REXIO: Indexing for Low Write Amplification by Reducing Extra I/Os in Key-Value Store under Mixed Read/Write Workloads

Zizhao Wang$^{1,2[0009-0007-1786-1236]}$, Qiang Qu^{1} Nan Han^{4}, Zhelang Deng^{1}, Yizhuo Ma^{5}, and Xiaowen Huang^{3,1} Jintao Meng$^{1\boxtimes[0000-0002-6208-4102]}$

[1] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China {zz.wang,jt.meng}@siat.ac.cn
[2] University of Chinese Academy of Sciences, BeiJing, China
[3] Shenzhen University, Shenzhen, China
[4] Department of Health Technology and Informatics, The Hong Kong Polytechnic University, Hong Kong, China
[5] Laboratory of Intelligent Collaborative Computing,University of Electronic Science and Technology of China, Chengdu, China

Abstract. In mixed read/write (r/w) workloads, Key-Value stores that utilize Log-Structured Merge trees face significant write amplification (WA) due to frequent writing, resulting in extensive data compaction. While current LSM-based KV stores effectively address WA in write-heavy workloads, they struggle with increased block cache invalidation and increased r/w I/O conflicts in mixed r/w workloads. This paper introduces REXIO, a novel indexing approach to reduce WA. It decouples RAM from Solid State Drives (SSDs) and stores addresses of KV pairs in the In-RAM hashing table, which reduces buffer invalidations and eliminates unnecessary data reorganization to lower extra I/Os, thereby reducing I/O conflicts. Additionally, REXIO stores keys and values in different blocks and employs *In-blocking logging* to optimize updating and deleting. The experimental results show that REXIO achieves a 68.3% reduction of WA and 3.4x throughput compared to state-of-the-art LSM-Tree approaches.

Keywords: read/write heavy workloads · key-value store · write amplification

1 Introduction

Nowadays, the proliferation of data-intensive applications results in the crucial demands of effective large-scale data management [1–5]. Key-Value (KV) stores are typically used in such applications to enable efficient processing and retrieving of massive amounts of data [6]. Among various KV stores, Log-Structured Merge (LSM) based KV stores [7] have emerged as a preferred choice for their superior write performance [8, 9].

However, current LSM-based KV stores encounter a problem that the amount of data written to disk exceeds the actual data written by users (i.e., WA) [10] in mixed r/w workloads. For instance, writing 1MB data in a KV store might result in 5 MB written to disk as data reorganization, exemplifying a write amplification factor (WAF) of 5. This problem significantly increases r/w latency and reduces throughput [11], thus hindering overall data processing efficiency. In LSM-based KV stores, several strategies such as *tiering* [12], *skipping* [13], and *data skew* [14] have been explored to reduce WA (For more details, see Section 2). While LSM-based KV stores efficiently reduce WA in write-heavy workloads, they still pose two significant challenges for mixed r/w workloads. First, extensive writing frequently triggers *compaction* in LSM, which is a read-merge-sort-rewrite process and occupies massive I/O resources. This *compaction* process competes with the I/O requests of reads in such workload [15], increasing read latency. Second, continuous writes result in numerous block cache invalidations in RAM due to the deletion of the original block. As a result, the block cache associated with the read operation needs to be rebuilt, leading to prolonged read latency [16].

To address these challenges, this paper introduces an innovative indexing approach for KV stores called REXIO (i.e., Reduce Extra I/Os). The principle of REXIO is decoupling RAM from SSD to reduce WA in mixed r/w workloads. REXIO maintains an extendible hashing table in RAM, which manages the keys and the corresponding addresses of all KV pairs, facilitating data retrieval and substantially reducing buffer invalidations. Moreover, the decoupling design discards re-ordering on-disk data, eliminating unnecessary data reorganization (such as on-disk data reorganization in LSM trees) to lower extra I/Os (See definition in Section 3), thereby reducing r/w I/O conflicts. To optimize deleting, REXIO employs an *In-block logging* strategy on the SSD, transforming delete operations into sequential *binarycode* writes. Additionally, REXIO employs a policy of storing keys and values separately in different blocks to allow for efficient updating.

In summary, the contributions of this paper include:

– We propose an indexing approach for KV stores in r/w heavy workloads that decouples RAM from the SSD, eliminating the extra I/Os.
– We employ an In-RAM hashing table that stores the keys and addresses of persistent KV pairs to reduce buffer invalidation and avoid data reorganization.
– We also introduce the *In-block logging* within the index, designed to transform deletions into sequentially writing *bianrycode*.
– We conduct experiments on an NVMe SSD. The results demonstrate that our method significantly reduces WA in r/w heavy workloads.

