Using NVMe drives and SSDs simultaneously in a cost-efficient manner
- Moving bloom filters to the metadata section of the SSTable with keys to improve the data scan performance
- DVC technique to maintain both keys and values in their sorted order
- Deduction-based data deduplication to avoid space amplification.

We used the LevelDB open-source code for implementing CaseDB because of its simplicity and availability. We evaluated our proposed solution by testing LevelDB, WiscKey, and CaseDB in the same hardware environments. The experimental results showed that CaseDB improves the write and read performance of LevelDB by 5.7 and 1.5 times, respectively. In addition, CaseDB outperforms WiscKey by 1.8 times in terms of data writes and avoids space amplification.

The remainder of the paper is structured as follows. In section II, we present the background work and motivations. Sections III and IV introduce our proposed method and the major differences with previous methods. Section V includes our experimental results and compares the results with those of previous studies. Finally, in sections VI and VII, we summarize related work and conclude the paper.

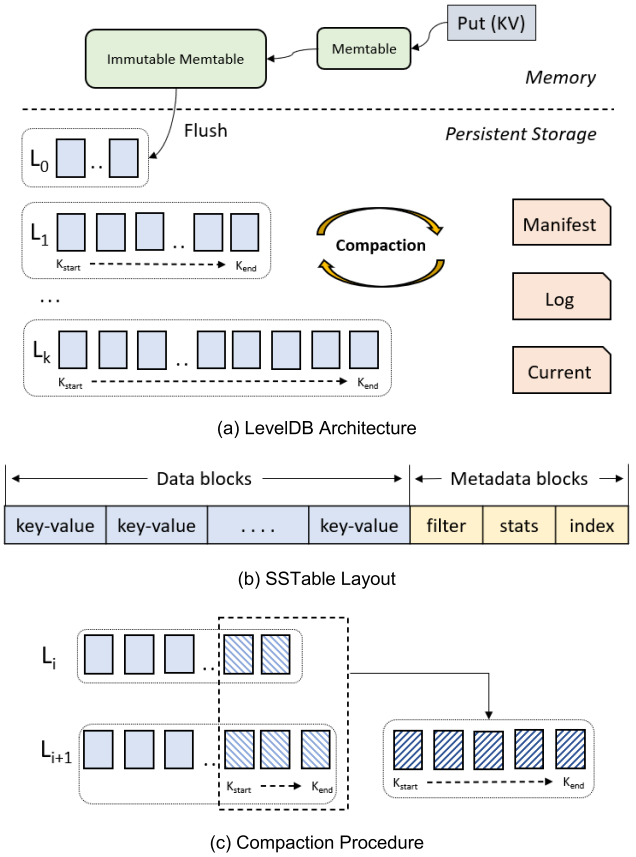


FIGURE 1. LevelDB architecture, SSTable layout, and compaction procedure in LevelDB.

II. BACKGROUND AND MOTIVATIONS

Over the years, LSM-trees have found use as one of the most common data structures with which to implement KV stores. LevelDB, from Google, is a prominent example of an LSM-tree that is used in real-life implementations of big data systems. It is open-source with clear and straightforward explanations of its architecture. Moreover, LevelDB is one of the most well-known KV stores with competitive performance. WiscKey extends LevelDB with its key-value separation technique and has become highly respected among LevelDB-based KV stores. Thus, in this study, we used the source code of LevelDB for CaseDB and compared its performance with that of both LevelDB and WiscKey.

A. LevelDB

LevelDB [4] is a state-of-the-art store and is based on Google's BigTable, which is distinguished from other families of LSM-trees by its simplicity. A general overview of the LevelDB architecture is shown in Fig. 1. LevelDB implements multiple memory and disk components. The write process starts with buffering new items in the Memtable, a memory component based on a skip list [35] into which

each item is inserted in sorted order. KV items are also logged in the write-ahead-log file for recovery purposes in the case of system failure. Once the size of Memtable reaches the threshold, it is moved to the read-only (Immutable) Memtable queue, where the data are flushed to persistent storage. The flushing step writes the sorted items into same-sized files i.e., SSTables, as shown in Fig. 1 (a). The layout of these tables, shown in Fig. 1 (b), is designed to hold sequentially written data blocks in sorted order. Following the data blocks, the filter, stats, and index blocks are written to the SSTable as metadata relating to the file. LevelDB implements a multilevel structure in persistent storage constructed of SSTables. Each lower level becomes larger than the previous one. When the data are flushed from memory, the newly created SSTables are written to L_0. Finally, the compaction process merges and sorts SSTables into lower levels as shown in Fig. 1 (c). The compaction process finalizes the write operation into LevelDB and is called whenever the level size reaches the maximum size limit. For example, the default level size for L_1 is 10 MB in LevelDB and each subsequent level is 10 times larger. Once the size of L_1 reaches 10 MB, the compaction process selects certain key ranges from L_1 and L_2 and reads all SSTables of the selected range to memory. After the data are merged and resorted, new SSTables are created at the lower level and are replaced with old ones. This process ensures that each KV item is moved to lower levels in the sorted order. After compaction, a key range that overlaps between two SSTables within one level does not exist.

Operations to read data from LevelDB are performed in hierarchical order from top to bottom. Initially, the Memtable and Immutable Memtables are searched for an item. If it is found using one of the memory components, the disk is not touched, and the item is returned from memory. Otherwise, each level in persistent storage is perused on a one-by-one basis starting from the highest. Owing to the compaction process, only one SSTable is searched on each level because the data are sorted level-base. When searching data from the SSTable, bloom filters [21] play a crucial role. Each SSTable includes a bloom filter, which is initialized while writing data into the SSTable and it is embedded to file as a filter block. The bloom filter requires the key as parameter and returns two possible outcomes (*maybe* and *no*) after performing several hashing functions on the key. If the bloom filter returns *no*, it guarantees the KV item is not available inside the SSTable and the read process continues at the next level. If the bloom filter returns *maybe*, the item is possibly available inside the SSTable, in which case the data blocks are searched. However, the possibility also exists that the item is unavailable even after the bloom filter returned a positive response, and this is addressed as the *false positive rate* of the bloom filter.

B. WiscKey

WiscKey [19] was developed on the basis of LevelDB and focuses on solving the write amplification problem. WiscKey inherited all the memory components from LevelDB but separates the keys from their values in persistent storage. In other words, when the data are flushed from memory to disk, the keys are separated from their values and inserted to the LSM-tree, as shown in Fig. 2. Each key is saved with reference to its corresponding value. The values are written sequentially into the Value Log (vLog) file and managed independently. Compaction is only applied to the keys, and when the data are deleted, only the key is removed from the LSM-tree and the actual value is retained in the vLog file. To remove the deprecated data from vLog, WiscKey employs GCs. Garbage collection, which occurs at fixed time intervals, removes all values that are not referenced by a key in the LSM-tree and then resorts the vLog file. WiscKey reduces the write amplification considerably by running compaction only on keys, which constitute a small proportion of the total amount of data. This means that the size of the LSM-tree of WiscKey is considerably smaller than that of LevelDB.

