Cluster Computing (2024) 27:1527–1540
https://doi.org/10.1007/s10586-023-04031-9

InPlaceKV: in-place update scheme for SSD-based KV storage systems under update-intensive Worklaods

Jianing Zhao[1] · Yubiao Pan[1] · Huizhen Zhang[1] · Mingwei Lin[2] · Xin Luo[3] · Zeshui Xu[4]

Received: 28 November 2022 / Revised: 28 April 2023 / Accepted: 13 May 2023 / Published online: 28 May 2023

Abstract
Modern key-value (KV) storage systems adopt append-only writes to update KV pairs with the out-of-place manner, because the performance of sequential accesses is much better than that of random accesses for HDDs. Compaction or GC operations will be deployed by traditional KVs or KV separation schemes due to updating KV pairs via append-only writes. Unfortunately, the system performance will be hurt because extra reads and writes are triggered during those operations, especially under update-intensive workloads.We find that the performance gap for SSDs between sequential and random accesses will get close when the request size becomes large in our experiments. Motivated by this, we propose InPlaceKV built atop SSDs, which adopts an in-place large-update scheme with a hotness-aware method to update KV pairs rather than use append-only writes with the LSM-tree. We further design the working flow of system operations with appropriate data structures. Finally, we compare InPlaceKV with state-of-the-art KV storage systems via extensive experiments under update-intensive workloads, and results validate the effectiveness of our design in improving the system throughput.

Keywords Big Data · KV Storage System · SSDs · Append-only Write · Update In-place · Throughput

1 Introduction

Key-value (KV) storage systems, which store massive data as KV pairs, are an emerging storage engine and widely used to support various big data applications and high performance distributed computing environments, such as LevelDB [1] for Chrome, RocksDB [2] for Facebook, Redis [3] for Twitter, HTAP database [4], graph stores [5, 6], search engine [7], and so on.

In order to fully utilize the performance of sequential writes, modern KV storage systems [1, 2, 8–13] usually adopt the *Log-Structure Merge tree (LSM-tree)* [14] as their fundamental structure. The main idea is to buffer random writes and turn them into a large sequential request by *an append-only write*, while keeping KV pairs fully sorted in this sequential request for efficient query operations. In particular, LSM-tree-based KV storage systems need *compaction* operations to move KV pairs from higher level of LSM tree to lower level and reclaim those invalid KV pairs. Readers can refer to Sect. 2 for the detailed description of LSM-tree-based KV storage systems. Due to tremendous extra I/Os caused by compaction, we emphasize that LSM-tree-based KV storage systems will suffer

✉ Yubiao Pan
panyubiao@hqu.edu.cn

✉ Mingwei Lin
linmwcs@163.com

Jianing Zhao
zhaojianing@stu.hqu.edu.cn

Huizhen Zhang
zhanghz@hqu.edu.cn

Xin Luo
luoxin@swu.edu.cn

Zeshui Xu
xuzeshui@263.com

1 The School of Computer Science and Technology, Huaqiao University, Xiamen 361000, Fujian, China

2 The College of Mathematics and Informatics, Fujian Normal University, Fuzhou 350000, Fujian, China

3 The College of Computer and Information Science, Southwest University, Chongqing 400000, Chongqing, China

4 The Business School, Sichuan University, Chengdu 610000, Sichuan, China

from read and write amplification, thus reducing throughput.

Though relaxing the degree of fully-sorted ordering (e.g., PebblesDB [8] and Dostoevsky [9]) for each level of an LSM-tree can alleviate read and write amplification, it can not completely eliminate the compaction overheads because KV storage systems still need compaction operations to move KV pairs from higher levels to lower levels. *KV separation* (e.g., WiscKey [10], DiffKV [11], HashKV [12] and FenceKV [13]) is a direction to reduce the compaction overheads, whose main idea is to keep keys and metadata information in the LSM-tree while storing values separately in a value log (short for vLog) via append-only writes. Because the size of keys is much smaller than that of values, the compaction overheads caused by the LSM-tree is negligible. Unfortunately, KV separation suffers from severe garbage collection (GC) overheads because it needs extra I/Os to reclaim space occupied by invalid values in the vLog. Additionally, KV separation may need more I/Os when responding query operations due to the segregated storage of keys and values, thus also degrading the system throughput.

In short, append-only writes make LSM-tree-based KVs (KV pairs are fully-sorted or partially-sorted) and KV separations deploy compaction operation and GC operation, respectively, thus causing these two kinds of KV storage systems to face the challenge of throughput degradation, especially when the amount of updated KV pairs becomes tremendous. In fact, update intensive workloads are common in many storage scenarios, such as online transaction processing [15] and enterprise servers [16].

There is an interesting question that whether it is still necessary to adopt append-only writes for KV storage systems. The reason why adopting append-only write is that the performance gap bweteen sequential and random operations is quite large for traditional hard disk drives (HDDs). However, this performance gap for modern SSDs is not as large as that for HDDs [10, 17, 18]. In order to exploit the performance of SSDs, we have conducted IOPS evaluations with an NVMe SSD by varying the request size from 4KB to 512KB. Figure 1 shows the performance comparison between sequential and random accesses with different request size. As shown in this figure, the performance gap between sequential and random access is getting smaller and smaller when the request size ranges from 4KB to 512KB. Furthermore, the IOPS of random write is almost same as that of sequential write when the request size is larger than 128KB.

How to make the best of the large random access performance in SSDs to perform in-place-update, rather than append-only writes, in KV storage systems for less read/write amplification still remains as an interesting research problem. In this paper, we propose a novel KV design, built atop modern SSDs, with hotness-aware in-place updates to overwrite invalid data, called *InPlaceKV*. Based on this design, InPlaceKV adopts a log structure with in-place updates instead of the LSM-tree structure to eliminate the read/write amplification caused by compaction, so as to improve the system throughput. In particular, we make the following contributions in this paper.

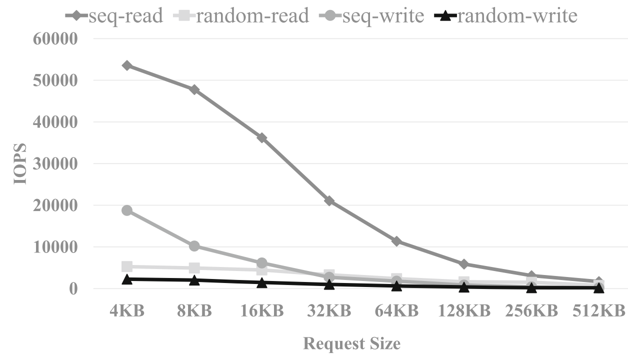


Fig. 1 Performance comparison between sequential and random access with different request size for SSDs

- We first propose an in-place large-update scheme for SSD-based KV storage systems while traditional KVs adopt out-of-place update via append-only writes. The key idea of this scheme is to take advantage of the large random access performance of SSDs to update KV pairs with a large unit in place rather than use append-only writes with the LSM-tree. Therefore, SSD-based KV storage systems with this scheme will alleviate the read and write amplification caused by compaction or GC compared to those KV storage systems using append-only writes to update KV pairs in an out-of-place manner.
- In order to trigger in-place large-updates, we adopt a hotness-aware method to group KV pairs according to their hotness. Specifically, we first calculate the hotness of each KV pair as the number of times that a KV pair has been accessed. Then, KV pairs are sorted by hotness, and those KV pairs with similar hotness are grouped and stored in the same data block. There is a high probability to update together for those KV pairs with similar hotness. Thus, we can conduct the in-place update in a large unit because the cost of large random access is similar to that of large sequential access in modern SSDs. Additionally, grouping KV pairs with similar hotness will help prolong the lifetime of SSDs.
- Based on the design above, we deploy the working flow of system operations with appropriate data structures in main memory and SSDs. Finally, we provide the system robustness and implement the InPlaceKV prototype on LevelDB. We validate the effectiveness of InPlaceKV

in improving system throughput with update-intensive workloads. Results show that compared to LevelDB, RocksDB and DiffKV, InPlaceKV achieves $1.2-14.6\times$ throughput over those KV storage systems under generated and YCSB core workloads.