2 Related Work

One way to optimize WA in LSM-based KV stores is to apply *tiering* to substitute *leveling*. These can be categorized into two methods: horizontal grouping and vertical hierarchy [17,18]. Vertical grouping involves distributing data across multiple parallel tiers, while vertical hierarchy organizes data in a layered structure. For instance, WB Tree [19] uses hash-partitioning for workload balance and a B^+-tree-like structure for self-balancing, where full nodes merge SSTables into their child nodes. The LWC-tree [20] uses a similar partitioned *tiering* design but adjusts key ranges for workload balance.

PebblesDB [12] uses a partitioned *tiering* design with vertical grouping to balance workload by selecting key *guards*. dCompaction [21] employs virtual SSTables to minimize merge frequency, triggering actual merges based on specific thresholds or queries. These four structures described above all share a similar high-level design based on partitioned *tiering* with vertical grouping. Zhang et al. [22] and SifrDB [23] also adopt a partitioned *tiering* design with horizontal grouping.

On the other hand, Skip tree [13] employs *merge skipping* to lift data directly to higher LSM tree levels, bypassing standard merges to reduce write costs. This necessitates *mutable* buffers at each tier to accommodate bypassed some standard level-by-level merges. Another approach is TRIAD [14], which separates *hot keys* from *cold keys*, efficiently managing *hot keys* in memory and using a transaction log on disk. WiscKey [10] offers an *separation key from value* solutions, followed by HashKV [24] and SifrDB [23]. It stores KV pairs in an append-only log, using the LSM-tree as a primary index for mapping keys to log locations. HashKV extends this by hash-partitioning the value log and independently garbage collection each partition, using a group-by operation on keys. While this separation greatly reduces the WA, it produces greater query latency.

3 Preliminary

We first introduce the symbols in Table 1 used in this paper. Next, we define mixed r/w workloads in KV stores. In our definition, it consists of two key operations: $W(k_i, v_i)$, which writes KV pair (k_i, v_i), and $R(k_i)$, which retrieves the value. The ordered sequence of these operations is denoted by $O = op_1, op_2, \dots, op_n$, where each op_i is either a read or a write. The distribution of reads and writes is modeled by the function $D(i, \alpha)$, where α represents the read-to-write ratio.

$$D(i,\alpha) = \begin{cases} 1 & \text{if } i \mod \left(\frac{1}{\alpha}+1\right) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This implies that for a read/write ratio of $\alpha = \frac{5}{5}$ (i.e., one read for every write), every 2^{th} operation in 1 is a read, reflecting a frequent interaction pattern that regularly alternates between read and write operations.

Problem Formulation. Consider a dataset $\mathcal{E} = \{(k_1, v_1), (k_2, v_2), \dots, (k_n, v_n)\}$ composed of n KV pairs. The total size of the dataset is $|\mathcal{E}|$. In KV stores, the write amplification factor (WAF) is defined as the ratio between the total amount of data written to disk and the amount of data written by the user:

$$A = \frac{\text{Total Data Written to Disk}}{\text{User Data Written}} \tag{2}$$

The goal of this paper is to minimize A by reducing unnecessary I/Os in mixed read/write workloads, thereby optimizing write performance.

Table 1: Symbols used in this paper

Symbol	Description	Symbol	Description
A	Write amplification	S_p/S_b	Page size/Block size
ht	In-RAM hashing table	N_l/N_d	The number of log/data pages in a block
S_B	Bucket size in hashing table	gd	Global depth of *extendible-hashing*
n	The number of inserted KV pairs	S_k/S_v	Size of key/Size of value

4 An Indexing for Reducing Extra I/Os

In this section, we introduce the detailed design of the proposed method, indexing for *Reduce Extra I/Os* on Solid State Drives.

4.1 Index Construction

We first present the construction process of REXIO. As shown in Figure 1, REXIO deviates from tree structure by decoupling RAM and SSD components, resulting in a streamlined process to reduce extra I/Os caused by operations like node splits and data reorganization. REXIO begins with organizing data within RAM using a special extendible hashing table. Unlike extendible hashing, where the SSD's page size constrains the bucket size, REXIO's buckets are not bound to the SSD's physical structure. REXIO allows flexible bucket sizing based on data schema without the need for I/Os during bucket splits.

