Journal of Systems Architecture 160 (2025) 103342

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

ProckStore: An NDP-empowered key-value store with asynchronous and multi-threaded compaction scheme for optimized performance ☆

Hui Sun [a,*], Chao Zhao [a], Yinliang Yue [b], Xiao Qin [c]

[a] Anhui University, Jiu long road 111, Hefei, 230601, Anhui, China
[b] Zhongguancun Laboratory, Cuihu North Road 2, Beijing, 100094, China
[c] Auburn University, The Quad Center Auburn, Auburn, 36849, AL, USA

ARTICLE INFO

Keywords:
Near-data processing (NDP)
LSM-tree
Asynchronous multi-threaded compaction
Write amplification
Key-value separation

ABSTRACT

With the exponential growth of large-scale unstructured data, LSM-tree-based key-value (KV) stores have become increasingly prevalent in storage systems. However, KV stores face challenges during compaction, particularly when merging and reorganizing SSTables, which leads to high I/O bandwidth consumption and performance degradation due to frequent data migration. Near-data processing (NDP) techniques, which integrate computational units within storage devices, alleviate the data movement bottleneck to the CPU. The NDP framework is a promising solution to address the compaction challenges in KV stores. In this paper, we propose ProckStore, an NDP-enhanced KV store that employs an asynchronous and multi-threaded compaction scheme. ProckStore incorporates a multi-threaded model with a four-level priority scheduling mechanism–covering the compaction stages of triggering, selection, execution, and distribution, thereby minimizing task interference and optimizing scheduling efficiency. To reduce write amplification, ProckStore utilizes a triple-level filtering compaction strategy that minimizes unnecessary writes. Additionally, ProckStore adopts a key-value separation approach to reduce data transmission overhead during host-side compaction. Implemented as an extension of RocksDB on an NDP platform, ProckStore demonstrates significant performance improvements in practical applications. Experimental results indicate a 1.6× throughput increase over the single-threaded and asynchronous model and a 4.2× improvement compared with synchronous schemes.

1. Introduction

The rapid development of large language models [1], graph databases [2], and social network [3] has led to the generation of real-time large amounts of data, contributing to a global surge in large-scale data. This data is growing exponentially and is increasingly manifested in semi-structured and unstructured formats, in addition to traditional structured data. For example, semi-structured and unstructured data have been grown in recent years according to IDC [4], and they now account for over 85% of total data volume. To cope with the large amount of unstructured data, LSM-tree-based key-value stores (KV stores) [5] have become widely adopted in large-scale storage systems.

LSM-tree structures are popularly used in modern database engines (*e.g.,* LevelDB) [6] RocksDB [7]). In the LSM-tree structure, key-value pairs are first written to an immutable MemTable in memory and then persist to disk as Sorted String Tables (SSTables) once preset threshold is reached. On disk, the LSM-tree is organized hierarchically, with each level having a capacity threshold that increases at a fixed rate as the level number grows. When the amount of data in a level exceeds its threshold, some data migrates to lower levels, potentially causing overlapping key ranges between SSTables in different levels. To maintain data organization and prevent duplication, SSTables with overlapping key ranges must be loaded into memory and merged. The sorted and de-duplicated key-value pairs are then rewritten as new SSTables at a lower level. This process, known as compaction, involves frequent read and write operations that consume a lot of I/O bandwidth between the host and storage devices, thereby delaying foreground requests and degrading system performance.

GPUs, DPUs, and FPGAs General-purpose graphics processing unit (GPGPU), data processing unit (DPU), and field-programmable gate array (FPGA) offer additional computational resources to address compaction performance challenges. Near-Data Processing (NDP), introduced in the late 1990s as the "smart disk" [8], has regained attention

☆ This work is supported in part by National Natural Science Foundation of China under Grants 62472002 and 62072001. Xiao Qin's work is supported by the U.S. National Science Foundation (Grants IIS-1618669 and OAC-1642133), the National Aeronautics and Space Administration, United States (Grant 80NSSC20M0044), the National Highway Traffic Safety Administration, United States (Grant 451861-19158), and Wright Media, LLC (Grants 240250 and 240311).