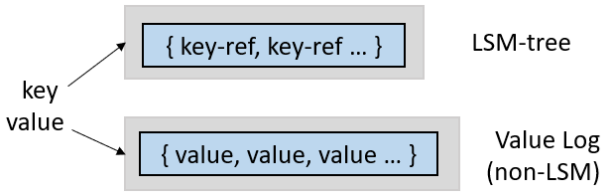


FIGURE 2. WiscKey key-value separation technique.

Similar to LevelDB, the hierarchical order of the read process in WiscKey is top to bottom. As the memory components are the same for both, the read process starts with Memtables. If the item is not found in memory, WiscKey searches the part of the LSM-tree in persistent storage. Because the LSM-tree of WiscKey does not contain values, only the keys are scanned. If the searching key is found in the LSM-tree, the value is retrieved from the vLog file.

C. DRAWBACKS

In section II-A and II-B, we discussed the LevelDB and WiscKey architecture and their differences in terms of the LSM-tree. As mentioned above, all SSTables at the upper levels are moved to lower levels during the compaction process. This results in repeated reading and writing of the files until they reach the lowest level. In other words, LevelDB is adversely affected by the write amplification problem, i.e., the number of file rewrites during the insertion process. In the default situation, each next level is ten times larger than the previous one in LevelDB; thus, at least 11 SSTables would have to be relocated to compact one SSTable from the upper level. In a computing environment with resource-limited edge embedded devices, write amplification is unacceptable as it reduces the data management performance dramatically. For example, one of the latest embedded boards, the Jetson AGX Xavier, includes an 8-core ARM v8 CPU and 16 GB of RAM, as shown in Table 1. Moreover, it includes only one M.2 slot for an NVMe SSD but does not include on-board Peripheral Component Interconnect (PCI) expansion slots;

TABLE 1. Configurations of server and embedded board.

Category	Jetson AGX Xavier Embedded Board
CPU	4*ARMv8 Processor
DRAM	16GB
SSD	Samsung SSD 860 EVO / 500GB
NVMe	Samsung SSD 970 EVO NVMe M.2 / 500GB
OS	Ubuntu 18.04 LTS

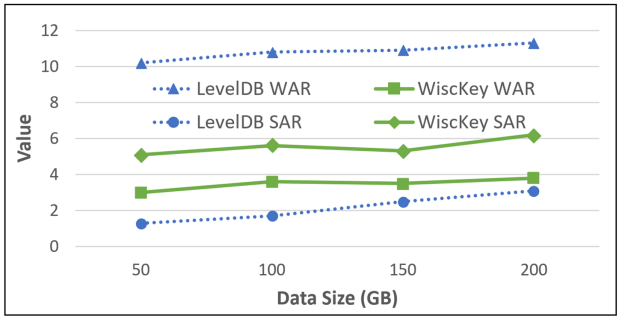


FIGURE 3. Write and space amplification of LevelDB and WiscKey on embedded board with 90% update share.

thus, a connector cable is required to attach the SSD. This considerably slows the speed of file read/write operations on external hard drives.

WiscKey solves the above-mentioned write amplification problem. However, deleting the actual value from the vLog file is highly dependent on the GC frequency. Increasing this frequency is not good practice when it results higher WAR. In workloads that are update intensive, this vulnerability of WiscKey causes a space amplification problem in that deprecated data are retained until they are removed by the GC. Therefore, the actual size of the database exceeds that of the workload during the write process, and is visualized by the space amplification ratio (SAR). Another drawback of WiscKey is that the vLog file does not guarantee maintaining the values in sorted order. This problem is graphically explained in Fig. 3 in which we present the results of evaluating the write and space amplification with a 90% update share for different amounts of data. These results clearly show that LevelDB experiences write amplification with WAR values higher than 10. Although WiscKey avoids this problem, it faces a new problem related to space amplification in update-intensive workloads. Moreover, WiscKey is unable to maintain the order in which values are sorted using vLog. Even though GC rearranges the values after removing deprecated data, the new values flushed from memory are inserted at the end of the vLog file. This prevents sequential data from being directly accessed when sequential reads are performed.

CaseDB was developed by considering the strengths and weaknesses of both LevelDB and WiscKey and it fully adapts to an embedded board environment by eliminating the above-mentioned problems.

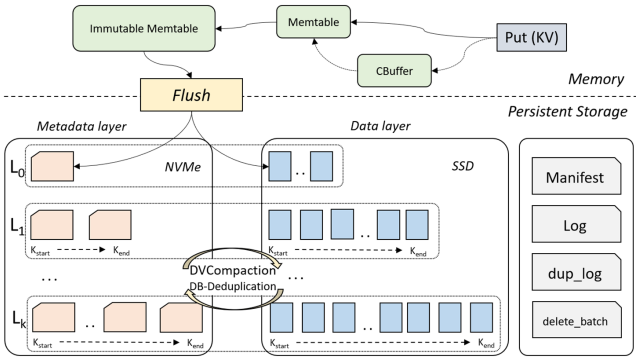


FIGURE 4. CaseDB architecture.

III. CASEDB ARCHITECTURE

CaseDB solves the aforementioned problems commonly encountered in LevelDB and WiscKey. A general overview of the proposed method is shown in Fig. 4. First, CaseDB avoids small data writes with the help of a memory component named CBuffer. In LevelDB, the SSTable includes both metadata and data, and the entire file is read to memory for compaction. However, most logical and arithmetical operations that are run on keys and values are read simply to update their location. Based on this scenario, we separated the keys as metadata from the SSTables and redesigned the compaction process to delay the merging of the SSTables. CaseDB stores the metadata on NVMe SSDs and the actual values on external SSDs. This dramatically reduces the number of reads and writes to the SSD, the interface of which is not directly connected to the embedded board on the Jetson AGX Xavier. We also moved the bloom filters in the metadata files to improve the data scanning performance without accessing the values. Moreover, the metadata only comprise a small portion of all the data, and the compaction allows for comparably less I/O than in LevelDB. Because the proportion of metadata is relatively small, the NVMe drive holds only the metadata that correspond to an amount of data that is several hundred times larger in size. This approach enables us to easily handle big data in a cost-efficient way. Apart from WiscKey, CaseDB employs a DVC technique and data deduplication on values. This guarantees the sorted order of values and avoids space amplification. In simpler words, CaseDB offers these four major advantages over LevelDB and WiscKey:

A. Buffer coalescence to avoid small writes
B. Metadata separation to reduce write amplification
C. DVC for sorting the values
D. Data deduplication to avoid space amplification

A. COALESCING BUFFER

When separating the keys from their values, the most notable challenge is to consider the size of the value. Each separated key includes a reference to the actual location of the corresponding value with the aim of reducing the data size for compaction. The question is: what if the size of the value decreases considerably to become even smaller than that of its key reference? Investigations showed that IoT

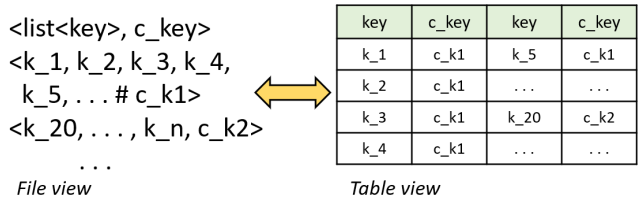


FIGURE 5. UMeta file layout in CaseDB.