Data organization in REXIO starts with an extendible hashing table in RAM. Keys are managed via a shift operation relative to *Globaldepth*, creating nodes that include the key, a deletion flag, and a *binarycode* indicating the KV pair's offset. These nodes are inserted into a *skiplist* within the respective bucket. Simultaneously, KV pairs are copied to a *write buffer* managed by RAM, with offsets dynamically adjusted by a write pointer, thus enhancing data organization and accelerating index construction without generating dirty data.

It is worth noting that the offset within a node is dynamically determined in RAM via a write pointer. This write pointer is managed together with a *write buffer*. As the write pointer is moved, the offset changes accordingly, allowing for efficient data organization and fast index construction. In addition, since our bucket size is not limited by the structure of the SSD, it obviates the necessity for extra I/Os typically associated with bucket splitting. Consequently, this approach negates the creation of dirty data within the SSD. The reorganization process is confined to RAM, streamlining data management.

On the SSD side, REXIO manages data based on logical block addressing (LBA), which involves a hierarchical structure where multiple LBAs constitute a *page*. Several such pages, in turn, form a *block*.[6] This configuration enhances data organization through a flexible page-block structure. Within RAM, REXIO capitalizes on this abstraction to optimize SSD resource management, fully exploiting the SSDs' sequential writing capabilities to improve efficiency.

[6] In this paper, *page* and *block* refers to a collection of LBAs

When writing data to the SSD in REXIO, the process is meticulously sequential, beginning from the first available page of the first unoccupied block. This sequential writing strategy is employed to maximize the advantage of SSDs in handling sequential write operations. By organizing data writes in this way, REXIO ensures that the SSD's high-speed sequential I/Os are fully harnessed, leading to improved write throughput. As the example in the Figure 1 shows, a block contains two pages. When the buffer in RAM overflows, the data is first written to page 1, then page 2. At this point, block 1 is full, and then the data will be written to block 2.

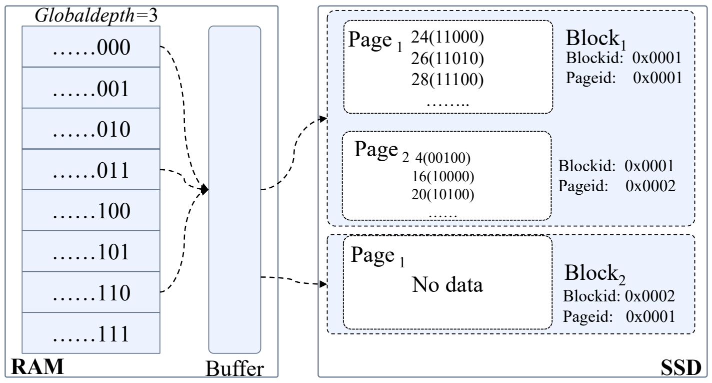


Fig. 1: Overall design of REXIO.

4.2 Operation Strategies

Insert. The insert process can be summarized through the pseudo-code presented in algorithm 1. This algorithm outlines the key steps in inserting a data entry into REXIO. The initial step involves calculating the associated bucket number in the hashing table using the *key*, as represented in line 1 of algorithm 1. Subsequently, using the bucket number, the associated *skiplist* within that bucket is queried to find the position for inserting the key, as outlined in lines 4-7 of algorithm 1. After determining the key's insertion location, if the key has not been previously added (see lines 15-26), a new node is created within the *skiplist*. The node's flag is set to 1, as shown in line 23, and the *binarycode* is provided by the global write pointer. Concurrently, as represented in line 21, the corresponding data entry is temporarily stored the *write buffer*. Upon reaching its threshold capacity, the buffer's contents are batchly flushed to the SSD. If the key is located and its associated flag is set to 0 (see lines 12-14), it indicates that the entry has been previously deleted. In response, the entry is reinserted into the *write buffer*. Subsequently, as detailed in line 12, the flag is reset to 1, and the *binarycode* is updated. If the key is identified with an accompanying flag set to 1, as illustrated in lines 9-10, it indicates that the data entry has been previously inserted. Consequently, the return a valid flag. Figure 2 shows the whole process of inserting a data entry with key 8. Initially, the key is recorded in the hashing table. Concurrently, the data entry is written sequentially to the *write buffer*, and when the buffer overflows, it is sequentially committed to the SSD.