* Corresponding author.
E-mail addresses: sunhui@ahu.edu.cn (H. Sun), chaozh@stu.ahu.edu.cn (C. Zhao), yylhust@qq.com (Y. Yue), xqin@auburn.edu (X. Qin).

https://doi.org/10.1016/j.sysarc.2025.103342
Received 31 October 2024; Received in revised form 30 December 2024; Accepted 11 January 2025
Available online 24 January 2025

as an emerging computational paradigm. The enhanced computational power within storage devices has fueled interest in NDP. NDP mitigates the overhead of data movement by reducing data movement to the CPU. The NDP paradigm advocates for "computation close to data" as an alternative to the computation-centered approach in large-scale systems. This model enables storage devices to use their internal bus for data processing rather than transfer data to the host, where the results would otherwise be computed. Most existing NDP-empowered KV stores, such as Co-KV [9] and TStore [10], to tackle compaction tasks using a synchronization-based approach. This work splits the compaction task, leveraging either averaging or dynamic time-awareness. In the synchronization model, the host and the device cannot complete tasks simultaneously, leading to long waiting time and inefficient resources usage. PStore [11] addresses waiting time by using an asynchronous model but fails to fully exploit the benefits of this approach due to its single-threaded processing.

To address these issues, we designed an asynchronous NDP scheme, ProckStore, which utilizes a multi-threaded strategy to perform compaction tasks concurrently. All compaction tasks are managed in a thread pool and scheduled using multiple threads, exploiting the benefits of asynchronous processing, where tasks do not interfere with one another. The tasks are executed independently by individual threads. A four-level priority scheduling mechanism is implemented to ensure efficient scheduling of compaction tasks within the thread pool, following the four stages of the compaction process. To address the write amplification issue, a triple-level filtering compaction method is employed, reducing unnecessary writes and alleviating write amplification during compaction on the host side. Furthermore, the transmission process in the NDP architecture and its compaction module is optimized by utilizing a key-value separation technique, minimizing transmission time by sending only the keys to the host. The contributions of this work are summarized as follows

▲ We designed ProckStore with an asynchronous and multi-threaded scheme. Then, compaction tasks are executed independently without interfering with each other in the thread pool, entirely using the asynchronous mode, which significantly improves write performance compared with the synchronous mode and the single-threaded asynchronous scheme.

We designed ProckStore using an asynchronous and multi-threaded architecture. Compaction tasks are executed independently within the thread pool, fully leveraging the asynchronous model. This approach significantly enhances write performance compared to the synchronous model and single-threaded asynchronous scheme.

▲ ProckStore employs a four-level priority scheduling to manage the compaction process, which consists of four stages: compaction trigger, compaction picking, compaction execution, and compaction distribution. This scheduling prioritizes tasks at different stages, ensuring optimal efficiency during asynchronous and multi-threaded compaction.

▲ To optimize performance in the NDP transmission architecture, we implemented key-value separation in the host-side compaction, reducing data transmission overhead. The device-side compaction module employs a cross-level compaction technique to alleviate computational load, thereby improving transmission efficiency and overall system throughput.

▲ ProckStore, an extension of RocksDB on the NDP platform, was evaluated using DB_Bench and YCSB-C. Experimental results demonstrate that ProckStore increases throughput by a factor of 1.6× compared to the single-threaded asynchronous PStore, and achieves a 4.2× throughput improvement over the synchronous TStore.

The paper is organized as follows. Section 2 presents the background and motivation encountered by ProckStore. Section 3 presents a system overview of ProckStore and information on each module. Section 4 lists the hardware and software configurations used in the experiments. Section 5 demonstrates the performance of ProckStore through extensive experiments. Section 6 elaborates on the extended experiments. Section 7 summarizes related work. Finally, we conclude our work in Section 8.

