WipDB: A Write-in-place Key-value Store that Mimics Bucket Sort

1st Xingsheng Zhao
University of Texas at Arlington
Arlington, USA
xingsheng.zhao@mavs.uta.edu

2nd Song Jiang
University of Texas at Arlington
Arlington, USA
song.jiang@uta.edu

3rd Xingbo Wu
University of Illinois at Chicago
Chicago, USA
wuxb@uic.edu

***Abstract*—Key-value (KV) stores have become a major storage infrastructure on which databases, file systems, and other data management systems are built. To support efficient indexing and range search, the key-value items must be sorted. However, this sorting process can be excessively expensive. In the KV systems adopting the popular Log-Structured Merge Tree (LSM) structure or its variants, the write volume can be amplified by tens of times due to its repeated internal merge-sorting operation.**

In this paper we propose a KV store design that leverages relatively stable key distributions to bound the write amplification by a number as low as 4.15 in practice. The key idea is, instead of incrementally sorting KV items in the LSM's hierarchical structure, it writes KV items right in place in an approximately sorted list, much like a bucket sort algorithm does. The design also makes it possible to keep most internal data reorganization operations off the critical path of read service. The so-called Write-in-place (Wip) scheme has been implemented with its source code publicly available. Experiment results show that WipDB improves write throughput by 3 to 8× (to around 1 Mops/s on one Intel PCIe SSD) over state-of-the-art KV stores.

***Index Terms*—key-value store, index, SSD, storage**

I. Introduction

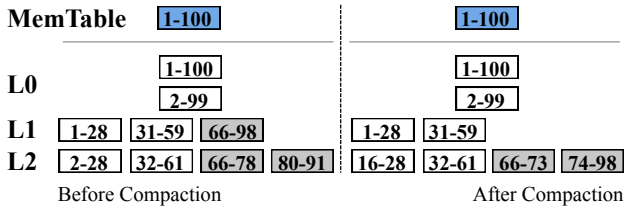
Key-value (KV) stores have become a major data management component in a storage system. By providing a KV API for writing, reading, and updating data items, the store makes it possible for the upper-level software to access the storage in its own defined key space for values of any sizes. In contrast to the rigid block interface provided by the block storage subsystems, such as physical/virtual disks and storage volumes, a KV interface makes user-facing software, such as file systems [1], [2] and database systems [3], [4], much easier to develop, as it can leave chores, such as address conversion between keys and block addresses and storage space management, to the KV system. More importantly, in a distributed system the key space can be conveniently partitioned into multiple shards, enabling a shared-nothing architecture to achieve linear horizontal scalability.

However, to realize its full potential a KV store must be well designed to simultaneously meet a number of goals, which are (1) good performance with small KV items in a storage system of large capacity, (2) support of range search, (3) low read amplification, and (4) low write amplification.

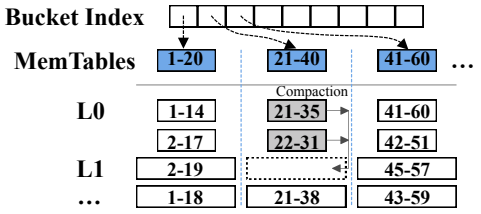
It is a challenge to simultaneously achieve all these goals in a KV system. It has been noted that in many real-world KV workloads small keys and values are common [5], [6]. A KV item can be tens of bytes. Each (4KB) disk block may contain tens of such items. In a server of multi-terabyte storage space there can be tens of billions of such items. If such a huge number of data items are individually indexed, the index size (tens of, and even hundreds of gigabytes) can be too large to be fully held in the memory. If (some) index data are only stored on the disk, it needs multiple disk reads (for index data and then KV data) to service a read request, leading to high read amplification. To achieve the first three goals, the KV items have to be sorted by their keys. If sorted, the store can index disk blocks, instead of individual KV items, to significantly reduce index size and use only one disk access to service a read request. When the items are sorted, range search (e.g., for all items between two keys, or for items of a common prefix) can be well supported.

Unfortunately, maintaining an always and fully sorted list on the disk is a prohibitive task. Analogous to expensive data shifting operations for maintaining a sorted array in the memory, writing new KV items into an on-disk sorted list, which is stored in one or multiple files, involves re-writing of file(s). This usually leads to exorbitant write amplification. High write amplification, on the one hand, compromises foreground throughput as more bandwidth is consumed by background disk I/O. On the other hand, it reduces lifetime of SSD disks. Unsurprisingly, major efforts on improvement of KV stores have been on the amelioration of the write-amplification issue.

Currently the most successful effort is to employ the log structured merge tree (LSM-tree) technique to maintain multiple and increasingly large levels of sorted lists [7] (See Figure 1a). Example KV systems adopting the technique include Google's LevelDB [8] and Facebook's RocksDB [9]. In these systems new KV items in the incoming write requests are progressively merge-sorted on different levels across the hierarchical structure and finally enter the last and the largest level of a sorted list. Though LSM-tree can significantly reduce write amplification, the amplification can still be as high as 20∼70X depending on store size [10]–[13]. To reduce the high write cost, researchers adopt the approximate sorting technique, attempting to avoid rewriting data in the same level and reduce the amplification to as low as number of levels (e.g., 5∼7X) [10], [11]. In such a sorting method, the *key*



(a) Compaction in LSM-tree



(b) WipDB

Fig. 1: *Architectures of LSM tree and WipDB*

space is partitioned into multiple buckets (each for a given key range), and KV items are globally sorted (across the buckets) but locally unsorted (within a bucket). While the level size grows exponentially in the LSM-tree hierarchy, the store usually has seven or fewer levels. If successful, the technique can substantially reduce write amplification.

However, current approximate sorting technique has two critical issues in its effort on achieving low write amplification. The first issue is that the approximate sorting technique cannot be effectively applied on the LSM-tree structure due to its demand on an appropriate key partitioning. As a consequence, the stores adopting this technique either do not support range search (e.g., LSM-trie [10]) or consume a considerable amount of memory in guards to reduce the write amplification. (e.g., PebblesDB [11]). The second issue is that the expensive across-level progressive sorting operations compromise performance of read requests. In terms of performance impact, the high write amplification is less of an issue on the write performance than on the read performance. If the sorting operations are not immediately conducted (e.g., postponed to less busy periods), writes can be quickly done. However, the read performance will be seriously degraded if the items are not adequately sorted (more explanation in Section II). However, if the sorting is carried out immediately, expensive across-level sorting would consume substantial bandwidth, also degrading the read performance.

In this paper we propose a new KV store architecture to fundamentally address the two issues. Accordingly, the write amplification can be effectively reduced and most sorting operations can be moved off the read service's critical path. In the way, all the four goals can be simultaneously achieved. The proposed KV store is named WipDB (Write-in-place). It mimics the bucket sort by partitioning the entire key space into a large number of buckets, and placing keys directly in the corresponding buckets. In contrast, existing LSM-tree-based KV stores, including LevelDB and the optimized ones using the approximate sorting technique, place KV items into a level's corresponding buckets (e.g., represented by SSTables in LevelDB) and move them down to next level's buckets using merge sort in a level-by-level manner, as shown in Figure 1a. However, similar to that in a bucket sort a key has only one bucket in WipDB holding it and KV items are not moved across buckets. KV items in a bucket are managed with a miniature LSM tree (see Figure 1b).

In this WipDB architecture, both of the aforementioned issues can be well addressed. First, the approximate sorting can be used in each miniature LSM tree of a bucket without key partitioning to reap its full performance benefit. Second, expecting existence of read locality in the key space, WipDB introduces a locality-aware sorting scheme. Sorting in different buckets can be scheduled so that sorting of KV items in buckets that do not actively service read requests can be postponed. This benefit for hiding internal data sorting cost is not possible in existing LSM-tree-based KV stores.