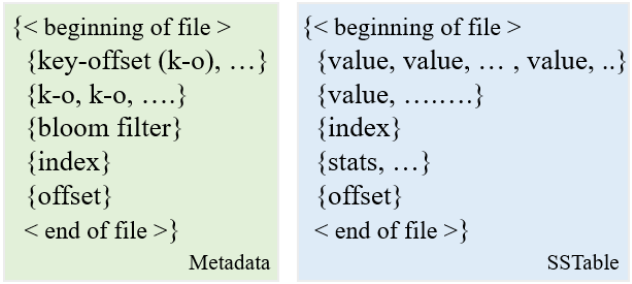
devices transfer different kinds of data, including large-sized images or videos or sensor data of an extremely small size [29]. Our solution to this problem was to introduce CBuffer, which is deployed as the memory component of CaseDB. CBuffer is a skip-list-based arbitrary memory buffer, which loads KV items and merges them to a specific size. All the merged data are logged in a UMeta file, as shown in Fig. 5, to allow reads from coalesced data. In other words, CBuffer ensures that all data at least exceed a specific size. It guarantees that the metadata separation method functions perfectly in all cases.

CaseDB uses a threshold value to check whether data are to be merged or inserted into the database directly. If the size of the data size exceeds the threshold value, the data are directly inserted into Memtable, and no space is allocated in CBuffer. However, if the data size is smaller than the threshold value, the data are inserted into CBuffer and delayed until the size of CBuffer reaches the minimum possible size to merge and generate one single KV item for the group. CBuffer groups the small key-value items in sequential order to generate bigger value. The merged data are referred to as a single value for the future insertion process. After merging the item, CaseDB generates a unique key for the coalesced group and creates logs in the UMeta file, which are then inserted into Memtable. The UMeta keeps records in logging manner that includes the list of coalesced keys followed by the group key as shown in Fig. 5. The write and read process using CBuffer and UMeta is discussed in detail in the next sections.

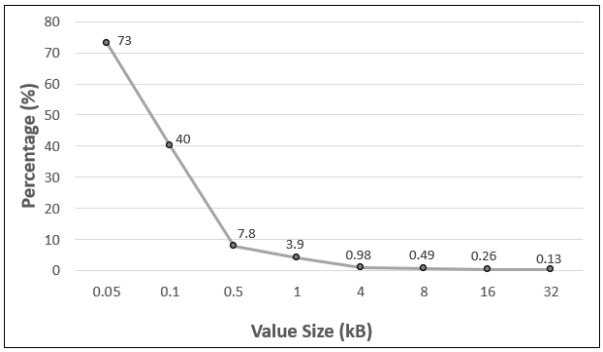
B. METADATA SHARE

During the flushing step, each key is separated with reference to the corresponding value and bloom filter. The file structure is motivated by that of the LevelDB SSTable layout, as shown in Fig. 6 (a). The metadata size is fixed where key-offset pairs are written in sorted order as byte-stream blocks at the beginning of the file. Following these blocks, the bloom filter and index blocks are written. Finally, the offset keeps a reference to the index and filter blocks at the end of the file. Similarly, the CaseDB SSTable resembles that of LevelDB, but it does not include keys in data blocks and does not store the bloom filter in the metadata blocks. Moreover, CaseDB also inherits the *varint64* data types from LevelDB for storing keys and offsets. In the default configuration, 16 and 20 bytes are allocated for the key and offset, respectively. This can be changed automatically during runtime if the size of the database increases.

$$\mathbf{M} = \mathbf{B(n)} + \mathbf{n}^{*}(\mathbf{K} + \mathbf{O}) \tag{1}$$



(a) Metadata and SSTable layout



(b) Metadata share for different value sizes

FIGURE 6. Metadata and SSTable layout in CaseDB and metadata share.

The metadata size can be calculated as in (1), where M indicates the size of the metadata, $B(n)$ is the size of the bloom filter, K is the size of the key, O is the offset size, and n is the number of entries for this metadata file. The value of B depends on n, and thus the variable that largely affects the size of the metadata file is n. Using the default configuration, we can calculate the size of the metadata by assigning different values to n and checking its proportion in the SSTables by altering the size of the value. As shown in Fig. 6 (b), the metadata share depends on the size of the value. The figure shows that the metadata share is considerably high for small-sized values; specifically, this share is 73% when the size of the value is 50 bytes. This percentage decreases as the size of the value increases. Moreover, the metadata share decreases linearly as the size of the value increases. This portion decreases to approximately 0.25% for 32-KB-sized values, and could be expected to become negligible as the size of the value continues to increase. We selected 4 KB as the default configuration for the smallest value threshold as the proportion of metadata is relatively tiny (∼1%).

C. DELAYED VALUE COMPACTION

The data insertion process in CaseDB starts by checking the size and by separating the values into those with small and ordinary sizes. The small values are inserted into CBuffer, where they remain until the size of CBuffer reaches the threshold. The ordinary-sized KV items are directly inserted into Memtable, which is converted into read-only when it fills up, whereupon a new Memtable is created. Data from the Immutable Memtable is then flushed into disk components. In this step, metadata containing keys and a bloom

Algorithm 1

```
1:  function DVCompactTion(sizeCheck)
2:
3:  procedure
4:      if needsCompaction == true then
5:          //Compaction Part 1
6:          while done == false do
7:              c ← Compaction()
8:              c.layer ← SelectMetadataLayer()
9:              c.l, c.s, c.e ← CompactRange(sizeCheck)
10:             if c.l ≠ null then
11:                 DoCompactionWork(c)
12:             else
13:                 done ← true
14:                 CleanupMetaCompaction()
15:
16:         //Compaction Part 2
17:         PrepareMetadata(c)
18:         c.layer ← SelectSSTableLayer()
19:         c.relocation ← true
20:         DoCompactionWork(c)
21:         CleanupSSTableCompaction()
22:         c.ReleaseInputs()
23:         DeleteObsoleteFiles()
24:
25:     //Deduplication part here
```

FIGURE 7. Delayed value compaction procedure.