ALGORITHM 1: Insert Algorithm of REXIO

```
   Input: $Key, Val$
   Output: true/flase
1  head←ht[key&(1 $c$ gd)-1]→local;
   /* c is left shift                                          */
2  temp = head →next;
3  update[MAX_LEVEL];
4  for i=head→level; i≥0; –i do
5      while temp→next[i] && temp→next[i]→key¡key do
6          temp = temp → next[I];
7      update[i] = temp;
8  if temp→key == key then
9      if temp→flag == 1 then
10         return true;
11     else
12         temp→flag =1;
13         temp→binarycode = async_write($Key, Val$);
14         ++Head→number;
15         return true;
16 else
17     v=RandomLevel();
18     if v > head→level then
19         for i=head→level; i≥v; ++i do
20             update[i] = Head→head;
21         head→level = v;
22     tbinary = async_write(hashkey,hashvalue);
23     ++Head→number;
24     new = NodeCreate(key,tbinary,v);
25     for i=0; i≥v; ++i do
26         new → next[i] = update[i]→next[i];
27         update[i]→next[i] = new;
28 return true;
```

Retrieval. Retrieval is similar to insert: the key is used to compute the *skiplist* where the key may be located. If the key is not found or the flag of the searched node equals 0, then return. If found, read the page from SSD based on *binarycode*. In addition, to speed up the data retrieval process, we use an *LRU buffer* to cache the retrieved data entries.

Delete and Update. The delete procedure is simple: find the node based on the key. If the entry has not been deleted, then set the *flag* to 0, and write the *binarycode* into *log buffer*. The update procedure can be divided into a delete operation and an insert operation of a new KV pair.

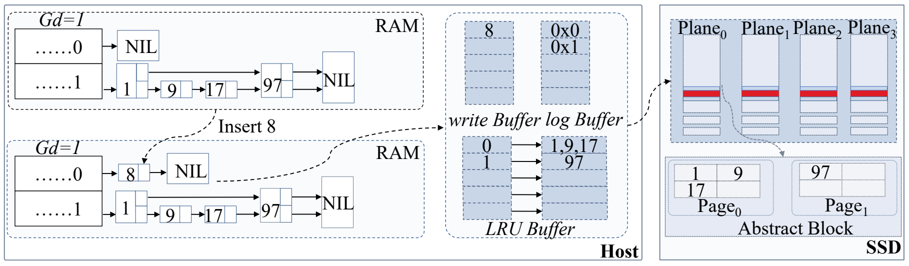


Fig. 2: An example of inserting operation in REXIO.

4.3 Improvement for Delete

To optimize deletes in REXIO for mixed r/w workloads, it employs *In-block logging*. This method transforms these operations into more efficient writes, prompting the speed of the delete operations in REXIO. Specifically, REXIO divides each block into data and log pages. Data pages are reserved for storing data, while log pages are dedicated to storing logs, that is, *binarycode* (We also call it fixed-length *binarycode*). The allocation of data and log pages within a block is not arbitrary. It is crucial to ensure that alongside the data pages for incoming data, there is a sufficient reserve of log pages to accommodate the deletion logs. To strike this balance, we pre-calculate the required number of each page type in a block, leveraging Equation 3 for precise allocation between log and data pages. N_{data} and N_{log} denote the number of data pages and log pages, respectively, in a block, B_b denotes the size of *binarycode*, and n denotes the total amount of data that will be written to this block.

$$n = \frac{N_{\text{d}} \cdot S_p}{S_v}, \quad S_p \cdot (N_{\text{d}} + N_{\text{l}}) = S_b, \quad n \cdot B_b = S_p \cdot N_{\text{l}} \tag{3}$$

In addition, we observe that employing fixed-length *binarycode* (FLB) results in suboptimal space utilization, mainly when dealing with comparatively large key and value sizes. Consequently, we have adopted a variable length *binarycode* (VLB) approach to mitigate space wastage further. Specifically, our approach is to calculate the number of the data in the block based on the page number and in-page offset and then convert the number into a variable byte *binarycode*, i.e., the highest bit of each byte is used as the marker bit. When the marker bit is 0, the subsequent 7 bits of the next byte pertain to the same data entry. Conversely, when the marker bit is set to 1, it indicates the termination of a *binarycode*. To calculate the number of log pages and data pages in a block with VLB, we can use a simple statistic that counts the *binarycode* of each data (key or value) to be written to the block. When $\sum_{i=1}^{n}(B_{vb}^{i} + S_d) \geq S_b$ (B_{vb}^{i} denotes the size of the VLB of the i_{th} data written to the block), it means that the block is full, and the next write operation will be to a new block.