In summary, in the paper we make three major contributions: (1) we propose a new LSM-tree-based KV store architecture that allows new keys to be written in their right places (buckets) in a sorted list at the outset by utilizing long-term key distribution; (2) by improving and hiding write-related sorting cost, WipDB can make both writes and reads efficient; and (3) we have implemented WipDB and extensively evaluate its efficiency by comparing it with state-of-the-art KV stores.

II. The Background

To motivate the design of the proposed WipDB, in this section we will discuss the design rationale of the LSM-tree architecture and its weaknesses, as well as existing efforts attempting to address the weaknesses and their inadequacies.

A. Why LSM-tree?

As we have indicated, KV items must be sorted in the storage for reduced index size (so that it can be all cached in the memory) and support of range search. This is especially the case for a KV store hosting a large number of small KV items. The challenge is to keep the items sorted when new items keep streaming into the store, as the cost can be huge. Assuming that the store currently has M KV items (of the same size) that have been fully sorted on the disk. A list of N (sorted) new items currently in the memory are to be written to the store. To maintain a fully sorted store, one has to write $M + N$ items, rather than just the N new items to the store. Specifically, one needs to first read the M items off the disk, merge-sort the new list of N items with the existing list of M items into one list, and write the resulting sorted list back to the disk. The operation is often named *compaction* [4]. The write amplification is $(M+N)/N$, which can be huge when the store size (M) and the ratio of the two lists (M/N), named *compaction ratio*, become increasingly large. To address the issue there are two potential ways. One is to reduce the ratio (M/N). LSM-tree takes this way. The other is to increase size of the in-memory list (N). This is the way taken by the proposed WipDB. However, there are significant

hurdles to overcome on the way towards its effectiveness (see Section II-C).

To limit the ratio and allow the store to grow to a very large size, LSM-tree introduces multiple sorted lists to form a multi-level hierarchy. As illustrated in Figure 1a, these levels are named Levels 0, 1, ..., and $n-1$ in a n-level hierarchy. Except for L_0, which sits at the top of the hierarchy and may have multiple sub-levels of similar size, every other level is up to 10X as large as the level immediately above it.

Because compaction is conducted only between two adjacent levels, the compaction ratio is bounded by 10, or the write amplification for moving KV items from one level to its next lower level is bounded by 11. For example, in a 5-level LSM-tree store, starting at Level 0 KV items move down the hierarchy level-by-level in a sequence of compactions and produce a write amplification of up to 44 when they arrive at Level 4. Though LSM-tree's write amplification is way much smaller than the store that inserts items directly into a single sorted list, its amplification is still significant. High write amplification consumes much of the I/O bandwidth and makes both write and read requests slow.

B. Why Approximate Sorting?

Apparently, LSM-tree's 20-70X write amplification is still a major performance concern. Recently, significant efforts have been made to reduce the cost by using *approximate sorting* to avoid rewriting data in the same level [10], [11]. As we know, to reduce write amplification LSM-tree does not directly sort KV items into a single list. Instead, it only sorts the items in the same level (horizontally sorted) and leaves items in different levels unsorted (vertically unsorted). In the approximate sorting, the key space in each level is partitioned into multiple segments, named buckets. Keys between the buckets are sorted. Keys within a bucket are not fully sorted. There can be multiple overlapping small sorted lists in a bucket. Each level is said to be approximately sorted.

By allowing each level to be horizontally unsorted, the key-value architectures, represented by LSM-trie [10] and PebblesDB [11], can potentially reduce a compaction's write amplification to 1, and possibly make the store's write amplification as low as its level count. Here is the reason. During a compaction, to move items in a bucket at a level to its next level, the store first merge-sorts the small lists in a bucket into one sorted list. It then uses the boundaries of buckets at the next level to segment the list and simply writes each segment to its corresponding bucket in the next level. Allowing KV items to be partially (either horizontally or vertically) unsorted, a read operation needs to search more sorted lists, such as levels in a LSM-tree and small lists in a bucket of an approximately sorted list. By using stronger bloom filters this may not be a performance concern. The real concern is on use of the approximate sorting in different levels.

For the approximate sorting approach to be effective, one has to segment the key space appropriately so that each bucket has about equal number of KV items [1]. Otherwise, some buckets may receive new items at a (much) higher rate than others, making them and their downstream buckets become very large, and their compactions expensive. The hierarchy would also grow in an unbalanced manner. This is why LevelDB introduces constant-size SSTable. It is also reminiscent of the tree balancing issue addressed in the B+ tree.

Currently, the issue is not well addressed in the use of approximate sorting, leaving its performance promises unfilled. To ensure a balanced LSM-tree structure, LSM-trie hashes user-supplied keys with a cryptographic hash function like SHA-1, and uses the hashed keys for sorting in the tree. A consequence of this approach is that range search is not supported. In contrast, PebblesDB attempts to preserve the support of range search by keeping user keys sorted. To this end, it uses a probabilistic function to select some keys as "guards" from all keys entering a level to form buckets (between two adjacent guards). Number of guards, or buckets, in a region of the key space in a level is roughly proportional to the region's key density. Intuitively, each bucket has about the same number of KV items. But positions of guards may keep changing, especially for guards in the higher levels (Level i where i is small), in response to variation of key density in the corresponding key regions. In PebblesDB, a guard in a higher level must also be a guard in all of its lower levels [11]. Adding a new guard into a level requires splitting of a bucket. This will either cause rewriting data in the same level, which defeats the very purpose of using approximate sorting, or requires an immediate premature compaction of the bucket to the next level, leading to a cascade of downward compactions. Both can significantly offset the benefit of the approximate sorting.

C. Overcoming Variation of Key Density

The key to success of the approximate sorting approach is stability of the key distribution in the key space. In other words, the key densities in different regions of the key space need to remain roughly unchanged (at least for an extended period of time), so as to keep the partitioning (into the buckets) stable. We assume user keys generated by applications have a long-term relatively stable distribution (e.g., for weeks, months, or even years). For real-world services, keys are often constructed by sequencing some descriptors of an object. For example, a key about an Amazon's product can be generated as *Grocery* $\rightarrow$ *Snack Foods* $\rightarrow$ *Cookies* $\rightarrow$ *Chocolate* $\rightarrow$ *Oreo Mini Chocolate Sandwich Cookies*. In the example of Google's BigTable [4], which decomposes a database table into KV items for storage in a distributed KV store, a key is generated by concatenating row name, column name, timestamp, such as *com.google.maps/index.html+Spanish+04/18/2019,12:00am*. In the real world the number of products or news articles under a certain category, or popularity of a category, can be expected to be stable within a relatively long time period. So is the long-term key-distribution stability in many significant application scenarios [2].

[1] This is a requirement similar to that on an efficient bucket sort.

[2] In Section IV-B, we experimentally show that WipDB's performance advantage is not contingent on existence of a strong stability at all.