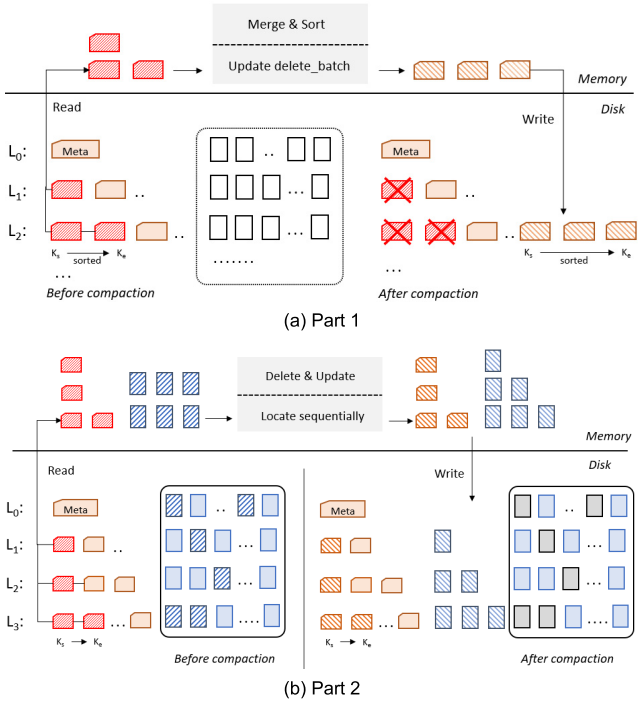


FIGURE 8. Delayed value compaction in CaseDB.

filter are flushed to a metadata layer, and this is followed by compaction (DVC). In contrast, SSTables are flushed to the default level, where they remain until the DVC process takes place in the next step. This compaction consists of two parts as shown in Algorithm 1 in Fig. 7. The algorithm requires a parameter named *sizeCheck*. This parameter is used for deduplication purposes, as discussed in section III-D. Only the metadata layer participates in the DVC part 1, as shown in line 8. The *CompactRange()* method checks for levels where the compaction needs to return the assigned values for the level (*c.l*) and key range (*c.s* and *c.e*), as shown in line 9. The *DoCompactionWork()* method performs a merge and sort operation on the metadata files in selected key ranges, as shown in Fig. 8 (a). In this step, deprecated keys are inserted into the *delete_batch* file; hence, the old values are deleted in the next part. *CompactRange()* returns null if compaction is unnecessary and the DVC part 1 terminates. In the second part of DVC, updated metadata files and corresponding SSTables are selected, as shown in lines 17 and 18. The number of metadata files are determined by the *PrepareMetadata()* method depending on the number of SSTables they refer. After enabling the *relocation* mode, the *DoCompactionWork()* method then performs value relocation on the SSTables, as shown in Fig. 8 (b). As shown in Algorithm 1, DVC accesses the metadata layer several times during part 1, but the data layer is accessed only once to relocate the values in the sorted order SSTables in each level. This technique significantly reduces redundant disk accesses during compaction. The final step of DVC is deduplication checkout, which is discussed in section III-D.

D. DATA DEDUPLICATION

Data duplication is a common problem for all KV stores. Because KV stores perform update operations by inserting the same key with the new value, the possibility exists that multiple same keys could have different values. The LSM-tree is designed to overcome this problem by storing data on multiple levels to perform compaction over the levels. This process determines whether duplicate keys exist in different levels and removes the old key. Thus, the newest KV items are always stored on the highest levels. However, compaction occurs only if the level size reaches the limit. In the LSM-tree design, each next level is larger than the previous one. For example, each LevelDB level is ten times larger the previous level. This means that the compaction frequency becomes ten times smaller for each successive level. In big data systems with workloads of several terabytes in size, compaction becomes a rare process for the lowest levels, resulting in the increase of duplication in runtime. Moreover, WiscKey does not retain the multilevel structure for values. The GC is responsible for removing updated values from the vLog file. However, it does not always keep track of the amount of duplicated data because it runs at fixed intervals. This causes the space amplification problem in update-intensive workloads. CaseDB addresses this problem by employing a deduction-based data deduplication technique, which runs unscheduled compaction for the lower levels to remove duplicated items.

Algorithm 2 in Fig. 9 shows the pseudocode for deduplication checkout of DVC after part 1 and 2. It is based on the probability of the workload having a larger update share. After completing DVC for both keys and values, CaseDB calculates the number of compacted (*COMP_KEYS*) and duplicated (*DUP_KEYS*) keys, as shown in lines 11 and 12. This information can easily be retrieved from the metadata and *delete_batch* files. The value of

```
Algorithm 2
function DVCompacttion(sizeCheck)

procedure
    //Compaction Part 1
    ...

    //Compaction Part 2
    ...

    //Deduplication starts here
    COMP_KEYS ← GetCompactedKeys(metadata)
    DUP_KEYS ← GetDupKeys(delete_batch)
    DUP_RATIO ← COMP_KEYS / DUP_KEYS
    if DUP_RATIO ≤ DEDUP_THRESHOLD then
        needsCompaction ← true
        DVCompacttion(false)
    //Deduplication ends here
```

FIGURE 9. Data deduplication procedure in DVC.

DUP_RATIO is calculated by dividing the compacted keys by duplicated keys and is used to determine the possibility of duplication in the database, with a lower value indicating that the possibility of duplication is significant. Thus, for a ratio lower than the threshold, the compaction process is revoked by setting the *sizeCheck* flag to *false*. In this case, the *CompactRange()* method runs compaction until the lowest level even though the size of the level does not reach the limit. In summary, CaseDB employs a deduction-based deduplication algorithm to perform unscheduled compaction for lower levels. This means that, if a high degree of duplication exists at the upper levels, this would also be possible at the lower levels; thus, it would be worth running an unscheduled compaction.

E. DATA READS

The read procedure of CaseDB is similar to that in WiscKey with certain exceptions. Retrieving data from CaseDB starts with scanning CBuffer, followed by the active and immutable Memtables in memory in this order. In case the item is not found in memory, the UMeta file is scanned from persistent storage before accessing the metadata layer. If the item being scanned is small and already flushed to disk, its key is available in the UMeta file and the group key can be found there. The group key is then assigned as the scanning key to retrieve the coalesced value from CaseDB. In case the key is not found in UMeta, the search process simply continues by scanning the metadata layer. In this scenario, the key can easily be scanned from metadata for three major reasons: the size of the LSM-tree in the metadata layer is considerably small, the bloom filters improve the scan performance, and the metadata layer is stored on the NVMe drive. The process of scanning the metadata layer with bloom filters is similar to multilevel data scanning in LevelDB. If the key is found in the metadata layer, it references the corresponding SSTable in the data layer. Finally, the actual value can be retrieved by a single SSTable access.

CaseDB employs one additional step for data reads while employing the CBuffer and UMeta components. Although this step might cause a delay, the bloom filters in the metadata layer cover the time delay to access the UMeta file. Moreover, the LSM-tree and the order in which the values are sorted are guaranteed in the data layer, which improves the range query performance significantly. CaseDB decreases the number of SSD accesses not only for write operations but also for reading, which makes it the most flexible KV store for embedded environments.

TABLE 2. Comparison of LevelDB, WiscKey, and CaseDB.