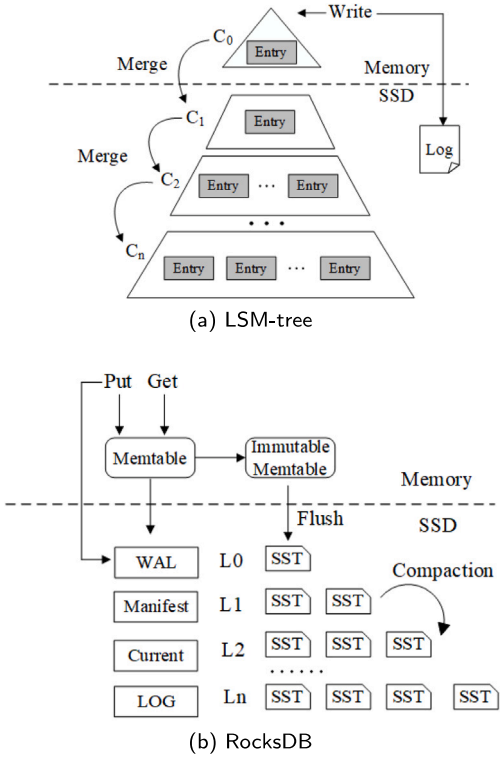

Fig. 1. The structure of LSM-tree and RocksDB. The LSM-tree is composed of components C_0, C_1, and C_n.

2. Background and motivation

2.1. Background

RocksDB is an LSM-tree-based key-value store developed by Facebook, and it is widely used in Facebook's storage systems to achieve high throughput. In RocksDB, the MemTable and Immutable MemTable are stored in memory, while the Sorted String Table (SSTable) is stored on disk. As shown in Fig. 1, key-value pairs from the application are first written to the commit log and then cached in a sorted data structure called the MemTable, which has a limited size (e.g., 4MB) in memory. Once the MemTable reaches its predefined capacity, it is converted into an Immutable MemTable. A background thread then writes the MemTable to disk as a sorted string table (SSTable). On disk, SSTables are organized in levels, with each level growing by a fixed multiple.

In Fig. 1(a), in LSM-tree, the hierarchy represents different components, such as components C_0, C_1....., and C_n. Component C_0 resides in memory. The new write data is first written into the sequential log file and then inserted into an entry placed in C_0. However, the high-cost memory capacity that accommodates C_0 imposes a limit on the size of C_0. To migrate entries to a component on the disk, LSM-tree performs a merge operation when the size of C_0 reaches the threshold, including taking some contiguous segment of entries from C_0 and merging it into a component on the disk. Component C_n (n> 1) resides on the disk in the LSM-tree. Although C_1 is disk-resident, the frequently accessed page nodes in C_1 remain in the memory buffer. C_1 has a directory structure like B-tree but is optimized for sequential access on the disk. The in-memory C_0 servers high-speed writes, and C_n (n> 1) on the disk is responsible for persistence and batch-sequential writes. Through the hierarchical and merging strategies, LSM-tree achieves a balance between write optimization and high-efficiency query.

In Fig. 1(b), the most recently generated SSTable is placed in the lowest level, L_0. SSTables in level L_0 can have overlapping key ranges,

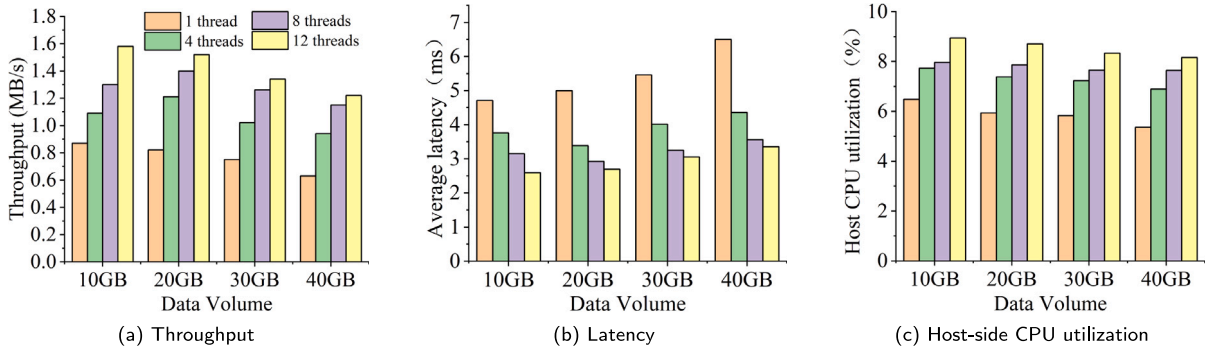

Fig. 2. The results of PStore with different numbers of threads (1, 4, 8, and 12) under the Fillrandom DB_Bench with various data volume.