4.4 Improvement for Update

Reflecting on the asymmetry in key-value updating, which means values frequently undergo modifications while their corresponding keys remain unchanged.

We propose a strategy for storing keys and values separately in SSD, inspired by Wisckey [10]. The design reduces WA and optimizes updates by avoiding unnecessary key rewriting. REXIO partitions the blocks into key and value blocks. While key blocks are allocated for storing both keys and associated logs, value blocks are dedicated to holding values and corresponding logs. Within a block, pages are divided into data and log pages. When a value within a KV pair is updated, REXIO initiates a series of operations to ensure data integrity and accuracy. The process begins with REXIO searching for the corresponding node in RAM using the key. If the node is successfully located and its flag is set to 1 (indicating that the value is valid). Concurrently, a deletion log for the old value is created. The new value and this deletion log are then written to the value block where the old value was previously stored. Following the update, the *binarycode* corresponding to the key in RAM is updated to reflect the recent changes.

4.5 Crash Recovery

In the event of a system (process) crash, REXIO undergoes a critical recovery process. Upon restart, REXIO scans the entire SSD to identify all *key* blocks. The data from the data pages in these *key* blocks is then read into RAM to rebuild the In-RAM hashing table. Subsequently, the log pages from the *key* blocks are used to perform the required redo operations to restore the *extendible hashing* table to its pre-interrupt state. Finally, the block buffers are initialized, and any unflashed data is rewritten to these buffers.

For space complexity, suppose inserting n KV pairs. For every stored key, we also need to store an address value location in VLB. We use $\lceil b/7 \rceil$ to compute the address, b is the *binarycode* representing the location of the value. Then the space upper bound of n KV pairs is:

$$S(n) = n \times (S_e + \lceil S_e/7 \rceil + \lceil b/7 \rceil) \tag{4}$$

This also is a linear trend with the amount of data inserted.

5 Performance Evaluation

In this section, we evaluate the performance of REXIO by comparing it with other KV stores. Our goal is to evaluate the performance of REXIO for WA in a write-heavy workload and in a mixed r/w workload.

5.1 Experiment Setup

Settings. Our experiments are conducted on a Huawei server with a 40-core Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz and bolstered by 256GB of RAM. The server, running Ubuntu 20.04.4 LTS with Linux kernel 5.4.0-122-generic on x86_64, is equipped with a Samsung 980 SSD, which offers maximum sequential read and write speeds of 3500 MB/s and 3000 MB/s, respectively. The SPDK v21.01 [25] is used in REXIO for I/O management. The source code of our implementation is provided in Github[7].

[7] `https://github.com/Zizhao-Wang/REXIO`

In REXIO, we configure the page size to 64KB, corresponding to the size of the *write buffer*. Additionally, we set the block size as 512KB and allocate 32MB for the *LRU buffer*. Other KV stores in our experiment retain their default parameter configurations for comparative analysis.

Test Methods and DataSet. To evaluate the performance of REXIO in mixed r/w workloads and write-heavy workloads, we compare our approach with three categories of LSM-based KV stores introduced in [26]: SifrDB [23], representing horizontal *tiering*; WipDB [27], embodying vertical *tiering* approaches; the widely-recognized LevelDB (v1.21) [28] and RocksDB (v8.10), serving as classic implementations of *leveling* in LSM tree. We use the db_bench [28] to generate a 268GB dataset for the write-heavy workload. The mixed r/w workload with 134GB is generated similarly to YCSB [29].

5.2 Write Amplification Analysis

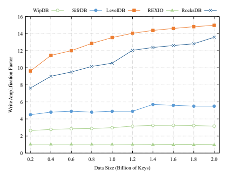
We evaluated the WAF across different KV stores by writing 2 billion KV pairs (268 GB). As shown in Figure 3a, the horizontal axis represents the number of pairs in billions, while the vertical axis indicates the WAF.