Category	LevelDB	WiscKey	CaseDB
LSM-tree	Yes	Only keys	Yes
Bloom filter	Yes	No	Yes
Small writes avoidance	No	No	Yes
WAR	High	Low	Low
SAR	Low	High	Low
Duplication	Moderate	High	Low
Merging	Compaction	GC	DVC
Sorting	Yes	Only keys	Yes
NVMe adapted	No	No	Yes

IV. COMPARISONS

In this section, we detail the major differences among CaseDB, LevelDB, and WiscKey. Both WiscKey and CaseDB were developed on the basis of LevelDB and use a similar technique to separate the keys from their values. Table 2 presents a brief overview of the main comparison factors. First, LevelDB stores KV pairs in SSTables and employ an LSM-tree for managing the database, whereas WiscKey only uses key compaction to reduce the write amplification and fails to sort the values. On the other hand, CaseDB uses compaction for both the keys and the values with the help of DVC. Moreover, CaseDB KV separation differs from that of WiscKey by moving the bloom filter to the metadata layer with the keys. Next, CaseDB uses CBuffer to merge the small-sized KV items before flushing them into permanent storage. This eliminates the problem associated with managing small amounts of data and ensures that CaseDB is a compatible KV store for storing sensor data.

The most common and largest problems with all the KV stores in the LSM-tree family are write and space amplification. As discussed in section II, LevelDB experiences a high degree of WAR because it employs extensive compaction of the SSTables. WiscKey solves the write amplification problem by diminishing the size of the LSM-tree dramatically but fails to perform value updates. For update-oriented systems, duplication is a common problem that could be alleviated by removing old values from the database before they cause space leakage and unexpected system failure. WiscKey runs scheduled garbage collection that cannot track the update proportion and fails in terms of storage utilization

during runtime. They offer a tradeoff by allowing the user to choose between write or space efficiency at the time of system deployment. On the other hand, CaseDB efficiently decreases both the WAR and SAR and achieves a significant advantage over both of the other databases.

Reading from the database is the next most crucial operation for all systems. LevelDB provides reliable read performance because the data are merged and sorted continuously. WiscKey only sorts the keys, while the values are managed in the single vLog file. WiscKey gains a promising latency on random reads by employing parallel seeking over vLog, but it cannot stand on sequential reads. CaseDB also addresses this problem and guarantees the sorted values as in LevelDB. The last but not the least notable difference of CaseDB is its NVMe adapted architecture designed to leverage the storage resources. In the default configuration, metadata comprises as much as 1% of all the data in size. Thus, the amount of data CaseDB manages is hundreds of times larger, and these data are stored in external SSDs, separate from the metadata layer stored in an on-board single NVMe drive in the Jetson AGX Xavier environment. Neither LevelDB nor WiscKey includes the configuration for managing the database on different drives. In this regard CaseDB therefore offers a considerable advantage for adaptation in different environments.

Considering the adaptability and efficiency of different workloads and hardware environments, as mentioned above, CaseDB provides the most adaptable KV store for edge computing.

V. EVALUATIONS

In this section, we discuss the extensive experiments that were conducted to evaluate the performance of LevelDB [4], WiscKey [19], and CaseDB. As discussed previously, LevelDB is representative of the family of LSM-trees developed by Google. In our experiments, we used version 1.20. WiscKey is an existing KV store that was built on LevelDB by separating the keys from their values. Compared to CaseDB, WiscKey retains the actual values in the vLog file, which is updated using GCs. We examined the impact of metadata separation and the effect of the compaction of values on the read and write throughput by enabling and disabling CBuffer. In addition, we compared the space efficiency of CaseDB with that of WiscKey.

For our evaluations, we used the data generated by the db_bench [28] benchmarking tool, which is most commonly used to evaluate KV stores. We adopted the db_bench for CaseDB and included new flags to enable or disable CBuffer and altered the thresholds for deduplication. This allows us to understand the impact of CBuffer more clearly. The default configurations for LevelDB were retained. Moreover, we ran experiments on the YCSB [26] macrobenchmark, which provides different combinations of workloads. Our target platform is a Jetson AGX Xavier embedded board with limited hardware resources, as specified in Table 1. We set the default threshold value of CBuffer to 4 KB to avoid memory leakage problems due to large values and to allow CaseDB to perform competitively. For the comparison, we used one million entries with different and increasing value sizes.

A. WRITE THROUGHPUT

Evaluation of the write performance of LevelDB, WiscKey, and CaseDB would enable us to easily analyze the effect of the metadata separation technique on the overall performance. We used one million entries and set the initial value size at 50 bytes. Then, we increased the size of the value to 64 kB. The results of our evaluation of the write performance are shown in Fig. 10 (a) and (b). The figure shows that CaseDB achieves a considerably high throughput for sequential write operations over LevelDB not only for large values but also for small values. In addition, the sequential write performance of CaseDB is almost stable at 40–45 MB/s for values up to 4 KB, whereas this rate steadily increases for LevelDB from 0.5–10 MB/s. The performance of WiscKey at approximately 70 MB/s is comparable on larger values of 64 KB. However, this performance is even lower than that of LevelDB for small values, that is, 0.3 MB/s, which proves that WiscKey is not compatible with an environment characterized by small writes. Moreover, overall, the throughputs of WiscKey for both sequential and random writes are lower than those of CaseDB even if they both run compaction on keys. This is because the WiscKey architecture is not adapted for using a combination of NVMe drives and SSDs, indicating that experiments would need to be performed on a single drive. As the value size increases, the metadata share decreases, and the performance of CaseDB increases to 83 MB/s for 64-KB values. We observed a similar result for random writes. It should be noted that the random write performance of LevelDB on an embedded board is considerably small and only increases slightly for large values from 0.5 MB/s at 50 bytes to 4.2 MB/s at 64 KB. In contrast, the average performance of CaseDB is 4.5 and 1.8 times faster than that of LevelDB and WiscKey, respectively, for values that exceed the threshold. These experiments show that the separation of metadata and the delay of SSTable compaction by using the metadata have a considerable effect on the write throughput and achieve an improvement of approximately 5.7 and 1.8 times over LevelDB and WiscKey, respectively, by reducing the I/O proportion and by using NVMe-adapted architecture.