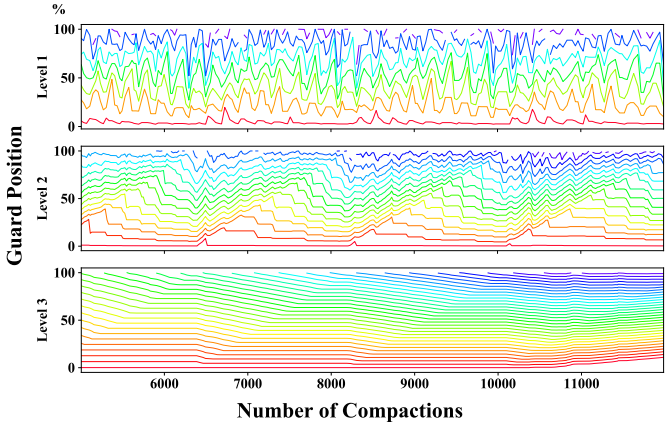


Fig. 2: *Guard positions in different levels in LevelDB (L1, L2, and L3) after certain number of compactions in the system. The position is expressed as a percentage of the guard key in the entire key space* ($0..10^9$)*. A workload with the uniform distribution is used here.*

The issue is that existing use of the approximate sorting approach cannot exploit the long-term stability, as such stability may only exist in the last one or two levels, which store majority of a store's KV items inserted over weeks, months, or even years reflecting the user keys' inherent and stable distribution. In contrast, KV items in the top levels are inserted in a short time period and may exhibit a key distribution that's different from the long-term stable one and can change quickly from time to time. For example, Level 0 has a capacity of tens of megabytes, and may only store KV items inserted in the last few minutes. However, when stores such as PebblesDB [11] applies the approximate sorting approach at every level, dramatic and expensive bucket adjustment is expected in the most (top) levels, which cascades to the lower levels.

To illustrate the situation, we continuously write 100-byte KV items, whose keys are generated using the `db_bench` tool in the LevelDB code release. In each level we contiguously place a hypothetical guard for every 50K keys, or a bucket between two adjacent guards holding 50K KV items. After 1 billion items have been inserted, we track variation of guard positions in Levels 1, 2, and 3 after every compaction. As Figure 2 shows, in all the three levels the guard positions vary but at different intensity. In other words, if we had fixed the guard positions the number of KV items in a bucket can be highly variable in Levels 1 and 2. However, in a lower level (Levels 3), where much more items are stored, spontaneous variation of key distribution can be smoothed out.

D. The Write-in-place Approach

To leverage the stable key space distribution, we propose to eliminate the top levels in the LSM-tree structure and partition the key space of the last level into equal-capacity buckets according to the workload's long-term key distribution. Within each bucket of limited capacity KV items are organized as a miniature LSM-tree, as illustrated in Figure 1b. While the proposed write-in-place (WipDB) approach similarly allows partially sorting in both horizontal and vertical dimensions, the novelty is to switch the order of the dimensions where the partially sorting technique is applied to make exploitation of the long-term stable key distribution possible.

LevelDB introduces the partially sorted structure on the vertical dimension (multiple overlapping sorted levels). PebblesDB further allows partial sorting on the horizontal dimension (multiple partially sorted buckets in each level). In contrast, WipDB first applies the partially sorting technique on the horizontal dimension by directly placing keys in the right buckets. It then uses the technique on the vertical dimension by using LSM-tree in each bucket. The new architecture has two benefits. First, because the LSM-tree in a bucket is of limited size, WipDB can apply the approximate sorting technique to avoid rewriting in the same level without further partitioning any level of the tree into smaller buckets. Second, the sorting operation in a bucket can be scheduled according to the read request pattern to potentially move it out of read operations' critical path.

III. The WipDB Design

By leveraging the approximate sorting technique at only one level that contains a long list of data items and exhibits stability of a key distribution, WipDB can pre-define buckets in the key space and write incoming KV items into the buckets they belong to (write-in-place). Conceptually, WipDB mimics the bucket sort with KV items sorted across the buckets. Within a bucket, the items are managed within an LSM-tree. The design of WipDB addresses a number of critical issues, including how to partition the key space into buckets, how to efficiently write KV items into the buckets on the disk, how to prevent new KV items in the memory from being lost upon power failure or system crash, how to minimize the performance impact of in-bucket sorting, and how to adapt the buckets to change of key distribution.

A. The WipDB Architecture

In the WipDB architecture (Figure 1b), the key space is partitioned into a certain number of buckets so that each bucket is supposed to contain about the same number of KV items according to the observed key distribution. Each of the buckets admits new KV items directly from a buffer in the memory that corresponds to the bucket. The buffer is responsible for receiving new KV items whose keys are in same range of its corresponding bucket. As the buffer plays a role similar to MemTable in LevelDB [8], we name it *MemTable* too. The difference is that there are multiple MemTables, each for a bucket on the disk. When a MemTable is full, it's written to its corresponding bucket on the disk as a file containing a sorted list of KV items. This process is similar to the minor-compaction operation in LevelDB, where a MemTable becomes an SSTable in Level 0 of the LSM-tree on the disk.

As we mentioned, we use the LSM-tree structure within each bucket. To reduce write amplification due to compaction

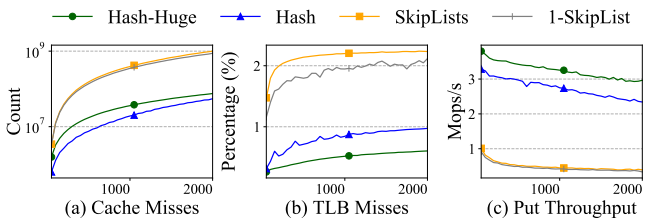


Fig. 3: *Performance comparison of skip list and hash table.*

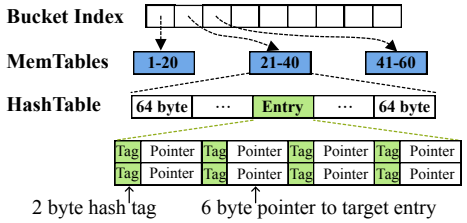


Fig. 4: *MemTable Design*

within the LSM-tree, we adopt the technique used in LSM-trie and PebblesDB, where a level consists of multiple overlapping sublevels, to avoid rewriting in the same level (see Figure 1b). Specifically, in a compaction operation KV items in the sublevels of Level i ($i = 0, 1, 2, \ldots$) are merge-sorted into one list, which is then written back as a new sublevel of Level $i + 1$ with a write amplification of one. Interestingly, though the three KV stores (WipDB, LSM-Trie, and PebblesDB) use the same technique for significantly reduced write amplification, only WipDB can take its full advantage. As we mentioned, when one big LSM-tree is used to manage a KV store, each level has to be partitioned into multiple segments, such as SSTables in LevelDB. The store conducts compactions on a few selected segments once at a time to cap the time of a compaction operation. To make the tree grow in a balanced manner, both LSM-trie and PebblesDB make a major effort attempting to maintain about the same number of items in each segment. This is challenging as short-term key distribution keeps changing. To this end, LSM-trie gives up support of range search by using SHA-1 hashed keys in the sorting. PebblesDB constantly adds guards in each level, incurring SSTable splitting operations and rewriting in the same level. WipDB addresses this issue by removing the need of introducing segments into a level. As a WipDB store can have a sufficient number of buckets so that each bucket won't grow very large (e.g, up to a 1GB). Each level is limited at a relatively small size (e.g., tens of Megabytes). This makes partitioning within a WipDB's level unnecessary. Therefore, all sorted KV items in a sublevel of a bucket's LSM-tree are stored in one file, which is named *LevelTable*. A compaction is applied on multiple LevelTables of a level. In this way, the write amplification can be as low as the number of levels.

B. The Operations

WipDB supports all basic KV store operations, including write, deletion, modification, and read. Like most other KV stores, WipDB executes write, delete, and modification operations, collectively named *update operations* as writing new KV items to the store. In particular, for deletion operation a special KV item, whose value is a tombstone marker indicating it's a deletion request, is written. The actual deletion and modification are actually performed during compactions.

A read operation can request for either one KV item (point search) or all items in a key range (range search). A point search starts at the MemTable and proceeds across the LevelTables in the order of levels until a key is found or it reaches the last (sub)level in the corresponding bucket. In the process, use of Bloom filters in each sublevel can avoid most or all access of files that do not contain the requested key. However, for range search every LevelTable must be read and searched for the keys in the specified range. The results from the sublevels in all relevant buckets are combined and returned to the requester. Therefore, range search is an I/O-intensive and expensive operation.