WipDB's WAF peaks at 3.25, around 1.6 billion KV pairs, then slightly decreases. SifrDB's WAF steadily rises, stabilizing at around 5.70. LevelDB and RocksDB's WAFs progressively increase to 14.99 and 12.60, respectively. REXIO consistently maintains the lowest WAF, decreasing from 1.04 to 0.98 (a 68.3% reduction of WipDB) as data volume increases. REXIO's WAF is less than 1 because it stores key and value separately. With the increase of duplicate data, REXIO update data only needs to write the new value and *binarycode* old value. Moreover, as the value size increases, the size of the VLB in REXIO decreases.

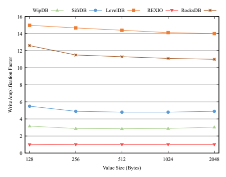
Next, we examined the impact of varying value sizes, as shown in Figure 3b. The x-axis denotes value size in bytes, and the y-axis measures the WAF. For LevelDB and RocksDB, WAF decreases as value size increases, aligning with their leveling compaction methodology, which writes fewer KV pairs for larger values. SifrDB's WAF initially decreases with larger values but stabilizes around 4.9, indicating limited optimization. WipDB's WAF decreases to a minimum of 2.858 at a 512-byte value size before increasing again. This stabilization and subsequent rise in WAF, along with increased latency for larger values, suggest that while SifrDB and WipDB handle smaller values efficiently, they are less optimized for larger sizes. REXIO consistently maintains a low WAF that decreases slightly with increasing value sizes, demonstrating its efficiency across various sizes.

Finally, we analyzed WAF in relation to key ranges in Figure 3c. WipDB improves its I/O efficiency, with WAF decreasing from 3.16 to 2.67 as the key range narrows to 1/8, indicating more efficient compaction. SifrDB's WAF decreases slightly to 4.90 as the key range quarters, then stabilizes. LevelDB's WAF decreases from 14.98 to 12.07 with reduced key ranges, and RocksDB shows a similar trend, dropping from 12.6 to 9.95. The reduction in WAF with narrower key ranges may be due to potential in-memory merging. REXIO consistently maintains a low WAF, demonstrating its robustness and efficiency across varying key ranges.

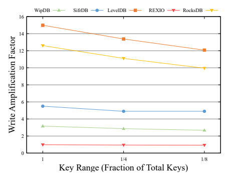
Following the analysis of WAF across dataset size, value sizes, and key ranges, we turn our attention to the throughput capabilities of the evaluated KV stores. Figure 3d illustrates the throughput results (268GB dataset with 2KB value), with the horizontal axis specifying the KV stores and the vertical axis quantifying the throughput in terms of thousands of operations per second (Kops/sec). The histogram distinctly illustrates that REXIO towers over its competitors, achieving a throughput of approximately 123.66 Kops/sec when writing a substantial 268.2 GB dataset. WipDB, while lagging behind REXIO, still manages a respectable throughput of 27.40 Kops/sec. On the other hand, LevelDB and RocksDB have similar throughputs of 6.98 Kops/sec and 6.44 Kops/sec, respectively. SifrDB has the lowest throughput of 3.58 Kops/sec. This stark contrast in throughput performance suggests that REXIO's mechanisms can have better performance for handling large-value writes.



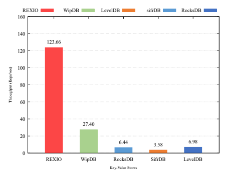
(a) WAF with different data size



(b) WAF with different value size

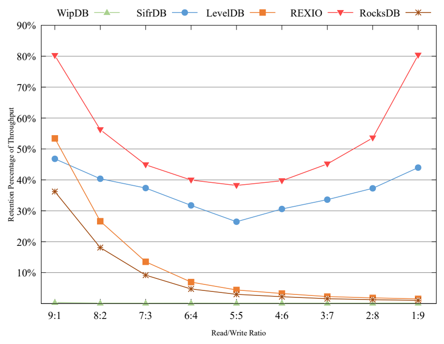


(c) WAF with different key range

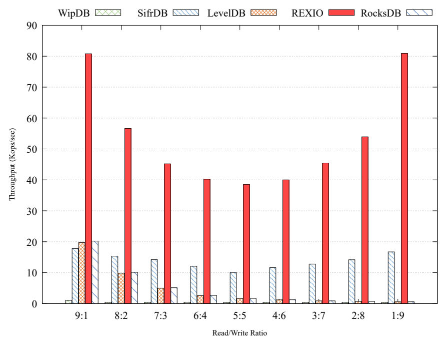


(d) Throughput in different KV stores

Fig. 3: Comparative analysis of WAF and throughput of all compared KV stores.