The rest of this paper is organized as follows. Section 2 introduces the necessary background on LSM-tree-based KV storage systems, and gives the motivation of our design. Section 3 describes the detailed design of our InPlaceKV. Sect. 4 presents an in-depth evaluation to validate the effectiveness of our design. Section 5 presents related work and Sect. 6 concludes the paper and shows the future work.

2 Background

In this section, we first review the background on LSM-tree-based KV storage systems, then discuss the overheads caused by append-only writes, and finally motivate the design of our InPlaceKV.

2.1 Background on LSM-tree-based KV Systems

We use LevelDB [1] as a representative example to introduce the background of LSM-tree-based KV systems. Figure 2 shows the architecture of LevelDB. Similar to most KV storage systems, LevelDB supports read, write, update, delete and range scan operations. Among them, write, update and delete operations are implemented using append-only writes. When a key-value pair is written, it is appended to a MemTable, an in-memory data structure. When the size of the MemTable reaches its limitation, it is converted to an Immutable MemTable, with all key-value pairs sorted. The Immutable MemTable is read-only and flushed to an external storage device (e.g., SSD). The same process is applied to update operations which use a global sequence number to ensure that the latest version of KV pairs can be requested. Differently, delete operations will adopt an append write to add a deletion mark for a certain KV pair. For read operations, LevelDB firstly searches the MemTable and Immutable MemTable, then looks up from Level 0, and goes through each level until the target KV pair is found or the lowest level is reached.

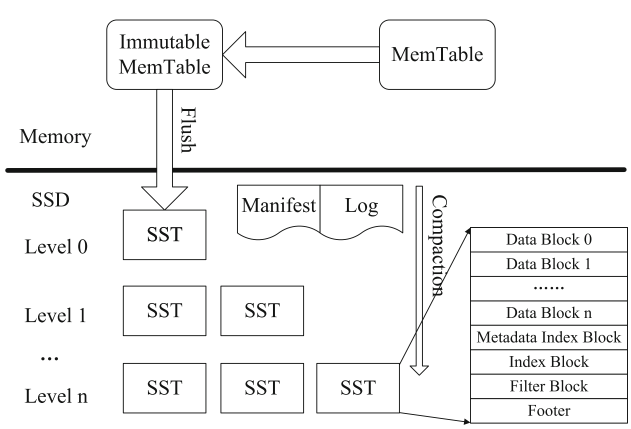


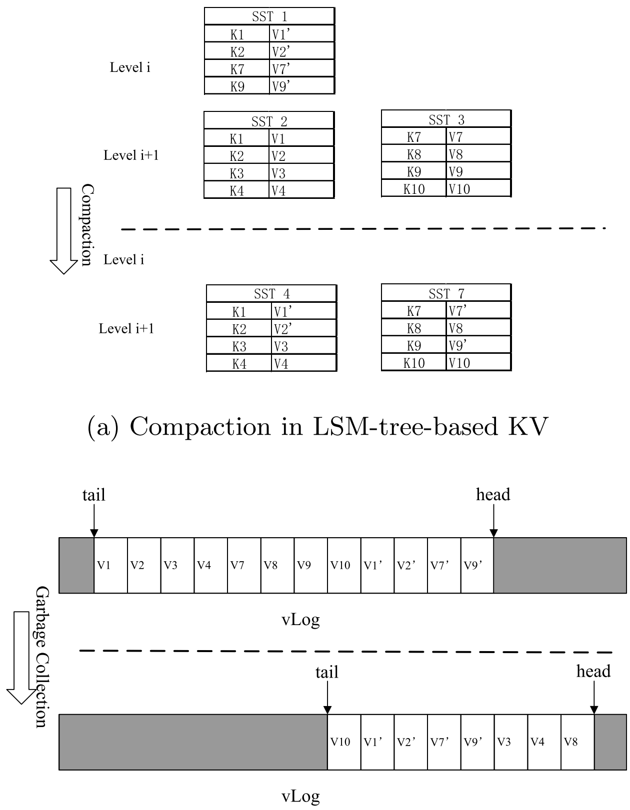
Fig. 2 The architecture of LevelDB

When KV pairs are flushed to the external storage device, they are formed as an SST file and written in Level 0 of the LSM-tree. The structure of an SST file is shown in Fig. 2. All KV pairs in one SST are sorted and stored in *Data Blocks*, each with a typical size of 4KB. Subsequently, KV pairs are merged and moved to a lower level via compaction when the number of SST files in Level 0 reaches a certain limitation. Specifically, one compaction operation firstly reads several SST files from Level 0 and all the overlapped SST files from Level 1; Then, all valid KV pairs are sorted by keys and formed as new SST files while those invalid KV pairs (e.g., deleted or old version KV pairs) are discarded; Finally, those new SST files are flushed into Level 1. Similarly, if Level i (where i ≥ 1) is full, KV pairs are merged and moved to Level i+1 via reading one SST file from level i and all the overlapped SST files from Level i+1. Note that all KV pairs are globally sorted by keys within each level, except Level 0, in which KV pairs are only sorted in each SST file. Compaction maintains the hierarchy of the LSM-tree and allows to reclaim the storage space occupied by invalid or old version KV pairs.

2.2 Overheads caused by append-only writes

As we stated in Sect. 1, both LSM-tree-based KVs and KV separations suffer throughput degradation caused by read/write amplification due to append-only writes. Figure 3 illustrates the read/write amplification problem of LSM-tree-based KVs and KV separations. In this example, we suppose that KV pairs, (K1,V1), (K2,V2), (K3,V3), (K4,V4), (K7,V7), (K8,V8), (K9,V9), (K10,V10), (K1,V1'), (K2,V2'), (K7,V7'), and (K9,V9'), are sequentially written into a KV storage system. Among them, (K1,V1'), (K2,V2'), (K7,V7'), and (K9,V9') change V1 to V1', V2 to V2', V7 to V7', and V9 to V9', respectively.

Due to append-only writes, LSM-tree-based KVs need compaction to reclaim the space occupied by those invalid or old version KV pairs. In Fig. 3(a), we assume that each SST file contains 4 KV pairs. (K1,V1), (K2,V2), (K3,V3) and (K4,V4) are stored in SST 2, (K7,V7), (K8,V8), (K9,V9) and (K10,V10) are stored in SST 3, and (K1,V1'), (K2,V2'), (K7,V7'), and (K9,V9') are stored in SST 1. SST 2 and SST 3 are in Level i+1, while SST 1 is in Level i. During compaction, the system needs to read SST 1, SST 2 and SST 3 into memory, discard old version values V1, V2,



(a) Compaction in LSM-tree-based KV

(b) Garbage Collection in KV Separation

Fig. 3 Overheads caused by append writes

V7 and V9, sort all valid KV pairs, regenerate new SST files, flush SST 4 and SST 7 into Level i+1, and delete SST 1, SST2 and SST 3. In this example, additional 12 KV-pair reads and 8 KV-pair writes are introduced.

As we described before, KV separations keep keys and metadata information in the LSM-tree while storing values separately in a vLog via append-only writes. In a system with KV separation, compaction is needed to move keys from higher levels to lower levels in the LSM-tree, but the overheads is small. For vLog, the system still requires additional operations, called GC, to reclaim those invalid KV pairs. Figure 3(b) shows an example of GC triggered on vLog. There are two pointers (*head and tail*) used for vLog. New values are appended in the head pointer, while the tail pointer tells us where to start freeing space when GC is triggered. When triggering one GC, the system reads several KV pairs from vLog, and checks whether those KV pair are valid or not by querying keys in the LSM-tree. Finally, valid KV pairs are appended back to the head of vLog. For ease of illustration, we only show values in vLog for Fig. 3(b). In this example, additional 7 KV-pair reads and 3 KV-pair writes are needed during this GC operation. Furthermore, some query and compaction overheads are introduced for the LSM-tree.

2.3 Motivation

Append-only writes make traditional KV stores with LSM-tree and KV separations generate compaction operations and GC operations, respectively, thus leading them to face the challenge of throughput degradation, especially when the amount of updated KV pairs becomes tremendous.