C. Efficiency and Persistence of MemTables

In LevelDB, MemTable is maintained as a sorted data structure (skip list), so that it can directly support range search and be readily written to the disk as a Level-0 SSTable. However, such a design can be problematic for WipDB.

WipDB may have a large number (e.g., a few thousands) of MemTables. Its enlarged working set and weakened access locality may lead to a very high CPU cache miss ratio. A write is always preceded by a lookup in a MemTable for its insertion location. A lookup in a sorted structure, such as a skip list, may require multiple memory accesses and incur multiple cache misses. To illustrate this, we set up systems of different number of MemTables, each with a capacity of 10K KV items, and write random keys into them. As shown in Figure 3, KV items in each bucket can be organized as a skip list ("SkipLists"), a hash table ("Hash"), or a hash table with huge page enabled in Linux ("Hash-Huge"). We also include results for all keys in one big skip list ("1-SkipList"). The system setup is described in Section IV-A. As shown, using skip lists causes much more cache and TLB misses, and accordingly produces a write throughput much lower than using hash table. This issue of using a sorted in-memory table is particularly serious when high-speed SSDs are used and the store has a low write amplification.

To address the issue, WipDB uses a hash table to implement a MemTable. As shown in Figure 4, each entry of the hash-table's directory has 64 bytes (the cacheline size), which consists of eight 8-bytes slots. The entry is 64-byte-aligned. One memory access can retrieve all its eight slots into the cache. Conceptually, each slot stores a KV item. In reality, we hash the key into a two-byte tag. The tag and a six-byte pointer pointing to the space storing the KV item are stored in the slot. The eight slots in an entry are used as a log. New KV items are appended at the end of the log. A lookup in the entry starts from current end of the log. When the hash table is full (i.e., any of its entry has overflown) WipDB freezes the table, sorts its data items, and writes them to the disk as a

LevelTable at Level 0. Meanwhile, a second empty hash table is set up to continue admitting incoming KV items.

Any KV items in the MemTables hosted in the DRAM are subject to loss due to power failure or system crash until they turn into LevelTable on the disk. Therefore, WipDB writes any new KV items into a write-ahead-log before their requests are acknowledged.

D. Support of Range Search

The hash-table-based MemTable does not directly support range search. When a range-search request arrives at a MemTable, WipDB immediately sorts the data items currently in the hash table and place them in a one-time-use buffer, which is discarded after the range search has completed its scanning. KV items are copied, rather than moved, from the hash table to the buffer.

This design choice is in stark contrast with FloDB [14], a KV store dedicated for improving operating efficiency of the skiplist based MemTable. FloDB adds a hash table on top of the MemTable and keeps pushing KV items from the hash table to the MemTable. WipDB cannot adopt such a design. With such a two-level in-memory structure a read request still needs to search the skip list (unless the item has been found in the hash table), causing many cache misses, which compromises performance. Interestingly, FloDB cannot use WipDB's choice either as it assumes a MemTable as large as 192 GB to take full advantage of memory's high speed. In contrast, each WipDB's MemTable has only one or a few megabytes, or a few thousand KV items. While a range search operation is very expensive, it is well affordable to sort this relatively small number of items in a hash table. In addition, if a bucket receives a large number of range queries during a time window, WipDB replaces the hash-based MemTable with a skiplist-based one to reduce sorting overhead for this bucket. If no more range-query requests arrive after next minor compaction, it changes back to hash-based MemTable. This adaptive strategy adjusts the MemTable structure of each bucket individually based on the workload, so that the sorting overhead can be minimized.

To ensure that items arriving after a range search request from being considered, WipDB leverages a global unique and monotonically increasing sequence number assigned to any incoming item in the order of their arrival. Such a sequence-number mechanism is also adopted in other KV stores such as LevelDB and RocksDB. When a search request is received, the sequence number currently available for assignment is attached to the search. During a search any items whose sequence numbers are equal to or larger than the sequence number with the search are skipped.

E. Bucket Splitting and Merging

WipDB does not pre-assign a large number of buckets when a store is initialized as the key distribution is not known yet. Instead, it has only one or a few initial buckets. When the store grows or the key distribution of the incoming KV items changes, a bucket may become too large, and have too many levels (or sublevels), which degrades read performance. In principle the WipDB's structure is similar to a hash table, whose bucket capacity also needs to be capped for desired lookup performance. The difference is that KV items in a WipDB are sorted across its buckets. For this reason, it can be much more efficient to reduce a bucket's size. Instead of reshuffling the entire store, WipDB can individually split a bucket once it reaches its capacity.

Assuming each level of the LSM tree in a bucket consists of maximally T sublevels. When a bucket reaches its capacity each of its levels consists of T full sublevels. We choose to evenly split the bucket into N smaller buckets when it exceeds its capacity threshold. We consistently apply the same set of $N-1$ splitters at each sublevel to produce N segments. Items in the ith segments ($i = 1, 2, \ldots, N$) of all sublevels constitute one of the N new buckets. As WipDB grows incrementally from a small number of buckets initially to thousands of buckets by this bucket splitting, the choice of the splitters is important to balancing buckets. To this end, similar to the sample sort, for each sublevel we first choose $N-1$ splitters that evenly partition it. Assuming there are L levels, or $L \times T$ sublevels, we then sort the list of the $L \times T \times (N-1)$ splitters and choose $N-1$ splitters that evenly split the splitter list for the bucket splitting. During the splitting, the bucket continues servicing incoming requests. When N new buckets are created, items in the MemTable are written to one of the new buckets according to their keys. Therefore, N new Memtables are created, each for a new bucket, to receive incoming items. In the meantime, the old bucket becomes read-only to serve read requests that cannot be satisfied in the new buckets. WipDB carries out a full compaction to turn the old bucket into one sorted list. It then partitions the list into N segments according to the selected splitters and places them in the new buckets respectively as their last levels. Eventually, all requests can be processed by the new buckets and the old bucket is removed.

A bucket shrinks after repeated deletion of its KV items. While existence of small buckets does not compromise performance, it may increase number of buckets and thus memory footprint. Small buckets can be removed by merging them with their neighboring buckets, which helps reduce WipDB's memory demand. Admittedly, by having multiple MemTables WipDB uses more memory for its in-memory data structure than other KV stores such as RocksDB and PebblesDB. However, its demand on memory is still very small. For example, for a 5TB disk filled with KV items of around 512B each, the memory demand is only 2GB, which is negligible on a commercial server equipped with several hundreds of GB memory. Furthermore, its use of the additional memory brings significant improvement of write performance, as will be shown in Section IV. This benefit is not available to existing KV stores even if a much larger memory is offered.

WipDB's write amplification (WA) is mainly attributed to (1) compaction operations on LSM trees within individual buckets; and (2) bucket splitting operations. As we have discussed, with its use of vertical approximate sorting WipDB's compaction-induced WA is bounded by L_{max}, the maximally

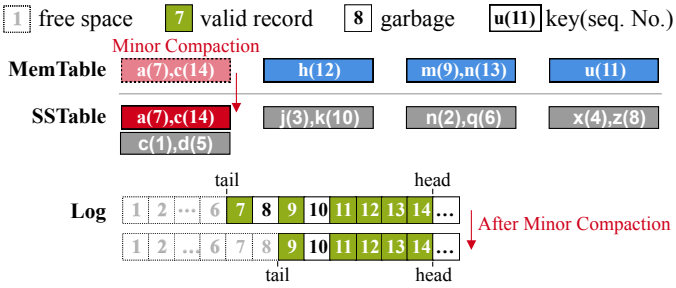


Fig. 5: *Space reclamation in WipDB's Write-Ahead Log*

allowed level count in a bucket. Assuming a full bucket's size is 1 and its splitting produces N new buckets, each with a size of $\frac{1}{N}$. Each of the new buckets will be split again with the entire bucket being re-written after it grows from size $\frac{1}{N}$ to 1 with a minimal $1-\frac{1}{N}$ data written into the bucket from users. Accordingly, the split-induced I/O write amplification is upper-bounded by the ratio between amount of write for bucket splitting and amount of write of new data from users, which is $1/(1-\frac{1}{N})$, or $\frac{N}{N-1}$. Considering write amplification due to both compaction and bucket splitting, the store's WA is upper-bounded by $L_{max}+\frac{N}{N-1}$, regardless of bucket count or the actual store size. Assuming a practically configured WipDB store whose $L_{max}=3$, $N=8$, the total WA is no more than 4.15.