B. CBuffer EFFECT

We conducted further experiments on CaseDB by enabling and disabling CBuffer to understand the role of this buffer. CBuffer is active only in those cases in which the size of the inserted data is smaller than the threshold value. In our evaluation, we set the threshold at 4 KB, and thus we only need to test the CaseDB performance for values up to 4 KB. For values exceeding 4 KB, CBuffer does not play a role and does not affect the performance of CaseDB. The experimental results corresponding to the effect of CBuffer are shown in Fig. 10 (c). For this evaluation, we varied the

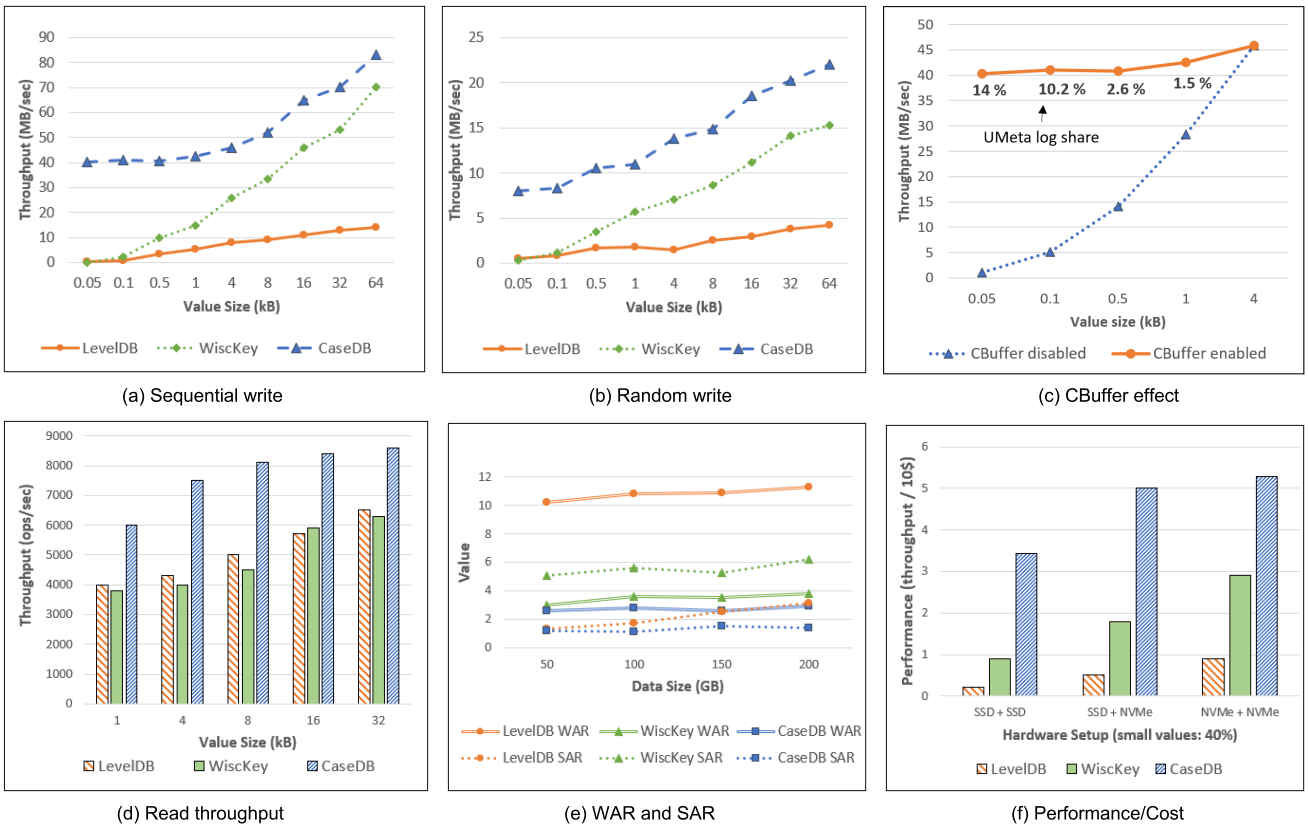


FIGURE 10. Performance evaluations of LevelDB, WiscKey, and CaseDB on Jetson AGX Xavier.

size of the value from 5 bytes to 4 KB. As shown in the figure, the performance of CaseDB is considerably low when CBuffer is disabled for small-sized items. For values sized 50 bytes, the performance is 1.1 MB/s, which increases as the size of the value increases. In contrast, when CBuffer is enabled, CaseDB achieves steady performance at approximately 40 MB/s for all values. The figure also shows the UMeta file share after compression over the workload for different values. It shows the coalescing log size gets bigger for small values where it is 14% for 50 bytes. That rate is still acceptable when WiscKey value referencing technique almost doubles the database size for small values. Moreover, the UMeta log share decreases dramatically to 1.5 % as the value size increases. The performance of CaseDB differs hugely depending on whether CBuffer is enabled or disabled. This difference is attributable to the proportion of metadata in relation to the data size. For small-sized values, the metadata forms a large share of the data and CaseDB starts to perform more I/O, similar to that by LevelDB. As the size of the value decreases, CaseDB stops separating metadata, and loses this advantage over LevelDB. Therefore, CBuffer is an important component of CaseDB and is essential for optimal database performance. In the case of workloads guaranteed not to include small-sized data, CBuffer can easily be disabled using the flags. The figure shows that for 4-KB values, CBuffer does not affect the performance. Thus, we can assume that it is always best practice to allow a certain amount of space for CBuffer in memory, which is significantly advantageous for small-sized values.

C. READ THROUGHPUT

In general, LevelDB achieves a significantly high read performance by providing sorted data across the levels. WiscKey, on the other hand, maintains the read throughput of LevelDB even after separating the values to the vLog file and applying parallel seeks for random reads. In addition, CaseDB improves the read performance with the help of NVMe drives and by moving the bloom filters to the metadata layer. Because the I/O speed is six times higher on NVMe drives compared to SSD, data scanning on the metadata layer is considerably faster on CaseDB compared to LevelDB and WiscKey. Fig. 10 (d) shows the evaluation results of read performance in the same hardware environment. The results show that LevelDB and WiscKey achieve almost similar throughput. WiscKey is slightly slower than LevelDB on data reads because it fails to perform sequential reads on the vLog file, but still performs very well with the help of parallel data seeks. The figure shows CaseDB performs approximately 1.5 times faster than the others. As mentioned earlier, data scanning on NVMe is considerably faster on CaseDB, and the use of bloom filters avoids unnecessary disk accesses to SSD for data retrieval. Nonetheless, the actual data is stored on external SSDs, which requires low-speed I/O on embedded boards. Thus, CaseDB achieves an improvement

of only 1.5 times on data reads with limited hardware resources.

D. WAR AND SAR

Clearly, for all families of LSM-trees, write amplification is the most common problem. CaseDB is designed to solve this problem and ensures the availability of sorted located data with low-cost I/O. Fig. 10 (e) compares the write and space amplification of LevelDB, WiscKey, and CaseDB for mixed workloads. As shown, WAR increases from 10 to 12 as the data size increases up to 200 GB on LevelDB. WiscKey and CaseDB offer a technique to reduce WAR by using data compaction by separating the keys. The figure shows the effectiveness of this method on values between 2 and 4. In the case of WiscKey, the WAR is slightly higher than for CaseDB because of scheduled garbage collection. Even if WiscKey does not perform compaction on values, the GCs rewrite the entire vLog file at specified intervals at the cost of additional I/O. On the other hand, DVC does not depend on the time and compaction is scheduled only when it is needed, hence the I/O is proportionally lower compared to WiscKey.