(a) Percentage in different ratio



(b) Throughput in different ratio

Fig. 4: Comparative analysis of relative throughput retention and absolute throughput of all compared KV stores in different r/w ratios.

Our experimental spanning dataset size, value size variations, and key range considerations demonstrate REXIO's performance in write-intensive workloads. The result also reveals REXIO's efficiency in minimizing the WAF while maintaining high throughput.

5.3 Performance in r/w Heavy Workload

We then evaluate the performance of all KV stores under a r/w heavy workload. Figure 4a illustrates the relative throughput retention as the ratio of the r/w changes, while Figure 4b presents the actual throughput figures across the same r/w spectrum.

For WipDB, starting retention of 0.35% in read-intensive conditions has a notable decline across all r/w ratios, barely maintaining 0.15%. SifrDB demonstrates an initial 46.0% retention, a dip to 31.0% in mixed r/w operations, and a recovery to 49.0% as write operations intensify. LevelDB exhibited an initial data retention rate of 53.0%, which then experienced a pronounced decline before stabilizing at approximately 1.5%. A similar pattern emerged in RocksDB, demonstrating an initial retention of 36.0% and tapered to an eventual 1.0%. We can see that as the r/w ratio gradually increases, especially at the most intensive r/w condition(i.e., ratio is 5:5), the throughput of LSM-based KV stores drops to a minimum. Throughput retention of REXIO dropped to a low of 40%. As the r/w ratio gradually changes, his throughput retention gradually increases, eventually reaching 80%. This situation is because read operations are very intensive at this time as they are time-consuming. Although REXIO and SifrDB show a similar trend, REXIO is different from the LMS-based KV store in principle. Increasing the *LRU buffer*'s size, REXIO will bring a more intuitive effect on the improvement because of the decoupling design. However, due to the compaction in LSM-based KV stores, other methods may only bring a slight improvement when increasing the buffer size.

Regarding throughput, REXIO starts with 80 Kops/sec in a 1:9 r/w and 81 Kops/sec in a 9:1 w/r. REXIO's high initial throughput is attributable to its capacity to ascertain the existence of KV pairs directly within RAM, thereby circumventing the need for extra I/Os. As the written data increas, most data have been written to the SSDs, leading to a diminished retention rate for REXIO. Despite this reduction, REXIO's retention rate remains better than other LSM-based KV stores, and its absolute throughput surpasses SifrDB's 3.4x when the r/w ratio is 5:5.

6 Conclusion

This paper introduces REXIO, a novel approach designed to significantly reduce WA in r/w heavy workloads for KV stores. At the heart of REXIO is the innovative RAM-SSD decoupling strategy, complemented by our unique *binarycode* logging technique. The logging technique uniformly transforms all operations in the KV store into sequential writes, particularly optimizing delete and update operations through *binarycode* conversion. This design simplifies the data structure and I/O management in SSD-based KV stores, leading to a substantial reduction in WA by minimizing I/Os.

7 Acknowledgement

This work was partly supported by the National Key Research and Development Program of China Grant No. 2023YFF1206103, the Key Research and Development Project of Guangdong Province under grant No. 2021B0101310002, Shenzhen-HongKong Joint Funding Project (A) under Grant No. SGDX20230 116092056010, Shenzhen Key Laboratory of Intelligent Bioinformatics under Grant No. ZDSYS20220422103800001.This work was also supported by Shanghai Zelixir Biotech Company by its Joint Lab of Zelixir-SIAT and Tencent with 3 years of continuous funding support. This work was also sponsored in part by the 3rd Xinjiang Scientific Expedition Program (2021xjkk1300), Shenzhen Science and Technology Plan Project (Shenzhen-Hong Kong-Macau Category C, No. SGDX20220530111001003) and the SIAT-GD Big Data&AI Joint Innovation Laboratory (E3Z092). We also thank support from Guangdong Basic Research (2023TQ07A264). We also thank Intel China for providing the Persistent Memory Platform.