F. Use of Write Ahead Log for DRAM resident MemTable

In order to tolerate power failure, WipDB adopts a basic principle similar to existing KV stores, such as LevelDB, which writes a KV item to a write-ahead log (WAL) when it is still in the MemTable. The main issue for WipDB to address is how to reclaim the log space occupied by the KV items that have been safely persisted into the KV store itself.

In LevelDB KV items are persisted on the disk in their arrival order, which is also their order in the log. Each item is assigned a monotonically increasing sequence number in the order. Therefore, when an item with a particular sequence number is written to the disk, any items whose numbers are not larger than the sequence number in the log, or any items before a corresponding offset in the log file, can be removed. However, this is true only for items within individual MemTables in WipDB. WipDB has multiple MemTables. All new items are written to a common log. But they may be distributed in different MemTables.

We track the smallest sequence number in each MemTable among those whose corresponding KV items are not yet persisted. We then choose the smallest of every MemTable's smallest number. All items in the log whose sequence numbers not greater than this smallest number, which are contiguous in the log file, can be removed. In this way, the log space can still be efficiently reclaimed. The process is illustrated in Figure 5, where new items are written at the log head and free space starts at the tail. The smallest sequence number is at the tail (e.g., Sequence number 7). Note that there can be removable (garbage) items interspersed among valid items, such as items with Sequence numbers 8 and 10. After the leftmost MemTable is flushed to the disk, the tail moves forward to Sequence number 9. The space before the position can them be reclaimed and reused.

To prevent the log from becoming too large, WipDB sets up a threshold on the log size. When the threshold is reached, MemTables at the log tail are written to the disk to shrink the log.

G. Read-aware Compaction Scheduling

In an LSM-tree-based multi-level KV store, read performance can be compromised by searching in SSTables in different levels. The more the levels, the more likely for Bloom filters in the SSTables to have false positives and for a read request to take more than one disk access. To this end, compaction must be conducted to push new items downwards and reduce level count. However, intensive compactions can consume much I/O bandwidth and slow down the concurrent read requests. Should we be able to schedule compactions with priority on KV items being intensively read and reduce number of levels hosting these read items, the negative impact of compactions on read requests can be reduced. However, this is very difficult as LSM-tree's top levels are covered by a few SSTables. And it's hard to separate items being read from those being written into different compactions.

WipDB addresses the issue in its design. Each of its buckets, managed as a small LSM-tree, is only responsible for a fraction of the entire key space. As long as write and read requests do not intensively fall in the same buckets, WipDB can prioritize compactions on read-intensive buckets to reduce their levels and improve read performance. Even if the write and read key spaces are highly overlapped, it's likely the case where newly written items are immediately read in the following requests and the read requests can be serviced at a cache before they reach the storage system [15], [16]). For scheduling of compaction of all sublevels of a level in a bucket, WipDB considers two factors, which are number of current sublevels, denoted *sub_count* and number of times any of the sublevels are accessed to serve read requests since the last compaction of the level, denoted *read_count*. *sub_count* should be in the range of [*min_count*, *max_count*]. A level becomes eligible for compaction when it has at least *min_count* sublevels. This level receives the highest priority for compaction when it reaches *max_count* sublevels. Suppose average of eligible levels' *read_count*s is *avg_read_count*, the relative read count for a level of *read_count* reads is rela_read_count $=\frac{\text{read\_count}}{\text{avg\_read\_count}}$. Also if the average of eligible levels' *sub_count* is *avg_sub_count*, the relative sublevel count for a level of *sub_count* sublevels is rela_sub_count $=\frac{\text{sub\_count}}{\text{avg\_sub\_count}}$. The priority ($p$) of a level's compaction is qualified as

$p = (read\_weight) \times (rela\_read\_count) + rela\_sub\_count$, where $min\_count \le sub\_count < max\_count$.

The *read_weight* adjusts the weight of the read performance relative to importance of a balanced compaction across the buckets according to the sublevel count. The priority values are dynamically updated and the N levels with the highest

values are selected for concurrent compactions. By adopting a large *read_weight*, WipDB allows read-intensive buckets to be aggressively compacted for high-performance read service. In the meantime, it leaves the write-intensive and read-little buckets lightly compacted to save more bandwidth and further improve read performance. Based on our empirical study, we set default values of *min_count*, *max_count*, and *read_weight* as 4, 20, and 10, respectively.

IV. Evaluation

We implement a prototype of WipDB and evaluate it against three state-of-the-art KV stores: LevelDB (v1.20), RocksDB [9] (v5.18), and PebblesDB [11] (`git #220d0fa`). In the evaluation, we will answer the following questions:

- How does WipDB improve write performance?
- How effective is WipDB's MemTable?
- How does WipDB perform with workloads of changing key distribution?
- How does the compaction scheduling improve WipDB's read performance?

A. Experiment Setup

The experiments were run on a Dell T440 server with two 4-core Intel Xeon Gold 5122 CPUs and 64GB DRAM. To minimize the interference between threads or cores, hyper-threading is turned off from BIOS. The server runs a 64-bit Linux (v4.20.0) with an ext4 file system on an Intel 750 SSD (PCIe, 1.2 TB). The SSD has up to 1200 MB/s sequential write throughput and up to 440K IOPS for random 4KB reads.

In the evaluation, WipDB uses 2MB MemTable for each bucket. LevelDB, RocksDB, and PebblesDB use 64 MB MemTables. We configure WipDB with $L_{max} = 3$, $T = 8$ and $N = 8$. Another SSD of the same type holds the log file(s) to avoid impact of logging on systems' frontend operations in each of the four stores. Meanwhile, every 1000 write requests are logged as a batch for high efficiency.

B. Write Performance

WipDB aims to substantially reduce the high write amplification ratio (WA) of LSM-tree-based KV stores. To evaluate the performance improvement of WipDB for write-intensive workloads, we use 16-byte keys and 100-byte values with uniform distribution and send 8 billion write requests (around 900 GB data) to each store. WipDB is configured to have 100 buckets at beginning except stated otherwise. All the stores, except LevelDB which only supports one background thread, use seven compaction threads. We show throughput and WA of WipDB, LevelDB, RocksDB, and PebblesDB in Figures 6.