The reason why adopting append-only writes in KV storage systems is that the performance gap between sequential access and random access is quite large for HDDs. The throughput of sequential access can reach several hundred MB per second for HDDs, while that of random access is several MB per second. However, SSDs, receiving lots of attention in research [19–22], are gradually replacing HDDs in storage systems and its performance gap between sequential access and random access is not as large as that of HDDs. Furthermore, the result of our evaluations shown in Fig. 1 indicates that the performance of sequential and random accesses are almost the same when the request size is large.

From the above discussions, we find that how to make the best of the SSD's performance of large random access to perform in-place updates rather than updating with append-only writes in KV storage systems is an interesting problem. On the one hand, in-place updates via large random access will only lead to little performance degradation compared to out-of-place update via append-only writes according to the result shown in Fig. 1. On the other hand, discarding append-only writes will improve system performance because overheads caused by compaction or GC can be avoided. In this paper, we address the problem above by developing InPlaceKV, which performs in-place updates with a hotness-aware scheme via large random accesses, and we present the design details in the next section.

3 Design

In this section, we first state our design objectives. Then, we present the architecture, data structure and working flow of InPlaceKV. Finally, we discuss several issues in practical implementation.

3.1 Design objectives

InPlaceKV mainly aims for improving system performance via performing in-place updates with large random accesses. Thus, it focuses on the following three design objectives.

- *Eliminating write/read amplification caused by append-only writes.* As we discussed above, append-only writes

in KV storage systems will lead to write/read amplification, thus degrading system performance. Therefore, our first objective is to use in-place updates with large random accesses to replace append-only writes in KV storage system which is built atop modern SSDs.
- *Proposing appropriate design for better performance.* A KV storage system adopting in-place updates with large random accesses is quite different from traditional one. Therefore, we must propose appropriate data structures and working flows for better performance.
- *Improving system robustness.* Though we propose to use in-place updates with large random accesses, there are still a few KV pairs staying in valid state within a large unit of in-place update. Because it is not possible that all KV pairs are invalid when each large unit of in-place update is created. Thus, we must design carefully to improve system robustness.

3.2 InPlaceKV overview

The main idea of our InPlaceKV is to update KV pairs in place with a large unit rather than using append-only writes. To reduce the overheads of in-place large updates, KV pairs, via a hotness-aware method, are grouped into a large unit (e.g., 128KB) which is called *Data Block*. Precisely, KV pairs are sorted by hotness, and those with similar hotness are stored in the same Data Block. Figure 4 shows our InPlaceKV architecture. The main data structures and working flows in InPlaceKV will be introduced in the next two subsections.

3.3 Data structure

3.3.1 In-memory data structures

Write_MemTable and Update_MemTable play a role as cache for newly written and updated KV pairs, respectively. In these two structures, a skiplist [23] is used to record inserted KV pairs. When the size of Write_MemTable or Update_MemTable reaches a predefined threshold (e.g., 4MB in a default configuration), the KV pairs within Write_MemTable or Update_MemTable will be sorted according to their hotness with the help of a Frequency_Table and flushed to the SSD.

A Frequency_Table will be created as soon as a new Write_MemTable or Update_MemTable is generated, and deleted after the corresponding KV pairs are stored on the SSD. In our hotness-aware design, the hotness of a KV pair is easily defined as the number of times that the KV pair has been accessed. Therefore, Frequency_Table records the access number for each KV pairs in a certain Write_MemTable or Update_MemTable.

A globally ordered *B-tree* [24] in memory is used as the index. Note that the address of each KV pair is stored in the B-tree after KV pairs are flushed to the SSD, so that a certain Data Block storing the target KV pair can be read directly. Also, range query operations can be performed via this B-tree.

FreeList records the space occupied by a certain KV pair which has been deleted or its new version data has been grouped with other KV pairs and overwritten to another Data Block due to change of its hotness. For example, we suppose that KV pairs A, B, C and D are stored in Data Block 1 at first. Then, the new version of KV pair A is grouped with E, F and G, and written to Data Block 2. Thus, A' old version address is recorded in FreeList, which means that those space in FreeList can be reused in the near future.

3.3.2 KVLog for external storage

Different from LSM-tree-based KVs, InPlaceKV adopts *KVLog* to store KV pairs. Each KVLog consists of several Data Blocks. KV pairs are stored in Data Blocks in a fully sorted order according their hotness as we described above. The format of Data Block is setting as $(ks, k, vs, v)_1, (ks, k, vs, v)_2, \ldots, (ks, k, vs, v)_n$, where k and v represent the key and value, ks and vs mean the size of the key and value, respectively.

3.4 Working flows

In this subsection, we present the working flows of system operations in InPlaceKV, including write, update, read, scan and delete.

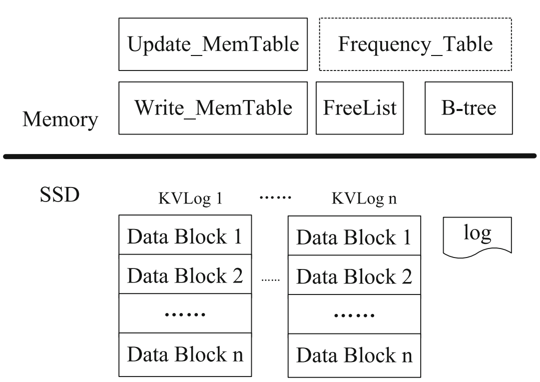


Fig. 4 Architecture of InPlaceKV

3.4.1 Write operation

When handling a write operation for one new KV pair, InPlaceKV will first check B-tree in memory via the corresponding key. B-tree will return NULL and inform InPlaceKV to buffer this new KV pair in the Write_MemTable. At the same time, the corresponding Frequency_Table will be updated. If the Write_MemTable is full, all KV pairs in this Write_MemTable will be merged because there may be several versions of one KV pairs (Please refer to Update Operation), then sorted according to their hotness recorded in the Frequency_Table, and finally flushed to the SSD as a KVLog.

Algorithm 1 Flush-from-Update_MemTable

```
Require: Update_MemTable
Ensure: SUCCESS or FALSE
Update_MemTable → Frequency_Table;
Frequency_Table → Iterator;
while Iterator.vaild() do
    if LDB.size() >= Update_Unit then
        while LDB.hasNext() do
            (K,V) ← LDB.curren();
            (K,V) Divided into
                districts by address;
            LDB.next();
        end while
        Update_Rate =
                district.max_Update_Rate;
        if Update_Rate == 1 then
            Update in-place for one PDB;
        else if Update_Rate > TH then
            Valid KV pairs migration;
            Part of old addresses → FreeList;
            Update in-place for one PDB;
        else
            All old addresses → FreeList;
            Update in an empty PDB;
        end if
    end if
    Iterator.next();
end while
Modify B-tree;
```

3.4.2 Update operation

When handling an update operation for one KV pair, InPlaceKV will also check the B-tree first. If the B-tree returns a key without its corresponding address, it means an old version of this KV pair has been written and buffered in the Write_MemTable. Otherwise, an old version of this KV pair has been flushed to the SSD. For the former situation, InPlaceKV buffers this updated KV pair in the Write_MemTable as handling a new write operation. For the latter one, this updated KV pair is buffered in the Update_MemTable. If the Update_MemTable is full, KV pairs will be merged, sorted, and flushed to the SSD according to the Flush-from-Update_MemTable flow shown in Algorithm 1, which presents the in-place large-update scheme for InPlaceKV.