Space amplification was measured by increasing the update share to 90% for all KV stores in the same hardware environment. Extensive compaction of the SSTables prevents space amplification on LevelDB; however, the possibility of deprecated data existing on lower levels still exists when the update share increases. Our experiments showed that the SAR of LevelDB reaches 3.3 with a 200-GB workload. This value is higher for WiscKey because of its dependence on interval-based GC and inability to track data duplication on the SSD. WiscKey has SAR values of 4–6 for workloads of different sizes. The deduction-based deduplication technique of CaseDB avoids space amplification by running unscheduled compaction if the probability of duplication increases. For this evaluation, we selected the deduplication threshold as 100, implying that compaction is scheduled if the probability of duplication is more than 1%. This threshold fully keeps track of the update proportion and avoids compromising space efficiency.

E. PERFORMANCE/COST

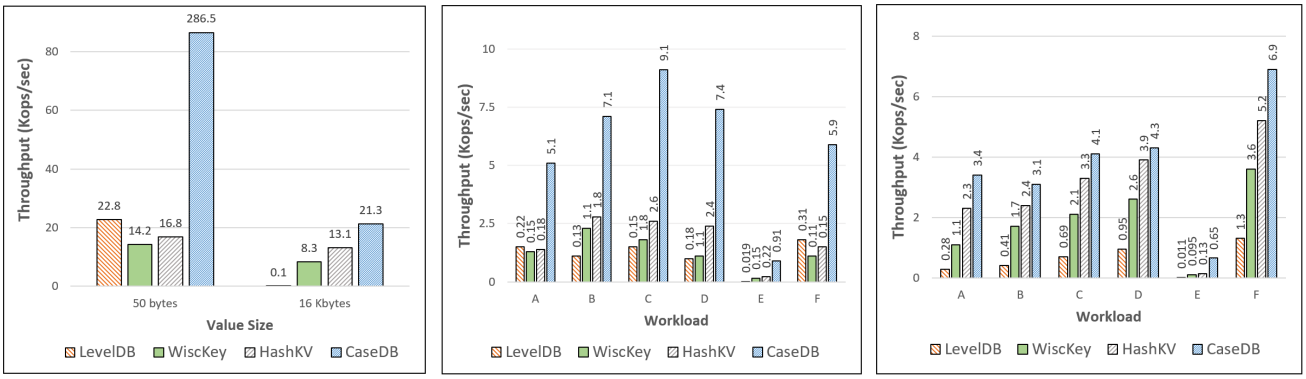
Another major evaluation factor for CaseDB is the applied environment and the tradeoff between performance and cost. Fig. 10 (f) shows the averaged throughput per $10 cost in an embedded environment for three KV stores when the small value load percentage is 40%. As the embedded environment is the same for all the databases, the main factor affecting the cost is the disk drives. We evaluated the performance for different combinations of SSD and NVMe drives. According to the description of the Jetson AGX Xavier board, only one drive can be attached to the on-board M.2 interface. An additional drive would have to be attached to the board using a connector cable, which would reduce the I/O performance. As discussed above, a feature of CaseDB is that it can access two different drives at the same time whereas the other two databases cannot. We therefore evaluated the performance of LevelDB and WiscKey on two drives in sequential order and calculated the average. The results in Fig. 10 (f) show that the performance of LevelDB and WiscKey increases linearly depending on the NVMe performance. On the other hand, CaseDB achieves promising throughput of 3.5 per $10 even without an NVMe drive. This performance was achieved by loading the metadata layer on the on-board SSD and by storing the actual values on the external SSD. Most I/O operations are performed on the M.2 interface and the number of accesses to the external drive is considerably reduced. Replacing the on-board SSD with an NVMe drive has a considerable effect on the performance of CaseDB in that the rate improves from 3.5 to 5. Moreover, replacing the external SSD with an NVMe drive does not have a considerable effect on the CaseDB throughput because most of the I/O is performed on the on-board drive.

F. YCSB BENCHMARK

YCSB [26] is the most well-known benchmarking tool for the evaluation of databases and KV stores. It provides six default workloads for running the evaluation under different criteria. We conducted experiments with YCSB on LevelDB, WiscKey, HashKV [30], and CaseDB. HashKV is a recent KV store and is based on the idea of WiscKey, namely by implementing key-value separation. HashKV uses a hash indexing method on multi-sector units to update the values. The results of YCSB evaluations are shown in Fig. 11. Extensive experiments were carried out on values of 50 bytes and 16 KB to compare the performance of the databases with both small and large values. Fig. 11 (a) and (b) show that CaseDB outperforms the other databases significantly when processing small values in that it performs approximately 17 times more operations per second than the other three databases. Moreover, CaseDB considerably outperforms the other KV stores for all workloads. Specifically, CaseDB performs an average of 30 times more operations than LevelDB, 16 times more than WiscKey, and 12 times more than HashKV. Moreover, in insertion and update intensive cases, LevelDB outperforms WiscKey and HashKV. This clearly proves that WiscKey and HashKV are not designed for small write workloads and that CaseDB solves this problem by employing CBuffer as the memory component.

As the size of the value increases, WiscKey and HashKV perform significantly faster, but CaseDB still has the highest throughput because of its flexibility in the embedded board environment. Fig. 11 (c) shows that WiscKey and HashKV outperform LevelDB for a value size of 16 KB because they employ KV separation. In addition, with the help of a hashing technique, HashKV utilizes GCs to update values and performs twice as efficiently as WiscKey in an edge-computing environment. However, both of these databases fail in terms of their utilization of hardware resources on embedded boards and CaseDB attains the highest performance.

The above-mentioned experimental results show that CaseDB achieves an improvement of 5.7 and 1.8 times over LevelDB and WiscKey, respectively, in terms of write



(a) Load throughput (200 GB) (b) Value size: 50 bytes (c) Value size: 16 Kbytes

FIGURE 11. **YCSB macrobenchmark performance.** ***(Notes on workloads: Workload A has 50% reads and 50% updates, Workload B has 95% read and 5% updates, Workload C has 100% reads, Workload D has 95% reads and 5% new insertions, Workload E has 95% range queries and 5% new insertions, and Workload F has 50% reads and 50% read-modify-writes. Keys are chosen from a Zipf).***

throughput and that its read throughput is 1.5 times faster. It also solves the space amplification problem of WiscKey. Moreover, CaseDB does not fully depend on the NVMe performance and achieves the best performance/cost tradeoff. In short, CaseDB fulfills all the needs of the embedded board environment.

VI. RELATED WORK

Several methods based on a log-structured file system have been developed [3], [17], [18], [20], [22], [23], [24], [31], [32], [34]. The LWC tree [10] offers metadata-based compaction without separating keys from values. The LWC tree functions by creating more metadata blocks in addition to data tables. It eliminates the reading of lower level data tables during compaction and reduces compaction I/O. However, this method loses consistency because metadata aggregation requires substantial effort to function efficiently. PCP [16] is a pipelined compaction procedure to reduce the compaction time. PCP divides each compaction step into subtasks and completes them in a pipelined manner. The PCP method cannot reduce compaction I/O but improves the write throughput by performing compaction efficiently. The VT-tree [27] uses a stitching technique to avoid unnecessary disk accesses for sorted and nonoverlapped key ranges. SkipStore [9] employs an additional buffer during compaction; this helps to delay the compaction of KV items for further levels. FlameDB [8] divides each level into groups and performs group-based compaction, in which all upper-component data are compacted to lower components as a group without merging.