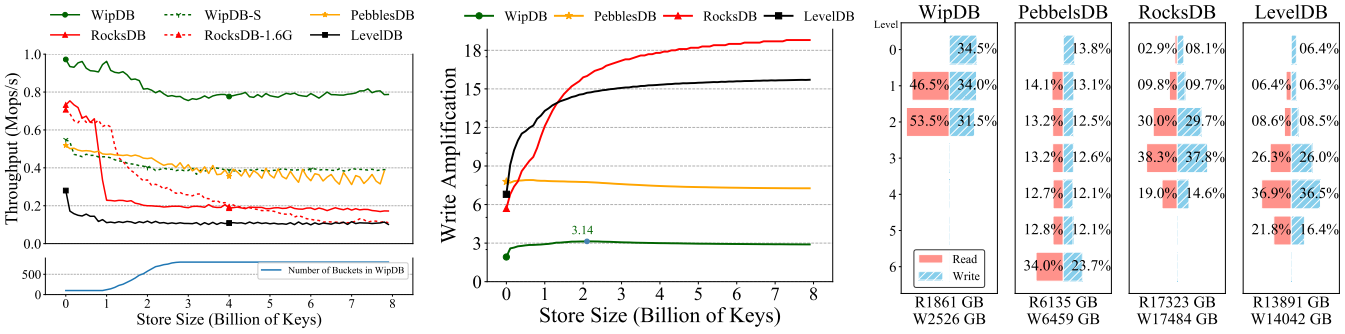
Write throughput and WA. As shown, WipDB has much higher throughputs than the other stores. This improvement is primarily due to its low write amplification (see Figure 6b). In particular, RocksDB and LevelDB's write amplification is about 5× to 6× higher than that of WipDB. Admittedly WipDB uses more memory for its MemTables. To reveal impact of this increase of MemTable size, we include an experiment of RocksDB whose MemTable is configured to be of 1.6GB, the peak size of WipDB's MemTables in the experiments. This RocksDB's results are shown as '*RocksDB-1.6G*' in Figure 6a. The throughput doesn't improve. A larger Memtable does help collect more writes for larger batched I/O and improved I/O efficiency. However, increasing the Memtable size will have diminishing return on fast devices and the dominant factor on the stores' performance is the amount of I/O, which is determined by a store's WA. PebblesDB's WA is also 2× of that of WipDB [3] The total amount of I/O during the writes is shown in Figure 6c. PebblesDB's extra writes are caused by having more levels and constantly splitting SSTables to generate new guards, while WipDB maintains at most three levels and conducting bucket splitting only when a bucket reaches its the size limit.

Since WipDB has a consistently low WA, its throughput remains stable and high at about 0.8 Mops/s. It is worth noting that initially WipDB's WA increases as the store grows. At the moment it reaches its peak value, 3.14 as shown in Figure 6b, the overall WA drops slightly. The reason is twofold. On the one hand, as WipDB starts to split, the number of buckets grows, and more data can be stored at Levels 0 and 1. Hence the WA becomes smaller with fewer levels. On the other hand, the workload contains update requests, which makes the size of new SSTables generated by compaction smaller than the total size of the SSTables being compacted.

WipDB's MemTable. To improve its memory access efficiency, WipDB initializes with hash-based MemTable, instead of skiplist. To assess the impact of this design, we include a version of WipDB that is initialized with skiplist-based MemTable, marked '*WipDB-S*' in Figure 6a. As shown, WipDB's write throughput is 2× of that of WipDB-S. WipDB-S's throughput remains stable at a lower rate (around 0.4 Mops/s), while its WA ratio is as low as that of WipDB. With the increase of bucket count from 100 to around 800, the working set of the MemTables grows much larger than the CPU cache size. Echoing observations shown in Figure 3, the degraded throughput of WipDB-S is due to its use of sorted skip list that causes much more CPU cache misses in its index walk than use of hash table. It is noted this issue of memory access performance arises only in the design of a high-performance KV store using high-speed storage devices.

Responding to changing key distribution. Real-world workloads are usually skewed [5] and their key distribution is likely to change over time, causing some buckets in WipDB to grow much faster and having more (sub)levels than the others. WipDB addresses this issue with bucket splitting. To evaluate how WipDB responds to key distribution change, we run a WipDB store initially with only one bucket. In the meantime, we separate the entire key range into four equal-size and

[3]In the experiment with PebbleDB, we manually change its opensourced code by setting *top_level_bits*, a variable controlling probability of generating guards, from 27 to 31 to reduce number of guards. This is necessary to allow PebbleDB to finish its execution. Otherwise, the program would run out of memory (on our server with 64GB memory) when the store reaches two billion KV items.



(a) Write Throughput (b) Write Amplification (c) IO at Different Levels

Fig. 6: *Write performance. "WipDB" uses the hash table for MemTable. "WipDB-S" uses skiplist. In (c) percentages of read and write amount at each level of a store are marked in the graph. The total read and write amounts, including those for compaction, are shown under respective graphs.*

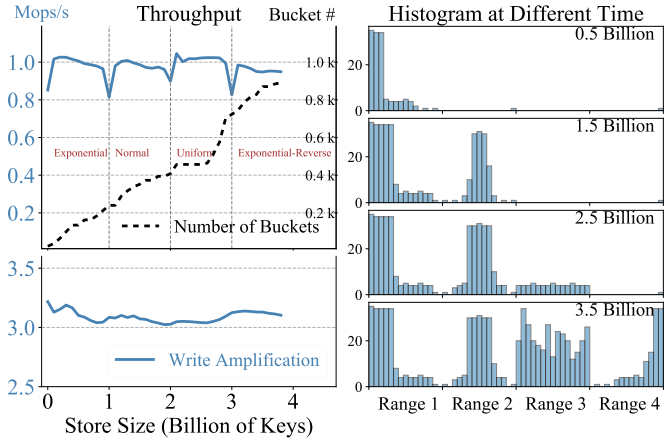


Fig. 7: *Write throughput when key distribution changes. Blue bars in the right graph show the distribution of buckets (number of buckets in each 1/60 key space) in the key space at four sample time points when the store reaches 0.5, 1.5, 2.5 and 3.5 billion items, respectively.*

non-overlapping regions, and serve four write workloads with different key distributions, each writing to a different region, one at a time, to simulate a workload of dynamical key pattern changes. The four regions, in the order of their key ranges, receive KV items (16 B keys and 100 B values) of exponential, normal, and uniform key distributions, and then exponential again with its key order reversed, respectively. Each region receives 1 billion KV items. The experiment results are shown in Figure 7. As shown in the left graph, the number of buckets increases with the increase of store size. The increase rate slows whey the workload switches to the uniform distribution without having skewed writes to rapidly fill and split a subset of buckets. The right graph of Figure 7 shows that the bucket distributions consistently match the key distributions in the key regions, demonstrating WipDB's adaptability in its bucket placement for equalized bucket sizes. During the execution, the WA may be modestly reduced. When buckets become filled and are then split into small ones with much fewer (sub)levels, their WAs accordingly become smaller. It is worth noting that the write throughput is the lowest at the moment when key distribution switches. Because new buckets are always generated by splitting existing buckets, this bucket splitting for accommodating new a key distribution leads to the throughput loss.

Note that in the WipDB's design, we do not assume an advance knowledge on number of buckets to be used and how the buckets should cover the key space (the store can be initialized with only one bucket). We only assume a long-term relatively stable key distribution to prevent an extreme scenario that may lead to excessively large number of buckets. In the scenario, some buckets receive many KV items and are filled. They are then split into new near-empty ones. The new buckets would receive few KV items afterwards. The consequence is that a very large number of buckets/MemTables leads to serious cache misses and compromised system's performance. However, from the experiment results we understand that the assumption on the key distribution does not have to be strong. The system's performance is highly tolerant to the change of the distribution and bucket count increase.

C. Read Performance

To evaluate read performance, we first build a store of 1 billion KV items (about 100GB). We then use eight threads to read items until another thread finishes sending 300 million write requests at a rate up to 150 Kops/s (by inserting time delay between requests). The read and write throughput is shown in Figure 8 (Note that the read Y axis is on the right of the graphs). In the meantime, to evaluate the impact of WipDb's read-aware compaction (RC) design, we include a version of WipDB that disable RC, marked 'WipDB-DRC'.