In particular, an iterator is first created for all hotness-sorted KV pairs. Those KV pairs are divided into several logical Data_Blocks (abbreviated for LDB) which will be flushed as a large unit. When the size of a logical Data_Block reaches the Update_Unit which is a configurable threshold (typically 4KB in our evaluations), we will calculate how many updated KV pairs belong to one certain physical Data_Block (short for PDB) stored on the SSD. After that, InPlaceKV will choose a PDB in which there are the most updated KV pairs, and get the Update_Rate which is defined as $\frac{N^{PDB}_{KVs}}{N_{KVs}}$, where N^{PDB}_{KVs} means the number of updated KV pairs whose old versions belong to one certain PDB and N_{KVs} is the total number of updated KV pairs in this LDB. For different Update_Rates, InPlaceKV will perform different operations:

- If the Update_Rate is equal to 1, which means all KV pairs in the PDB are updated simultaneously, InPlaceKV will trigger an in-place large-update for this PDB.
- If the Update_Rate is larger than TH (short for threshold, e.g., 0.8 as we configure) and less than 1, which indicates that most of KV pairs in the PDB needed to be updated in concurrently, InPlaceKV will migrate those valid KV pairs into a Write_MemTable, add old addresses of KV pairs whose old versions do not belong to the PDB to FreeList, and perform an in-place large-update for this PDB.
- Otherwise, InPlaceKV will flush KV pairs into a new empty PDB and add all old addresses to FreeList. Note that, a new empty PDB can be allocated from free space of InPlaceKV or from one PDB whose proportion of invalid KV pairs recorded in FreeList is larger than TH (e.g., 0.8 as we configure). For the latter situation, InPlaceKV will trigger a few migrations.

Finally, the B-tree must be modified after KV pairs in a LDB are flushed into one PDB.

3.4.3 Read operation

When a query request for a KV pair comes, InPlaceKV will first check the Write_MemTable and Update_MemTable. If this KV pair does not exist in the Write_MemTable and Update_MemTable, InPlaceKV will search the B-tree to confirm whether the KV pair is stored in a certain Data_Block or not. As we described above, the address of each pair is stored in the B-tree. Note that the structure of one leaf node to record the address of one KV pair is (KVLog#, DB_Size, DB_Offset, Start_Point), where KVLog# is the number of KVLog, DB_Size means the size of a certain Data_Block in which the KV pair is stored, DB_Offset is the offset of the Data_Block, and Start_Point is the offset of the KV pair in this Data_Block. It should be noted that the DB_Size here is not equal to the update unit size set by the system, but a specific value, which may be slightly larger or smaller than the update unit. This setting is used to handle the cases of different DB_Sizes due to variable length key-value pairs. Therefore, InPlaceKV will directly get the address of the required KV pair from the B-tree or be informed from the B-tree that this KV pair does not exist in this system. Finally, InPlaceKV will read a certain PDB, and send the result to users.

3.4.4 Scan and delete

For a scan operation, InPlaceKV first checks the B-tree to obtain leaf nodes in the range of this operation. Then, InPlaceKV will read KV pairs from the SSD if the obtained leaf nodes contain keys and addresses, simultaneously; Otherwise, InPlaceKV will read KV pairs from the Write_MemTable and Update_MemTable.

For a delete operation, InPlaceKV will add the address of this KV pair to FreeList, and modify the B-tree to logically delete the KV pair. Note that the space occupied by the content of this KV pair will be overwritten during an update operation in the near future.

3.5 Implementation issues

In this subsection, we present two implementation issues of our InPlaceKV design. At first, we discuss the system robustness, and then analyze the storage overheads of InPlaceKV.

3.5.1 System robustness

Even though we adopt a hotness-aware method to group KV pairs to trigger in-place large-updates, there still exist a few valid KV pairs in a chosen PDB. For example, KV pairs A, B, C and D are stored in PDB 1, E, F, G and H are stored in PDB 2, respectively. We suppose that A, B and C are updated to A', B' and C', and are grouped with an updated E' to perform an in-place update in PDB 1. There are three kinds of KV pairs needed to be processed. (1) KV pair D, which is still a valid KV pair in PDB, must be migrated to memory and its index must be modified in the B-tree. (2) KV pairs A', B' and C' are directly flushed in place. (3) KV pair E', whose old version is stored in PDB 2, is flushed with A', B' and C' to overwrite PDB 1; Then, the address of E must be added to FreeList; Finally, its index must be modified in the B-tree. After updating PDB 1, we support that KV pair F', G', H' and I will be updated for PDB 2. Therefore, InPlaceKV will find that KV pair E stored in PDB 2 is invalid by checking FreeList. In our implementation, KV pairs in an Update_MemTable can be grouped into several LDBs. To maintain data consistency and reduce the costs of checking FreeList, a temporary variable smallFreeList is set during the process of turning those LDBs to corresponding PDBs, instead of directly operating on FreeList. After one Update_MemTable is fully flushed, the final smallFreeList is then merged into FreeList.

3.5.2 Storage overheads

As we described in Fig. 4, the Frequency_Table, FreeList and B-tree are additional data structures compared to traditional KV storage systems. One Frequency_Table is created for a certain Write_MemTable or Update_MemTable, and will be deleted when one Write_MemTable or Update_MemTable has been flushed to the SSD. FreeList records those invalid addresses for update and delete operations, and an address will be removed from FreeList when its physical space is used to be overwritten. Therefore, the Frequency_Table and FreeList cost a few space in main memory. As we present in Read Operation, the structure of one leaf node in B-tree is (KVLog#, DB_Size, DB_Offset, Start_Point). In our configuration, DB_Size uses 3 Bytes (24 bits) to represent the size of a certain Data_Block, which means the largest Data_Block we can configure will reach $2^{24}B = 16MB$. Besides, we also use 3 Bytes to represent DB_Offset, as well as Start_Point and KVLog#. Therefore, one leaf node in the B-tree as least consumes 20B in main memory. If the average size of one KV pair is 1KB, the storage overheads are less than 2% of the size of total KVLogs. If the size of KV pair increases, the overheads will be reduced. Additionally, we can flush some cold indexes from the B-tree to the SSD to further reduce the amount of memory consumption.

4 Evaluation

In this section, we evaluate and compare InPlaceKV with LevelDB [1], RocksDB [2] and DiffKV [11]. LevelDB and RocksDB are two well-known traditional KV storage systems, while DiffKV is the state-of-the-art KV separation scheme.

4.1 Setup

4.1.1 Testbed

Our experiments are conducted on an HP Z2 Tower G4 Workstation with an Inter(R) Core(TM) 3.20 GHz processor, 8GB RAM, and a 512 G Intel 660P series SSD. The machine runs Ubuntu 20.04 LTS with the Linux 5.11 kernel and ext4 file system.

4.1.2 Workload

The system performance was tested mainly using YCSB [25] to generate workloads and YCSB core workloads. We compare the system performance under update-intensive workloads, thus our generated workloads include five stages with a fixed KV size of 1KB. At first, 10 million KV pairs are loaded randomly (denoted as Load stage). Then, 10 million read requests come into the systems (Read-1 stage). After that, we trigger two 10-million updates successively (Update-1 and Update-2, respectively). Finally, 10 million reads are issued again (Read-2 stage). Each stage in the above workloads are generated by YCSB with Zipfian distribution, except the Load stage. Furthermore, we also consider the YCSB core workloads, referred to Table 1, to validate the effectiveness of InPlaceKV in improving the system performance.

4.1.3 System configuration

For each KV storage system in our experiments, we set MemTable size as 4MB, which is the same size as that of the Write_MemTable and Update_MemTable in InPlaceKV. It should be noted that the InPlaceKV consumes 8MB because it has two caches for updates and writes respectively. All systems are configured with a table cache (1000 in a default configuration), and none has Bloom filters turned on. We configure a large block cache of 500MB for LevelDB, RocksDB and DiffKV, while we run InPlaceKV without block cache for fair comparison. The reason is that the B-tree in InPlaceKV will consume additional memory which is different to other three KV storage systems.

Table 1 YCSB core workloads

Workload	Features
Load	100% random inserts
A	50% reads and 50% updates
B	95% reads and 5% updates
C	100% reads
D	95% reads and 5% writes
F	50% reads and 50% read-modify-write

4.2 Performance comparison

In this subsection, we compare the throughput of different KV storage systems under our generated workloads. Figure 5 shows that InPlaceKV achieves a higher throughput than LevelDB and RocksDB for each stage. Compared with DiffKV, the throughput of InPlaceKV is better in the Load, Update-1 and Update-2 stages.