Several other efforts have attempted to improve the read performance of KV stores based on an LSM-tree. RocksDB uses multithreaded read operations to look up keys on all levels in parallel to improve the scanning performance. COLA [25] and FD-tree [33] use forward pointers to accelerate the lookup process for specific data. SLM-DB [12] uses B+-tree indexing that considerably increases the read performance by keeping the reference for all KV items in memory. CaseDB eliminates the drawbacks of all the above-mentioned solutions and provides superior performance in an edge-computing environment.

VII. CONCLUSION

The major disadvantage of the level-by-level data-storing structure of an LSM-tree is that it causes write amplification. Moreover, its performance on an embedded board equipped with a low-power CPU and limited memory is unacceptably low. CaseDB efficiently reduces the write amplification by separating the keys and bloom filters as metadata and by delaying value compaction with the help of metadata compaction on NVMe drives. CaseDB also employs an additional memory component, CBuffer, in the upper level of Memtables to merge small-sized values into a larger KV item and generates a unique internal key for the coalesced group. Compared to WiscKey, CaseDB employs a deduction-based data deduplication technique to avoid space amplification in update-intensive workloads. Extensive experiments show that CaseDB outperforms LevelDB and WiscKey write performance by 5.7 and 1.8 times, respectively, on average. Moreover, CaseDB does not fully depend on the NVMe performance and achieves the best performance/cost tradeoff. Although CaseDB is designed for data storage on embedded boards, it can easily be adopted for data storage on servers. The use of CaseDB in an edge-computing environment is recommended to enhance the response time.

REFERENCES

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996.
[2] K. Tulkinbekov, M. Pirahandeh, and D.-H. Kim, "CLeveldb: Coalesced leveldb for small data," in *Proc. 11th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, Zagreb, Croatia, Jul. 2019, pp. 567–569.
[3] R. Sears and R. Ramakrishnan, "BLSM: A general purpose log structured merge tree," in *Proc. Int. Conf. Manage. Data*, 2012, pp. 217–228.
[4] (2016). *Leveldb Main Page*. [Online]. Available: https://code.google.com/p/leveldb/
[5] (2016). *Rocksdb Main Page*. [Online]. Available: http://rocksdb.org/
[6] (2019). *NoSQL Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/NoSQL
[7] 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," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.
[8] W. Zhang, Y. Xu, Y. Li, Y. Zhang, and D. Li, "FlameDB: A key-value store with grouped level structure and heterogeneous Bloom filter," *IEEE Access*, vol. 6, pp. 24962–24972, 2018.

[9] Y. Yue, B. He, Y. Li, and W. Wang, "Building an efficient put-intensive key-value store with skip-tree," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 961–973, Apr. 2017.
[10] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, "Building efficient key-value stores via a lightweight compaction tree," *ACM Trans. Storage*, vol. 13, no. 4, pp. 1–28, Dec. 2017.
[11] 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 *Proc. USENIX Annu. Tech. Conf.*, Santa Clara, CA, USA, Jul. 2015, pp. 71–82.
[12] O. Kaiyrakhmet, S. Lee, B. Nam, S. Noh, and Y. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol.*, Boston, MA, USA, Feb. 2019, pp. 1–7.
[13] *Connected Devices*. (2018). [Online]. Available: https://comparitech.Com/internet-providers/iot-statistics/
[14] (2019). *Big Data*. [Online]. Available: https://www.techjury.net/stats-about/big-data-statistics/
[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[16] Z. Zhang, "Pipelined compaction for the LSM-tree," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, Oct. 2014, pp. 777–786.
[17] M. Gosh, I. Gupta, S. Gupta, and N. Kumar, "Fast compaction algorithms for NoSQL databases," in *Proc. 35th Int. Conf. Distrib. Comput. Syst.*, pp. 452–461, 2015.
[18] C. Lai, "Atlas: Baidu's key-value storage system for cloud data," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, Oct. 2015, pp. 1–14.
[19] 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*, 2016, pp. 133–148.
[20] P. Wang, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.
[21] (2008). *Bloom Filter*. [Online]. Available: https://antognini.ch/papers/BloomFilters20080620.pdf/
[22] B. He, J. X. Yu, and A. C. Zhou, "Improving update-intensive workloads on flash disks through exploiting multi-chip parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 152–162, Jan. 2015.
[23] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[24] H. Lim, D. G. Anderson, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *Proc. 14th USENIX Conf File Storage Technol.*, 2016, pp. 149–166.
[25] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming B-trees," in *Proc. 19th Annu. ACM Symp. Parallel Algorithms Archit.*, 2007, pp. 81–92.
[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, Indianapolis, IN, USA, Jun. 2010, pp. 143–154.
[27] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building Workload-independent storage with VT-trees," in *Proc. 11th USENIX Conf. File Storage Technol.*, vol. 13, San Jose, CA, USA, 2013, pp. 17–30.
[28] (2019). *db_bench*. [Online]. Available: https://github.com/google/leveldb/blob/master/benchmarks/db_bench.cc
[29] (2019). *Apollo Dataset*. [Online]. Available: https://data.apollo.auto/?locale=en-us&lang=en
[30] H. H. W. Chan, Y. Le, P. P. C. Lee, and Y. Xu, "HashKV: Enabling efficient updates in KV storage via hashing," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2018, pp. 1007–1019.
[31] S. Chen and Q. Jin, "Persistent B+ trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 15, Feb. 2015, pp. 786–797.
[32] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th Usenix Conf. File Storage Technol.*, vol. 15, 2017, pp. 257–270.
[33] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1195–1206, Sep. 2010.
[34] F.-F. Pan, Y.-L. Yue, and J. Xiong, "DCompaction: Speeding up compaction of the LSM-tree via delayed compaction," *J. Comput. Sci. Technol.*, vol. 32, no. 1, pp. 41–54, Jan. 2017.
[35] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, pp. 668–676, Jun. 1990.

KHIKMATULLO TULKINBEKOV received the bachelor's degree in information and communication engineering from Inha University in Tashkent, Uzbekistan, in 2018. He is currently pursuing the master's degree with the Intelligent Embedded Systems Laboratory, Inha University, South Korea. He is also a Research Assistant with the Intelligent Embedded Systems Laboratory, Inha University. His research interests include storage systems and intelligent cloud storage.

DEOK-HWAN KIM (Member, IEEE) received the Ph.D. degree in computer science from the Korean Advanced Institute of Science and Technology (KAIST), in 2003. He has been a Professor with Inha University, South Korea, since 2006. He has authored or presented more than 50 publications in major journals and at international conferences. His research interests include storage systems, intelligent cloud storage, embedded and real time systems, and biomedical systems.