In the experiment where read requests have no locality (uniform), as shown in Figure 8(a), read throughput of all the four stores become lower after the write requests arrive. However, only WipDB sees its read throughput recovered during the writes, while others' read throughput keep dropping and then stay at its low level. This is mainly due to WipDB's consistently lower WA values. *WipDB-DRC*'s read throughput

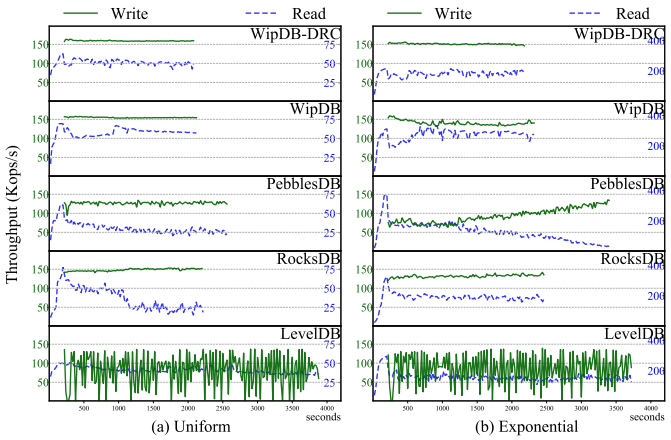


Fig. 8: *Throughput with mixed read/write requests. One thread sends 300 million random (uniform) write requests at the rate of 150 Kops/s(if possible). Eight threads send read requests until all the writes finish. WipDB that Disable Read-aware Compaction is marked as 'WipDB-DRC'.*

shows slight difference with *WipDB*. As there exists weak access locality, RC makes a minor contribution to improving performance. LevelDB uses only one thread for compactions and aggressively compacts SSTables to the last two levels. Accordingly its write throughput is intensively fluctuated.

We then repeat the experiment with exponential key distribution for the read requests. The results are shown in Figure 8(b). With a very strong access locality the read throughput is multiple times higher than that with the uniform key distribution. When the write requests arrive, all stores, except WipDB, observe much lower read throughput. Only WipDB roughly maintains its read throughput with some fluctuations. During the time period, WipDB's read throughput often more than doubles that of other stores. As shown, WipDB-DRC's read throughput is 30% lower than WipDB. This is apparently the consequence of WipDB's read-aware compaction scheduling design that leaves much of the compaction in the key range with light reads off the read requests' critical path. WipDB identifies read-intensive buckets and applies more aggressive compaction with priority on them. Therefore, the buckets that serve most read requests can be compacted faster and have fewer sublevels for fast read. Therefore, among all the stores WipDB has the lowest read latency, as shown in Table I.

TABLE I: Read Latency for 99 Percentile

Store	Uniform	Exponential
WipDB_DRC	439 us	247 us
WipDB	365 us	190 us
PebblesDB	1698 us	324 us
RocksDB	765 us	293 us
LevelDB	526 us	249 us

D. Impact of WAL on Restart Time

A WipDB store has hundreds of or even more MemTables. New KV items from different MemTables are written to a common log. This will lead to a log file much larger than

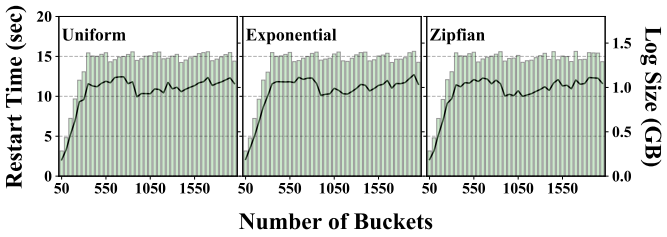


Fig. 9: *Restart time (curves) and log size (bars).*

those for stores using only one MemTable. When the store unexpectedly crashes and requires a re-start, it may take a longer time period to read the log for a restart. Figure 9 shows the log size and the restart time when a store is built with write requests of different key distributions at a size in terms of number of buckets. Initially, the log and the restart time grow linearly with the number of buckets. When the WipDB store reaches around 400 buckets, the log size and the restart time stay stable at around 1.5GB and 12 seconds, respectively.

E. Results of the YCSB Benchmarks

The Yahoo! Cloud Serving Benchmark (YCSB) [17] is a popular benchmark used to evaluate performance of NoSQL databases. We modify *db_bench* to support YCSB benchmarks and run eight threads and send 8 million requests for each workload. All the stores are pre-loaded with 1 billion (around 100GB) KV items. The results of the benchmarks are shown in Figure 10.

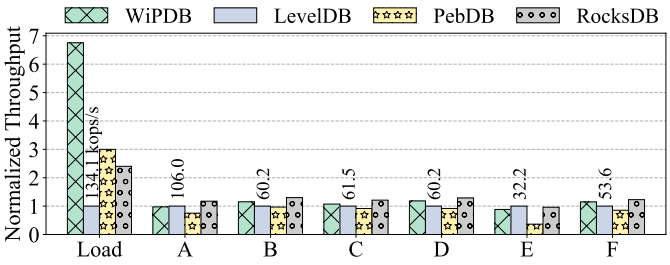


Fig. 10: *Throughput of YCSB benchmarks*

For the all-write workload ("Load") that pre-loads 1 billion items into the store, as expected WipDB has a much higher throughput than other stores. The other workloads consist of only or mostly read requests. WipDB is proposed to mainly optimize write operations so as to improve performance of writes and that of reads concurrent with writes. When writes accounts for only a fraction of total I/O load in the workload, WipDB is not expected to make a substantial difference. As shown, Workloads B and C have a 95/5 reads/write mix and a 100% read, respectively. WipDB's read throughput is comparable to others' performance. It's a little lower than that of RocksDB. Note that RocksDB is a production-level system that has been carefully engineered with numerous optimizations. WipDB is an experimental prototype with only limited optimization efforts. Therefore, this small performance gap is not a surprise. Though Workload A has a 50/50 reads/write mix, its 50%

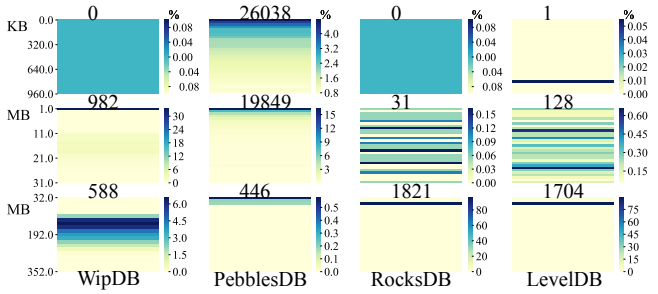


Fig. 11: *File size histogram. The number above each figure is the number of files within the size range.*

TABLE II: Latency for 99 Percentile

Store	A	B	C	D	E	F
WipDB	349	344	343	323	1133	351
LevelDB	253	342	328	341	688	364
PebblesDB	443	534	543	603	3008	671
RocksDB	237	241	241	241	638	248

read requests contribute over 90% of the execution time (as each read usually needs to read an entire data block). Therefore, the performance gap still exists. Nevertheless, for almost all the workloads, WipDB outperforms LevelDB and PebblesDB. Workload E consists of short range queries that require searching a store's every (sub)level. WipDB has more sublevels than levels of LevelDB and RocksDB, and thus has a higher read amplification. The read latency (us) is shown in Table II. WipDB's latency is comparable to that of LevelDB, expect for workloads A and E. PebblesDB has the highest latency because the randomly chosen guards cause file fragmentation. As shown in Figure 11, more than half of its file are smaller than 1 MB. Meanwhile, PebblesDB has over 20× files than the other store, causing more file-system overhead.

V. Related Work

LSM-tree has become the most popular data structure for the storage layer of NoSQL databases due to its optimized write performance compared to other structures such as B+-tree. LSM-tree achieves this by avoiding expensive in-place writes and moving the internal data reorganization to the background. However, write amplification in LSM-tree-based systems can still easily go over 10×, which leaves a wide gap between the user-perceived performance and that offered by the low-level storage devices, the SSDs. Because of this, improving write efficiency in LSM-tree-based KV stores has become a daunting task for persistent KV stores. This section discusses representative works on amelioration of the high WA.