Specifically, InPlaceKV achieves 5.1X, 4.9X and 4.8X throughput in Load stage than LevelDB, RocksDB and DiffKV, respectively. For Update-1 and Update-2 stages, InPlaceKV achieves 1.2-1.4X throughput compared to other three KV storage systems. The results validate that InPlaceKV indeed improves the throughput under update-intensive workloads because it adopts the in-place update scheme rather than append-only writes, thus eliminating the costs caused by compaction in traditional LSM-tree KVs or GC in KV separations. For Read-1 and Read-2 stages, DiffKV and InPlaceKV achieve higher throughput than LevelDB and RocksDB because InPlaceKV has a globally indexed B-tree, while DiffKV applies a key-value separation design. However, DiffKV manages values with partially-sorted ordering while InPlaceKV just groups values according to their hotness. Therefore, DiffKV achieves better spatial locality, thus leading to better read performance compared to InPlaceKV.

4.3 Throughput analysis

In this subsection, we analyze the throughput of InPlaceKV, LevelDB and RocksDB for each 0.2 million requests during 5 stages in generated workloads. Each subfigure in Fig. 6 shows the result for each stage.

Figure 6(a) shows that the throughput of InPlaceKV in Load stage is significantly better than others. Because the compaction operation has a very obvious impact on the throughput of LevelDB and RocksDB, while InPlaceKV

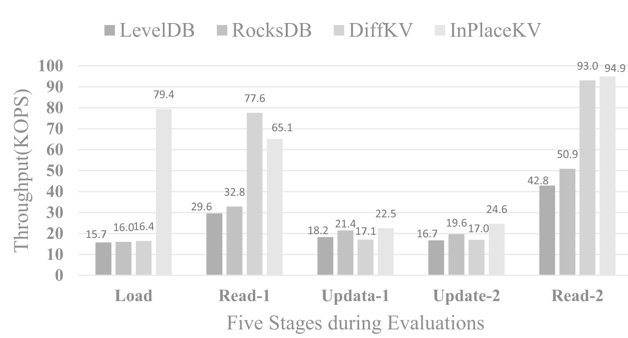


Fig. 5 Throughput of generated workloads

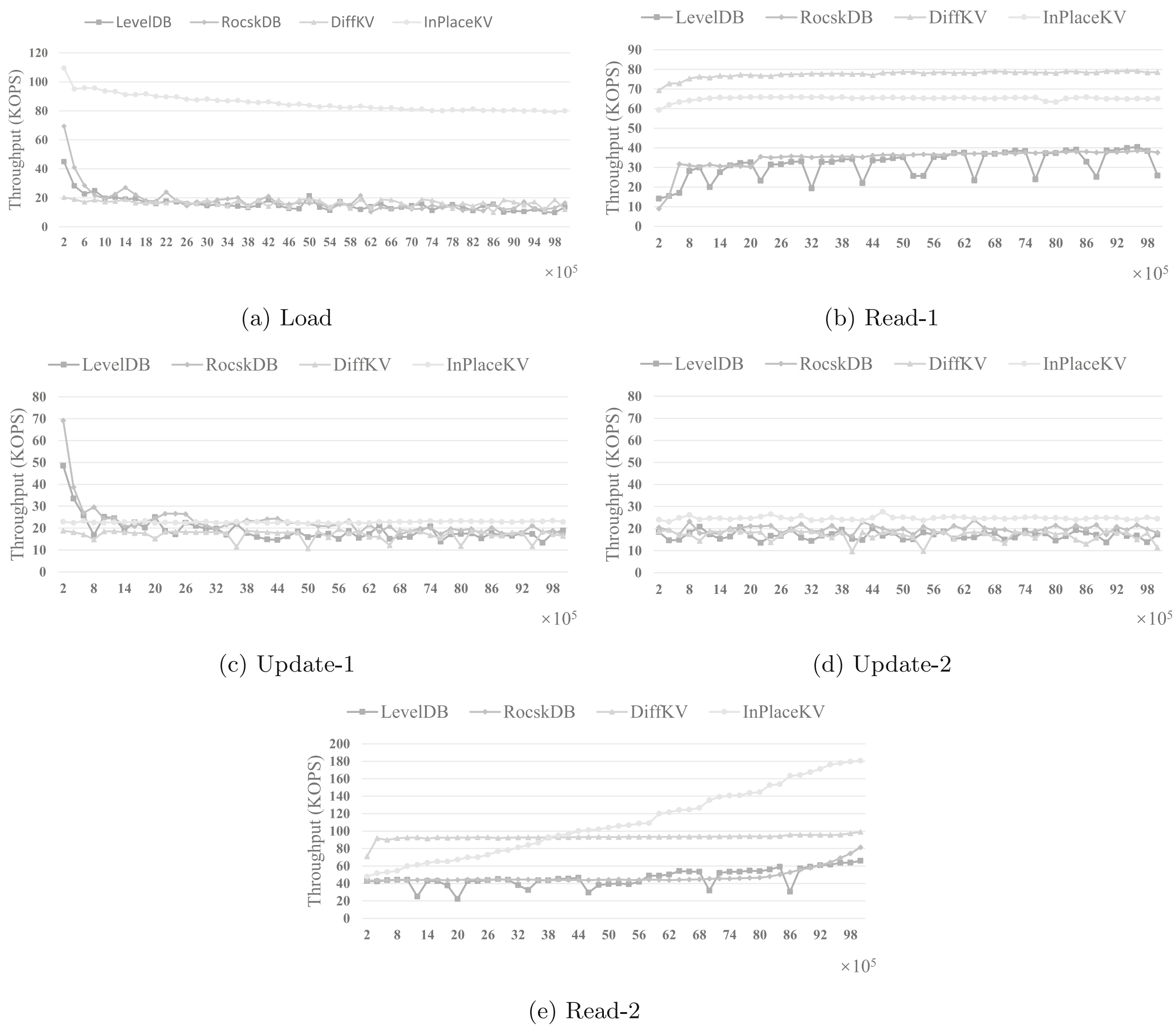


Fig. 6 Trends of throughput

always writes all new KV pairs to the SSD using appending. Though the throughput of InPlaceKV will decrease as the scale of B-tree increase, the overall throughput is still much better than LevelDB and RocksDB.

InPlaceKV achieves a better throughput in two read stages in Fig. 6(b, e). The main reason is that InPlaceKV uses the address in the B-tree to read the target KV pair directly with a small read amplification, while LevelDB and RocksDB will cost more I/Os to respond a read request. For Read-2 stage shown in Fig. 6(e) after two update stages, the throughput of all KVs will get some improvements, because updates will gather hot KV pairs in the high level of LSM-tree for LevelDB and RocksDB, and group KV pairs together with similar hotness in a PDB for InPlaceKV. In addition, DiffKV has a superior read performance with the application of key-value separation technique, which has performance advantage for key lookup of a smaller LSM tree. However, For Read-2 stage shown in Fig. 6(e), it can be found that the read performance of InPlaceKV is improved with the help of hot aggregation and the system cache. The main reason is that those data, in InPlaceKV, with the similar hotness are grouped and stored in the same data block, thus leading to a more efficient cache.

For Update-1 stage, Fig. 6(c) shows that InPlaceKV achieves a little throughput improvement even though it eliminates the overheads caused by append-only writes via the in-place large-update scheme. The reason is that update process of InPlaceKV is more complex, including B-tree insertion, FreeList checking, valid data migration, B-tree updating, and so on. In the consecutive update, Update-2 stage, Fig. 6 shows that InPlaceKV performs better. This is because the hotness-aware scheme works well to reduce the overheads of update process in InPlaceKV after execution of Update-1 stage. For example, much more KV pairs in one PDB will be updated locally in Update-2 stage.

4.4 Tunable parameter

We further study the impact of block size on the write/update performance of InPlaceKV. In this subsection, we vary the block size from 4KB to 256KB, and show the throughput of InPlaceKV for Load stage, Update-1 stage, and Update-2 stage in Fig. 7.

As we analyze and state in Sect. 1, the random performance of SSD gradually increases as the request size increases. With the increase of the update unit, block size, the KOPS in Load, Update-1 and Update-2 stages, increases gradually. For Load stage, because InPlaceKV write new KV pairs via append writes, so block size does not affect on the performance. For update stages, with the increase of block size, InPlaceKV can obtain fewer random writes and better random performance.