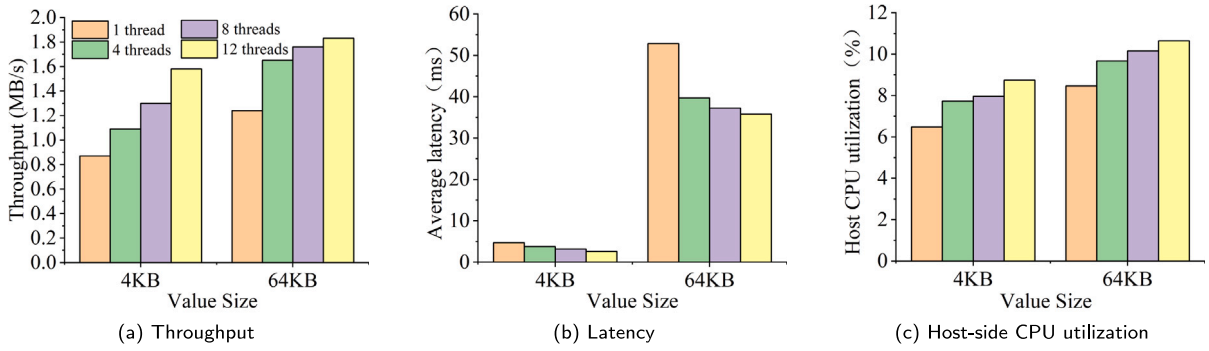

Fig. 3. The results of PStore with different numbers of threads (1, 4, 8, and 12) under the Fillrandom DB_Bench with various value sizes.

while higher levels are organized by key ranges. Each level has a size threshold for its total SSTables. When this threshold is exceeded, the KV store migrates SSTables from level L_k to level L_{k+1} during compaction. The compaction process selects SSTables from level L_k and searches for overlapping key ranges in level L_{k+1}. A merge operation is then performed on the SSTables with overlapping key ranges to produce new SSTables, which are stored in level L_{k+1}. Obsolete SSTables in level L_{k+1} are deleted from the disk. This compaction process incurs computational and storage overhead, which negatively impacts response time and throughput – a significant drawback of the LSM-tree.

Graphical computing [2], machine learning [12], and large language models [1] demand substantial data for model training and inference. The data transfer overhead from storage devices to the CPU for computation becomes higher as data volumes grow, consuming system resources and incurring bottlenecks between storage and memory in high-performance systems. As data volumes increase, the overhead associated with transferring data from storage devices to the CPU for computation rises, leading to resource consumption and performance bottlenecks between storage and memory in high-performance systems. Traditional storage architectures struggle to meet the demands of data-intensive applications under these conditions. NDP mitigates this challenge by fully utilizing the device's internal bandwidth. By incorporating embedded computing units, storage devices can perform computational tasks, offloading these operations from the host and eliminating the overhead of moving large data volumes. The results can then be retrieved from the storage device, reducing the need for additional data movement. Furthermore, the KV store can leverage NDP to perform compaction tasks internally, improving compaction efficiency.

2.2. Motivation

Most existing studies focus on compaction processing in a single-threaded context. For instance, Co-KV and TStore process compaction tasks synchronously and in a single-threaded mode. PStore, on the other hand, demonstrates its effectiveness in an asynchronous and single-threaded setting. Notably, the asynchronous approach allows compaction tasks to be performed independently; however, it is difficult to fully leverage the benefits of asynchronous processing in a single-threaded environment. Therefore, we investigate the performance of PStore using different thread configurations. Fig. 2(a) presents the throughput of PStore under workloads with 4-KB value and various data volumes. We can draw two key observations.

▵ As the number of threads increases, the throughput of PStore does not grow exponentially as expected. The throughput improvement is minimal during multi-threaded writes, particularly when the number of threads is 12.

▵ With a large number of threads, the throughput of PStore increases slowly. Under 20-GB workloads, when the thread count increases from 8 to 12, the throughput only increases by 0.12 MB/s.

These findings indicate that the asynchronous compaction advantages of PStore in single-threaded mode are insufficient to handle the large volume of multi-threaded writes. As a result, increasing the number of threads does not enhance throughput, particularly as the thread count becomes large. While the asynchronous approach in PStore takes into account the computational imbalance between the host and the NDP device, it fails to implement an appropriate asynchronous compaction method. The limitations of the single-threaded mode hinder the full potential of the asynchronous compaction mechanism in the KV store.