Optimizations for efficient compaction. Real-world KV workloads often demonstrate skewed patterns [5], which have been leveraged by researchers to apply specialized optimizations. For example, bLSM [18] uses a merge scheduler that aims to minimize write stalls by coordinating compaction operations across multiple levels. Thonangi et al. [19] introduces ChooseBest, a compaction policy that selects an SSTable at L_k with the fewest overlapping SSTables at L_{k+1} to minimize the merge cost. Skip tree [20] allows KV items to be written to a deeper level without going through the level-by-level merges. VT-tree [21] reduces disk writes by reusing existing data in the old tables. TRIAD [22] takes advantage of the skewed workload where hot keys are likely to be short-lived. It identifies the hot keys and keeps them in the MemTable and the WAL instead of moving them to the Level-0 SSTables. In this way, the write traffic to the top levels can be effectively reduced. The above optimization are orthogonal to WipDB and may help to further improve the efficiency of WipDB.

The tiering merge scheme. In LevelDB [8] and RocksDB [9], compaction operation needs to rewrite a significant amount of data at the deeper level (L_{k+1}) but moves only $1/N$ of that amount of data from L_k. The tiering merge scheme was proposed to eliminate the significant rewrites. By merging multiple SSTables from $Level_k$ and writing to $Level_{k+1}$ without rewriting any $Level_{k+1}$ data, write amplification can be effectively reduced to only L_{max}, the number of levels in the store. wB-Tree [23] uses a B^+-tree-like structure to organize the tables to maintain a small L_{max}. Similarly, LSM-trie [10] uses a prefix-tree (trie) structure for the same purpose. However, since both wB-tree and LSM-trie depend on hashing to maintain balanced tree structures, they gives up the ability of performing range operations. sDB [11] employs a probabilistic method to partition the keys at each level to enable tiering with the range-query capability retained. SifrDB [24] also employs tiering. In LSM-Bush [25] and Dostoevsky [26], a lazy-leveling scheme is introduced to use tiering for levels from 1 to $L_{max} - 1$ and use leveling at Level L_{max}, the last level. In this way, the WA is $O(L_{max} + T)$, where T is the size ratio between adjacent levels. Different from the above schemes, WipDB employs a partitioning approach to limit its bucket's size and accordingly the number of levels. As a result, WipDB achieves a lower write amplification ($O(L_{max} + \frac{N}{N-1})$) (N is the split-ratio) without sacrificing read efficiency. Both of L-Store [27] and WipDB aim to improve write performance. Additionally, L-store converts the data from a write-optimized organization to a read-friendly one to serve read-intensive OLAP workloads. This is similar to regular LSM-based stores that finally compact all data into the last level if writes do not keep coming. WipDB is designed to opportunistically improve read performance even with write requests by prioritizing hot buckets' compaction for fast read.

Key-value separation. It is observed that rewriting the values in KV items can contribute to a major amount of I/O in the compaction for relatively large values, compared to the size of metadata and key which is usually tens of bytes. WiscKey [12] proposed a simple yet effective method, KV separation, to write values into a log and keep them from the compaction. However, the value log requires regular GC operations to reclaim free space. The log GC can be particularly expensive under skew real-world workloads. Significant amount of cold data needs to be consistently removed, which drives up the store-wise write-amplification to up to 20× and offsets the benefit of KV separation [13]. To reduce this GC overhead, HashKV [13] replaces the log with a sophisticated mechanism

that divides the log into partitions and separates the cold items from the hot data, which again shifts the overhead to read by adding another layer of indirection. WipDB solves the high write-amplification issue by directly partition the key space without creating any indirection.

In-memory key-value stores. For applications demanding high concurrency and low latency, in-memory key-value stores are always preferred. HotRing [28] is proposed as a hotspot-aware and lock-free design to speed up multi-core performance for highly skewed workloads. Redis and Aerospike [29] provide a hybrid solution which provides memory-access speed as well as on-disk data persistency when specified conditions are met. In contrast, WipDB is still a on-disk KV store providing always data persistency and expecting on-disk data access for most read requests.

VI. Conclusion

We introduce WipDB, a key-value store designed to manage small key-value items in a storage system of large capacity. By introducing approximate sorting and the write-in-place LSM-tree scheme, WipDB minimizes write amplification for LSM-tree-based KV stores. Meanwhile, the read-aware scheduling of compaction moves most compaction off the critical path of read service. Our results show that WipDB can significantly improve for both write and mixed read/write workloads. Source code of our WipDB implementation is available at https://gitlab.com/sjiang-lab/wipdb .

Acknowledgment

We are grateful to the paper's anonymous reviewers who helped to improve the paper's quality. This work was supported by US National Science Foundation under Grants CCF-1815303.

References

[1] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, *et al.*, "Betrfs: Write-optimization in a kernel file system," *ACM Transactions on Storage (TOS)*, vol. 11, no. 4, pp. 1–29, 2015.

[2] K. Ren and G. Gibson, "Tablefs: Enhancing metadata efficiency in the local file system," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 145–156, USENIX Association, 2013.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, (New York, NY, USA), pp. 53–64, ACM, 2012.

[6] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook," in *USENIX NSDI '13*, pp. 385–398, 2013.

[7] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[8] Google, "Leveldb." https://github.com/google/leveldb, 2020.

[9] Facebook, "Rocksdb." https://rocksdb.org.

[10] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, (Berkeley, CA, USA), pp. 71–82, USENIX Association, 2015.

[11] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 497–514, ACM, 2017.

[12] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, 2017.

[13] H. H. Chan, C.-J. M. Liang, Y. Li, W. He, P. P. Lee, L. Zhu, Y. Dong, Y. Xu, Y. Xu, J. Jiang, *et al.*, "Hashkv: Enabling efficient updates in {KV} storage via hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 1007–1019, 2018.

[14] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, "Flodb: Unlocking memory in persistent key-value stores," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), pp. 80–94, ACM, 2017.

[15] zhichao Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 209–223, USENIX Association, Feb. 2020.

[16] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (USA), p. 47–60, USENIX Association, 2010.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," 2010.

[18] R. Sears and R. Ramakrishnan, "blsm: A general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, (New York, NY, USA), pp. 217–228, ACM, 2012.

[19] R. Thonangi and J. Yang, "On log-structured merge for solid-state drives," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 683–694, IEEE, 2017.

[20] Y. Yue, B. He, Y. Li, and W. Wang, "Building an efficient put-intensive key-value store with skip-tree," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 961–973, April 2017.

[21] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2013.

[22] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "Triad: Creating synergies between memory, disk and log in log structured key-value stores," in *USENIX ATC '17*, pp. 363–375, 2017.

[23] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, pp. 786–797, Feb. 2015.

[24] F. Mei, Q. Cao, H. Jiang, and J. Li, "Sifrdb: A unified solution for write-optimized key-value stores in large datacenter," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, (New York, NY, USA), pp. 477–489, ACM, 2018.

[25] N. Dayan and S. Idreos, "The log-structured merge-bush & the wacky continuum," in *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, (New York, NY, USA), p. 449–466, Association for Computing Machinery, 2019.

[26] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), pp. 505–520, ACM, 2018.

[27] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim, "L-store: A real-time OLTP and OLAP system," in *EDBT 2018, Vienna, Austria, March 26-29, 2018* (M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, eds.), pp. 540–551, OpenProceedings.org, 2018.

[28] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "Hotring: A hotspot-aware in-memory key-value store," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 239–252, USENIX Association, Feb. 2020.

[29] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Comput. Surv.*, vol. 51, Apr. 2018.