4.5 YCSB evaluation

The YCSB benchmark is widely used to evaluate NoSQL databases with six different load settings. Among them, Load workload contains 100% random inserts; A includes 50% reads and 50% updates; B consists of 95% reads and 5% updates; C contains 100% reads; D contains 95% reads and 5% writes; F includes 50% reads and 50% read-modify-writes. We run LevelDB, RocksDB, DiffKV and InPlaceKV under those workloads, and experimental results are shown in Fig. 8.

For Load, the throughput of InPlaceKV is much higher than other KV storage systems, e.g., 5.9X, 5.4X, and 3.7X over LevelDB, RocksDB and DiffKV. For B, C and D, which are read-intensive workloads, InPlaceKV outperforms 2-2.5X over LevelDB and RocksDB, while DiffKV is better than InPlaceKV. As we mentioned above, the read performance of DiffKV is better than that of InPlaceKV, so DiffKV shows better throughput than InPlaceKV in read-

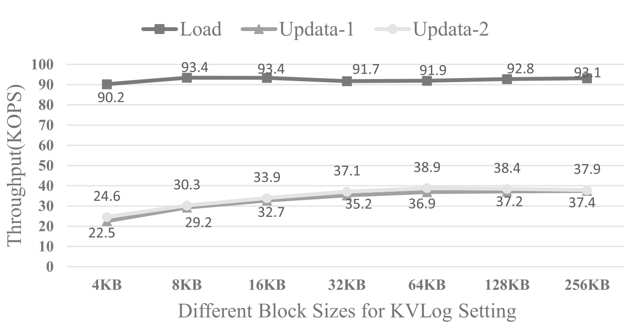


Fig. 7 Throughput under different block sizes

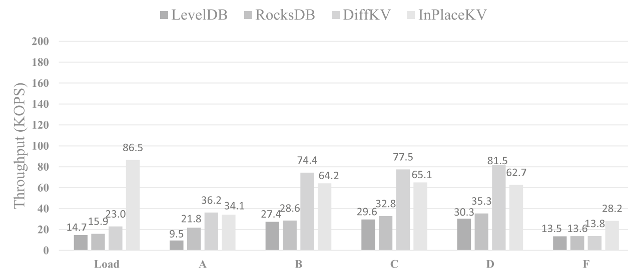


Fig. 8 Throughput under YCSB benchmarks

intensive workloads. For A and F with balanced read and update operations, InPlaceKV outperforms 1.5-2.5X over LevelDB, RocksDB. And due to the existence of garbage collection mechanism in DiffKV, the performance of DiffKV decreases in the case of an increased proportion of update and write operations.

The experimental results show that InPlaceKV performs better than LevelDB and RocksDB in read-intensive workloads, because the indexes in the B-tree are used to locate directly into the target data block, making the read operations more efficient. However, DiffKV is better than InPlaceKV under above workloads. InPlaceKV performs better in workloads where write/update operations are more intensive. The reason for this situation is that intensive appending of KV pair will cause other three KVs to frequently trigger compaction or GC that affects system performance, while InPlaceKV can eliminate the overheads caused by append-only writes. Furthermore, the update performance for InPlaceKV is lower compared to the write performance because the update process in InPlaceKV is more complex.

5 Related work

Many research works focus on optimizing the write performance for KV storage systems. LevelDB [1] and RocksDB [2] adopt the traditional LSM-tree via append-only writes and compaction operations to make KV pairs fully-sorted in each level of LSM-tree. To reduce the overheads caused by compactions, several works [8, 9, 26–29] try to relax the degree of fully-sorted ordering. PebblesDB [8] proposes a Fragmented Log-Structure Tree, which divides KV pairs into several segments for each level, allows KV pairs in each segment to keep unsorted, and ensures segments are not overlapped in each level. Dostoevsky [9] proposes a lazy leveling scheme by adopting tiering policy for all levels of LSM-tree except the lowest level which is implemented leveling policy. The idea of partially-sorted for the LSM-tree can also be found in SlimDB [26], dCompaction [27], VT-tree [28], and SifrDB [29]. Differently, InPlaceKV bulit atop on the SSD proposes an in-place large-update scheme instead of updating via the LSM-tree for eliminating the overheads caused by append-only writes, while maintaining similar performance compared to sequential writes.

KV separation is a direction to reduce the compaction overheads. WiscKey [10] is the first work of proposing the idea which is to store values in the vLog via append-only writes while keeping keys and metadata in the LSM-tree. DiffKV [11] follows the idea of KV separation and introduces a vTree for maintaining the values with partially-sorted ordering. The ideas of KV separation are also found in HashKV [12] and FenceKV [13]. As we discussed in Sect. 2, KV separations will suffer overheads caused by append-only writes because GC needs to be trigger to reclaim the space occupied by those invalid values. Because InPlaceKV updates KV pairs in an in-place manner with large units rather than append-only writes, it will eliminate the costs due to additional GC operations.

Besides, many works [30–37] focus on improving the read and scan performance. bLSM [30] is the first work to use Bloom Filter to boost read operations via eliminating unnecessary read operations. Monkey [31] adopts differentiated Bloom Filters in different levels in the LSM-tree, and ElasticBF [32] proposes a fine-grained elastic Bloom Filter method to improve read performance. Learning models have also been applied to the optimal use of Bloom Filters, such as L-FBF [33]. AC-Key [34] designs an adaptive cache mechanism to accelerate read performance. TridentKV [35] designs an adaptive learning index structure to speed up file indexing to improve read performance. Rosetta [36] introduces a probabilistic range filter to bring benefits for range queries without hurting point queries, and REMIX [37] designs a compact multi-table index data structure for fast range queries in LSM-trees. Compared to those works, InPlaceKV focus on eliminating the overheads caused by append-only writes.

Unlike those works on performance improvement above via data structure optimization, many researchers build their KV storage systems on emerging hardware, such as persistent memory (PM). The work [38] optimizes the system performance via PMs. SLM-DB [39] designs a Single-Level Merge DB which takes advantage of both the B+-tree index and the LSM-tree by leveraging the fast PM. FlatStore [40] is a PM-based KV storage system which combines a persistent log structure and a volatile index for efficient storing and fast indexing, respectively. HiLSM [41] proposes a hybrid KV storage system with non-volatile memory and SSD to provides a cost-efficient solution for WAL synchronization, while SpanDB [42] uses SSDs and a NVMe SSD to build a KV storage system which provides high-speed parallel WAL. The idea of making use of emerging hardware to build KV storage systems can also be found in GearDB [43], SplinterDB [44], FlashKey [45], LogStore [46]. Another research works, e.g., PLSC-tree [47] and KVSSD [48], design a friendly key-value management for SSDs. Differently, InPlaceKV is just built atop SSDs and make the best use of the performance of random access with a large unit.

Actually, there are some works [17, 18] discussing the issue of the performance gap of random accesses and sequential accesses on SSDs and designed KV stores with in-place update. However, their schemes of data aggregation for in-place update are different from InPlaceKV. KVell [17] groups KV pairs with similar size, and stores them in the same file according to writing order, while TreeLine [18] captures the read benefits of a classical disk-bsed B+ tree and writes KV pairs by grouping logically adjacent data together. Differently, InPlaceKV stores KV pairs with similar hotness in the same data block in order to make full use of the performance of large random access in SSDs.

6 Conclusion and future work

Append-only writes make KV storage systems generate compaction or GC operations during which the system performance will be degraded due to extra reads and writes, especially under update-intensive workloads. Our experiments indicate that the SSD's performance gap of sequential and random accesses gets close when the request size is large. Motivated by this, we propose InPlaceKV, which is built atop SSDs, to improve system performance via an in-place large-update scheme with a hotness-aware method to update KV pairs. Its novelty lies in leveraging the performance of random access with a large unit in SSDs so as to group and flush updated KV pairs with an in-place manner rather than using append-only write with the LSM-tree. Our evaluations validate the effectiveness of InPlaceKV in improving the system throughput with update-intensive workloads.

In future work, we will limit the size of B-tree via storing partial nodes of B-tree in the persistent memory with hot-cold separation to reduce the memory consumption and ensure data consistency when system crush happens. Furthermore, we plan to implement Parallel Scan with multi-threading for InPlaceKV via making use of the large degree of internal parallelism in SSDs. All those efforts will further improve the performance of our InPlaceKV.