As shown in Fig. 2(b), the average latency decreases under workloads with various data volumes, but this reduction is most pronounced when using a small number of threads. Specifically, the most significant decrease occurs between 1 and 4 threads, where the average latency reduces by 27.8%. Additionally, the CPU utilization on the host supports these observations (see Fig. 2(c)), with a 34% increment in 12 threads over 1 thread under 10-GB workloads. The CPU utilization exhibits a 19% increment as the number of threads grows from 1 to 4. The result reveals that PStore is suitable for single- or fewer-threaded workloads, and it is challenging to adapt to multi-threaded applications.

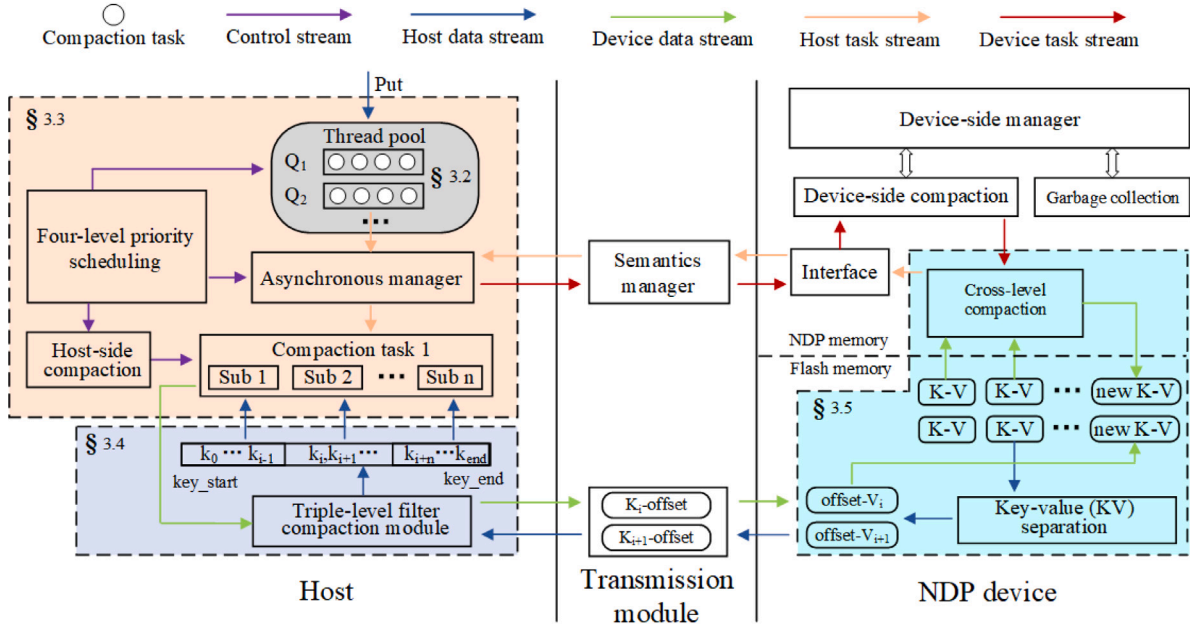

Fig. 4. Overview system of ProckStore. Q_1 is the first compaction-task queue. *Sub i(0<i<n+1)* represents the sub-compaction task of task 1. K_i and V_i denote the *i*th key and value, respectively.

We conducted an experiment to study the impact of multiple threads on the performance under workloads with 4- and 64-KB value sizes. As shown in Fig. 3, we observe similar findings under workloads with various data volumes. The throughput of PStore is improved under the large-sized value. When increasing the number of threads to 4 under workloads with a 64-KB value, the throughput of PStore is 1.65 MB/s. Furthermore, this metric increases to 1.83 MB/s in the case of 12 threads, and there is only an increment of 11% (see Fig. 3(a)). In Fig. 3(b), the average latency decreases when the number of threads increases to 4. The average latency of 4 threads decreases by 24.8% compared to one thread. The degree of decrement becomes little as the number of threads increases. The host-side CPU utilization becomes smoother in Fig. 3(c), but the improvement is still most pronounced when there are a limited number of threads.

Thus, we plane to use a multi-threaded approach to implement the asynchronous compaction mechanism in the KV store. We have redesigned the asynchronous compaction mechanism extended from RocksDB, and fully leveraged its internal multi-threaded capability to develop the asynchronous compaction solution – ProckStore.