References

1. Cao, Z., Dong, S., Vemuri, S., Du, D.H.: Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In: 18th USENIX Conference on File and Storage Technologies (FAST 20). pp. 209–223 (2020)
2. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: A survey. IEEE Transactions on Knowledge and Data Engineering **27**(7), 1920–1948 (2015)
3. Siddiqa, A., Hashem, I.A.T., Yaqoob, I., Marjani, M., Shamshirband, S., Gani, A., Nasaruddin, F.: A survey of big data management: Taxonomy and state-of-the-art. Journal of Network and Computer Applications **71**, 151–166 (2016)
4. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD international conference on Management of data. pp. 239–250 (2015)
5. Chen, G.J., Wiener, J.L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., Yilmaz, S.: Realtime data processing at facebook. In: Proceedings of the 2016 International Conference on Management of Data. pp. 1087–1098 (2016)
6. Idreos, S., Callaghan, M.: Key-value storage engines. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 2667–2672 (2020)
7. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**, 351–385 (1996)
8. Li, Y., Chan, H.H., Lee, P.P., Xu, Y.: Enabling efficient updates in kv storage via hashing: Design and performance evaluation. ACM Transactions on Storage (TOS) **15**(3), 1–29 (2019)
9. Lu, K., Zhao, N., Wan, J., Fei, C., Zhao, W., Deng, T.: Tridentkv: A read-optimized lsm-tree based kv store via adaptive indexing and space-efficient partitioning. IEEE Transactions on Parallel and Distributed Systems **33**(8), 1953–1966 (2021)
10. Lu, L., Pillai, T.S., Gopalakrishnan, H., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Wisckey: Separating keys from values in ssd-conscious storage. ACM Transactions on Storage (TOS) **13**(1), 1–28 (2017)
11. Lee, E., Kim, J., Bahn, H., Lee, S., Noh, S.H.: Reducing write amplification of flash storage through cooperative data management with nvm. ACM Transactions on Storage (TOS) **13**(2), 1–13 (2017)

12. Raju, P., Kadekodi, R., Chidambaram, V., Abraham, I.: Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 497–514 (2017)
13. Yue, Y., He, B., Li, Y., Wang, W.: Building an efficient put-intensive key-value store with skip-tree. IEEE Transactions on Parallel and Distributed Systems **28**(4), 961–973 (2016)
14. Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., Konka, P.: Triad: Creating synergies between memory, disk and log in log structured key-value stores. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 363–375 (2017)
15. Zhang, T., Wang, J., Cheng, X., Xu, H., Yu, N., Huang, G., Zhang, T., He, D., Li, F., Cao, W., et al.: {FPGA-Accelerated} compactions for {LSM-based}{Key-Value} store. In: 18th USENIX Conference on File and Storage Technologies (FAST 20). pp. 225–237 (2020)
16. Wang, X., Jin, P., Hua, B., Long, H., Huang, W.: Reducing write amplification of lsm-tree with block-grained compaction. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE). pp. 3119–3131. IEEE (2022)
17. Dayan, N., Idreos, S.: 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. pp. 505–520 (2018)
18. Athanassoulis, M., Chen, S., Ailamaki, A., Gibbons, P.B., Stoica, R.: Masm: efficient online updates in data warehouses. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. pp. 865–876 (2011)
19. Amur, H., Andersen, D.G., Kaminsky, M., Schwan, K.: Design of a write-optimized data store. Tech. rep., Georgia Institute of Technology (2013)
20. Yao, T., Wan, J., Huang, P., He, X., Wu, F., Xie, C.: Building efficient key-value stores via a lightweight compaction tree. ACM Transactions on Storage (TOS) **13**(4), 1–28 (2017)
21. Pan, F.F., Yue, Y.L., Xiong, J.: dcompaction: Speeding up compaction of the lsm-tree via delayed compaction. Journal of Computer Science and Technology **32**, 41–54 (2017)
22. Zhang, W., Xu, Y., Li, Y., Li, D.: Improving write performance of lsmt-based key-value store. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). pp. 553–560. IEEE (2016)
23. Mei, F., Cao, Q., Jiang, H., Li, J.: Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 477–489 (2018)
24. Chan, H.H., Li, Y., Lee, P.P., Xu, Y.: Hashkv: Enabling efficient updates in kv storage via hashing. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 1007–1019 (2018)
25. `https://github.com/spdk/spdk`
26. Luo, C., Carey, M.J.: Lsm-based storage techniques: a survey. The VLDB Journal **29**(1), 393–418 (2020)
27. Zhao, X., Jiang, S., Wu, X.: Wipdb: A write-in-place key-value store that mimics bucket sort. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE). pp. 1404–1415. IEEE (2021)
28. `https://github.com/google/leveldb`
29. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)