Author contributions Conceptualization: JZ, YP, HZ; Methodology: JZ, YP, HZ; Investigation: JZ, YP, HZ; Formal analysis: JZ; Software: JZ; Validation: JZ; Writing - original draft: JZ; Supervision: YP, ML; Writing - review and editing: YP, ML, XL, ZX; Resources: ML, XL, ZX.

Funding This research work was supported by the National Natural Science Foundation of China under Grants No. 61802133 and No. 61872086, the Young Top Talent of Young Eagle Program of Fujian Province, China under Grant No. F21E0011202B01, the Natural Science Foundation of Fujian Province under Grants No. 2021J01319 and No. 2022J06020, and the Fundamental Research Founds for the Central Universities of Huaqiao University under Grant No. ZQN-910.

Data availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Chemawat S, Dean J.:Leveldb. https://github.com/google/leveldb, (2022)
2. Facebook. Rocksdb. http://rocksdb.org/, (2022)
3. Redis, Sanfilippo S.: https://redis.io, (2022)
4. Huang, Dongxu, Liu, Qi., Cui, Qiu, Fang, Zhuhe, Ma, Xiaoyu, Fei, Xu., Shen, Li., Tang, Liu, Zhou, Yuxing, Huang, Menglong, Wei, Wan, Liu, Cong, Zhang, Jian, Li, Jianjun, Xuelian, Wu., Song, Lingyu, Sun, Ruoxi, Shuaipeng, Yu., Zhao, Lei, Cameron, Nicholas, Pei, Liquan, Tang, Xin: Tidb: a raft-based htap database. Proc. VLDB Endow **13**(12), 3072–3084 (2020)
5. Elyasi, N., Choi, C., Sivasubramaniam, A.: Large-scale graph processing on emerging storage devices. In Proceedings of the 17th USENIX Conference on File and Storage Technologies, pages 309–316. USENIX Association, (2019)
6. Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Venkataramani, V.: Tao: Facebook's distributed data store for the social graph. In Proceedings of the 2013 USENIX Conference on Annual Technical Conference, pages 49–60. USENIX Association, (2013)
7. Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, Hsieh, Wilson C., Wallach, Deborah A., Burrows, Mike, Chandra, Tushar, Fikes, Andrew, Gruber, Robert E.: Bigtable: a distributed storage system for structured data. Acm Trans. Comput. Syst. **26**(2), 1–26 (2008)
8. Raju, Pandian., Kadekodi, Rohan., Chidambaram, Vijay., Abraham, Ittai.: Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In Proceedings of the 26th Symposium on Operating Systems Principles, pages 497–514. Association for Computing Machinery, (2017)
9. Dayan, Niv., Idreos, Stratos.: 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, pages 505–520. Association for Computing Machinery, (2018)
10. Lu, Lanyue., Pillai, Thanumalayan Sankaranarayana., Arpaci-Dusseau, Andrea C., Arpaci-Dusseau, Remzi H.: Wisckey: Separating keys from values in ssd-conscious storage. In Proceedings of the 14th USENIX Conference on File and Storage Technologies, pages 133–148. USENIX Association, (2016)
11. li, Yongkun., Liu, Zhen., Lee, Patrick P.C., Wu, Jiayu., Xu, Yinlong., Wu, Yi., Tang, Liu., Liu, Qi., Cui, Qiu.: Differentiated key-value storage management for balanced i/o performance. In Proceedings of the 2021 USENIX Annual Technical Conference, pages 673–687. USENIX Association, (2021)
12. Chan, Helen H. W., Li, Yongkun., Lee, Patrick P. C., Xu, Yinlong.: Hashkv: Enabling efficient updates in KV storage via hashing. In Proceedings of the 2018 USENIX Annual Technical Conference, pages 1007–1019. USENIX Association, (2018)
13. Tang, Chenlei, Wan, Jiguang, Changsheng, Xie: Fencekv: enabling efficient range query for key-value separation. IEEE Trans. Parallel Distrib Syst. **33**(12), 3375–3386 (2022)
14. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**(4), 351–385 (1996)
15. TPC. Tpc-c is an on-line transaction processing. http://www.tpc.org/tpcc/, (2022)
16. Kavalanekar, S., Worthington, B., Zhang, Q., Sharda, V.: Characterization of storage workload traces from production windows servers. In 2008 IEEE International Symposium on Workload Characterization, pp. 119–128. IEEE, Piscataway (2008)
17. Lepers, Baptiste., Balmau, Oana., Gupta, Karan., Zwaenepoel, Willy.: Kvell: The design and implementation of a fast persistent key-value store. page 447-461, (2019)
18. Yu, Geoffrey X., Markakis, Markos, Kipf, Andreas, Larson, Per-Åke, Minhas, Umar Farooq, Kraska, Tim: Treeline: an update-in-place key-value store for modern storage. Proc. VLDB Endow. **16**(1), 99–112 (2022)
19. Zhang, Jianpeng, Lin, Mingwei, Pan, Yubiao, Xu, Zeshui: Crftl: cache reallocation-based page-level flash translation layer for smartphones, pp. 1–9. Piscataway, IEEE Transactions on Consumer Electronics (2023)
20. Luo, Yuhan, Lin, Mingwei, Pan, Yubiao, Zeshui, Xu.: Dual locality-based flash translation layer for nand flash-based consumer electronics. IEEE Trans. Consumer Electron. **68**(3), 281–290 (2022)
21. Pan, Yubiao, Lin, Mingwei, Zhixiong, Wu., Zhang, Huizhen, Zeshui, Xu.: Caching-aware garbage collection to improve performance and lifetime for nand flash ssds. IEEE Trans. Consumer Electron. **67**(2), 141–148 (2021)
22. Pan, Yubiao, Li, Yongkun, Zhang, Huizhen, Chen, Hao, Lin, Mingwei: Gftl: Group-level mapping in flash translation layer to provide efficient address translation for nand flash-based ssds. IEEE Trans. Consumer Electron. **66**(3), 242–250 (2020)
23. Pugh, William W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
24. Bayer, R., Mccreight, E.M.: Organization and maintenance of large ordered indexes. Acta Informatica **1**(3), 173–189 (1972)
25. 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. Indianapolis, pages 143–154. Association for Computing Machinery, (2010)
26. Ren, Kai, Zheng, Qing, Arulraj, Joy, Gibson, Garth: Slimdb: a space-efficient key-value storage engine for semi-sorted data. Proc. VLDB Endow. **10**(13), 2037–2048 (2017)
27. Pan, F., Yue, Y., Xiong, J.: dcompaction: delayed compaction for the lsm-tree. Int. J. Parallel Program. **45**(6), 1310–1325 (2017)
28. Shetty, P., Spillane, R., Malpani, R., Andrews, B., Zadok, E.: Building workload-independent storage with vt-trees. In Proceedings of the 11th USENIX conference on File and Storage Technologies, pages 17–30. USENIX Association, (2013)
29. 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 2018 ACM Symposium on Cloud Computing, page 477-489. Association for Computing Machinery, (2018)
30. Sears, Russell., Ramakrishnan, Raghu.: Blsm: A general purpose log structured merge tree. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pages 217–228. Association for Computing Machinery, (2012)

31. Dayan, Niv., Athanassoulis, Manos., Idreos, Stratos.: Monkey: Optimal navigable key-value store. In Proceedings of the 2017 ACM International Conference on Management of Data, pages 79–94. Association for Computing Machinery, (2017)
32. Li Yongkun, Tian, Chengjin., Guo, Fan., Li, Cheng., Xu, Yinlong.: Elasticbf: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In Proceedings of the 2019 USENIX Annual Technical Conference, pages 739–752. USENIX Association, (2019)
33. Byun, Hayoung, Lim, Hyesook: Learned fbf: Learning-based functional bloom filter for key-value storage. IEEE Trans. Comput. **71**(8), 1928–1938 (2022)
34. Wu, Fenggang., Yang, Ming-Hong., Zhang, Baoquan., Du, David H. C.: Ac-key: Adaptive caching for lsm-based key-value stores. In Proceedings of the 2020 USENIX Annual Technical Conference, pages 603–615. USENIX Association, (2020)
35. Kai, Lu., Zhao, Nannan, Wan, Jiguang, Fei, Changhong, Zhao, Wei, Deng, Tongliang: Tridentkv: a read-optimized lsm-tree based kv store via adaptive indexing and space-efficient partitioning. IEEE Trans. Parallel Distrib. Syst. **33**(8), 1953–1966 (2022)
36. Luo, Siqiang., Chatterjee, Subarna., Ketsetsidis, Rafael., Dayan, Niv., Qin, Wilson., Idreos, Stratos.: Rosetta: A robust space-time optimized range filter for key-value stores. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pages 2071–2086. Association for Computing Machinery, (2020)
37. Zhong, W., Chen, C., Wu, X., Jiang, S.: Remix: Efficient range query for lsm-trees. In Proceedings of the 19th USENIX Conference on File and Storage Technologies, pages 51–64. USENIX Association, (2021)
38. Ge, Xuran., Liu, Yang., Wu, Lizhou., Ou, Yang., Chen, Zhiguang., Xiao, Nong.: Pm-based persistent key value stores: a survey. pages 1–7, (2022)
39. Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., Choi, Y.: Slm-db: single-level key-value store with persistent memory. In Proceedings of the 17th USENIX Conference on File and Storage Technologies, pages 191–205. USENIX Association, (2019)
40. Chen, Youmin., Lu, Youyou., Yang, Fan., Wang, Qing., Wang, Yang., Shu, Jiwu.: Flatstore: An efficient log-structured key-value storage engine for persistent memory. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 1077–1091. Association for Computing Machinery, (2020)
41. Li, Wen-Jie., Jiang, Dejun., Xiong, Jin., Bao, Yungang.: Hilsm: an lsm-based key-value store for hybrid NVM-SSD storage systems. In Proceedings of the 17th ACM International Conference on Computing Frontiers, pages 208–216. Association for Computing Machinery, (2020)
42. Chen, Hao., Ruan, Chaoyi., Li, Cheng., Ma, Xiaosong., Xu, Yinlong.: Spandb: A fast, cost-effective lsm-tree based KV store on hybrid storage. In Proceedings of the 19th USENIX Conference on File and Storage Technologies, pages 17–32. USENIX Association, (2021)
43. Yao, Ting., Wan, Jiguang., Huang, Ping., Zhang, Yiwen., Liu, Zhiwen., Xie, Changsheng., He, Xubin.: Geardb: A gc-free key-value store on HM-SMR drives with gear compaction. In Proceedings of the 17th USENIX Conference on File and Storage Technologies, pages 159–171. USENIX Association, (2019)
44. Conway, Alexander., Gupta, Abhishek., Chidambaram, Vijay., Farach-Colton, Martin., Spillane, Richard P.: Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In Proceedings of the 2020 USENIX Annual Technical Conference, pages 49–63. USENIX Association, (2020)
45. Ray, Madhurima., Kant, Krishna., Li, Peng., Trika, Sanjeev.: Flashkey: A high-performance flash friendly key-value store. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium, pages 976–985. IEEE, (2020)
46. Menon, Prashanth, Qadah, Thamir M., Rabl, Tilmann, Sadoghi, Mohammad, Jacobsen, Hans-Arno.: Logstore: A workload-aware, adaptable key-value store on hybrid storage systems. IEEE Trans. Knowl. Data Eng. **34**(8), 3867–3882 (2022)
47. Chen, Yen-Ting., Yang, Ming-Chang., Chang, Yuan-Hao., Shih, Wei-Kuan.: Parallel-log-single-compaction-tree: Flash-friendly two-level key-value management in kvssds. In Proceedings of the 25th Asia and South Pacific Design Automation Conference, pages 277–282. IEEE, (2020)
48. Wu, Sung-Ming., Lin, Kai-Hsiang., Chang, Li-Pin.: KVSSD: close integration of LSM trees and flash translation layer for write-efficient KV store. In Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition, pages 563–568. IEEE, (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Jianing Zhao was born in Jinzhong, China,in 1998. She received her bachelor's degree in software engineering from Huaqiao University in Xiamen, China in 2016. She is currently studying for a master's degree in the School of Computer Science and Technology, Huaqiao University. Her current research interest is key-value storage.

Yubiao Pan was born in Longyan, Fujian province, China in 1987. He received the B.S. and Ph.D. degree from the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China, in 2010 and 2015, respectively. He is currently an associate professor with the School of Computer Science and Technology, Huaqiao University in Xiamen. He is also a researcher with Xiamen Key Laboratory of Data Security and Blockchain Technology. He has published more than 10 research papers in international journals and conference proceedings. His current research interests include solid-state devices, distributed storage system, and data deduplication.

Huizhen Zhang was born in Longyan, Fujian Province, China in 1983. He received the B.S. degree in Computer Science from University of Science and Technology of China in 2005, and the Ph.D. degree in Computer Architecture from University of Science and Technology of China in 2010. He is currently an associate professor in School of Computer Science and Technology, Huaqiao University in Xiamen. He has published more than 10 research papers in international journals and conference proceedings. His research mainly focuses on reconfigurable computing, compiler, performance evaluation and optimization of computer systems.

Mingwei Lin was born in Putian, Fujian Province, China in 1985. He received his B.S. degree in software engineering in 2009 and Ph.D. degree in computer science and technology in 2014 from Chongqing University, Chongqing, China. Currently, he is a professor with College of Mathematics and Informatics, Fujian Normal University, China. He has published more than 60 research papers in international journals and conference proceedings such as Nonlinear Dynamics, Complexity, International Journal of Intelligent Systems, IEEE Internet of Things Journal, IEEE Access, Sustainable Cities and Society, Artificial Intelligence Review, IEEE Transactions on Consumer Electronics, Journal of the Operational Research Society. He has published three ESI highly cited papers. His research interests include decision making and information fusion. He got the CSC-IBM Chinese Excellent Student Scholarship in 2012.

Xin Luo received the B.S. degree in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 2005, and the Ph.D. degree in computer science from the Beihang University, Beijing, China, in 2011. He is currently a Professor of Data Science and Computational Intelligence with the College of Computer and Information Science, Southwest University, Chongqing, China. He has authored or coauthored over 200 papers (including over 90 IEEE Transactions papers) in the areas of his interests. His current research interests focus on Data Science. Dr. Luo was the recipient of the Outstanding Associate Editor Award from IEEE/CAA JOURNAL OF AUTOMATICA SINICA in 2020. He is currently serving as an Associate Editor for IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, and IEEE/CAA JOURNAL OF AUTOMATICA SINICA.

Zeshui Xu received the Ph.D. degree in management science and engineering from Southeast University, China, in 2003. From April 2003 to May 2005, he was a Postdoctoral Researcher with School of Economics and Management, Southeast University. From October 2005 to December 2007, he was a Postdoctoral Researcher with School of Economics and Management, Tsinghua University, Beijing, China. He is currently a Professor with the Business School, Sichuan University, Chengdu, and also with the College of Sciences, PLA University of Science and Technology, Nanjing, China. He is an IEEE Fellow, IFSA Fellow, IET Fellow, BCS Fellow, RSA Fellow, Distinguished Young Scholar of the National Natural Science Foundation of China, and the Chang Jiang Scholars of the Ministry of Education of China. He has been selected as a Thomson Reuters Highly Cited Researcher (in the fields of Computer Science (2014-2018) and Engineering (2014, 2016, 2018), respectively), included in The World's Most Influential Scientific Minds 2014-2018, and also the Most Cited Chinese Researcher (ranked first in Computer Science, 2014-2018, released by Elsevier). His h-index is 114. He serves as currently the chief editor of Scholars Journal of Economics, Business and Management, and also the associate editors of IEEE Transactions on Fuzzy Systems, Information Sciences, International Journal of Machine Learning and Cybernetics, International Journal of Fuzzy Systems, a member of the advisory boards of Information Fusion, Knowledge-Based Systems, Granular Computing, and also a member of Editorial Boards of more than thirty professional journals. He has contributed more than 550 journal articles to professional journals and Conferences. His current research interests include information fusion, group decision making, computing with words, and aggregation operators.