3. Design of ProckStore

3.1. System overview

In this paper, we propose ProckStore, an NDP-empowered KV store that incorporates an asynchronous and multi-threaded compaction scheme. ProckStore consists of a host-side subsystem, an NDP device, and a communication module that connects the two sides, as shown in Fig. 4. The host-side subsystem manages I/O requests, while the NDP device, which serves as a storage unit, extends computational resources to process tasks offloaded from the host. The NDP device stores persistent data, with read and write operations akin to those of standard storage devices. We implement various modules on both the host and NDP device to accommodate task-offloading requirements. Data is stored as SSTables in a leveled structure on the NDP device. SSTables are either transferred to the host for compaction or to the NDP for compaction via the communication channel. During transmission, the SSTables pass through a key-value separation module, ensuring that only the keys of the KV pairs are sent to the host for the merge operation. The data flow occurs between the NDP device, the compaction module on the host side, and the compaction module on the device.

As shown in Fig. 4, we illustrate the data flow between the NDP device, the host-side compaction, and the device-side compaction modules. Initially, the data is written from the host side (see host data stream in Fig. 4), and multiple compaction tasks are accumulated in the thread pool. These tasks are allocated to the host and device by the asynchronous manager (see host task stream and device task stream in Fig. 4). After completing the compaction tasks, the data is written to the flash memory inside the NDP device through the transmission module (see device data stream in Fig. 4). The dark blue line represents the host-side data flow, where ProckStore transfers the data from flash memory to the host for compaction tasks. The four-level priority scheduling module manages the thread pool, the host-side compaction module, and controls the asynchronous manager (see control stream in Fig. 4). When a compaction task is generated, the four-level priority scheduling module places it in the compaction queue of the thread pool. It then determines whether the task should be executed on the host or device and notifies the asynchronous manager to allocate the task. When the host executes a compaction task, the scheduling module issues instructions to the host-side compaction module to execute it.

We provide the asynchronous compaction mechanism with the multi-level task queue module in Section 3.2, where compaction tasks are kept in the thread pool. Section 3.3 presents the four-level priority scheduling module, which controls the priority scheduling in the compaction process. The triple-level filtering compaction module is in Section 3.4. We describe the transport mechanism on the NDP device and the cross-level compaction module in Section 3.5.

The host-side asynchronous compaction management module allocates compaction tasks to the device side. A multi-level queue stores the tasks awaiting execution and calculates the computational capabilities of both the host and device. These tasks are then scheduled to the compaction modules on the host and device sides. The host-side compaction module executes the tasks, while the device side must transmit compaction information via the semantic management module, which facilitates communication between the host and device. The processed information is sent to the device-side compaction module for task execution. The four-level priority scheduling module manages the entire process, from task triggering to execution. Data and commands are transmitted between the host and device through the semantic management module. The NDP device encodes (decodes) interacting

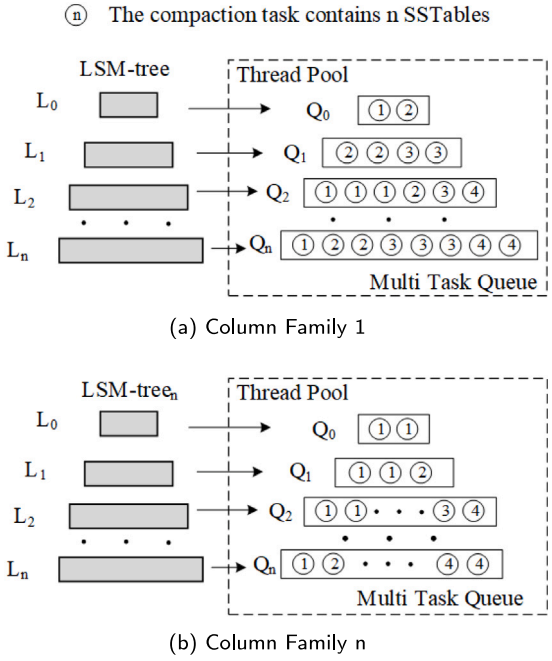

Fig. 5. Multilevel Task Queue in ProckStore.

data, storing the SSTable based on key-range granularity, performing garbage collection, maintaining information, and executing compaction tasks on the files.

3.2. Asynchronous mechanisms

To implement the asynchronous strategy, we decouple the two phases – compaction triggering and execution – to establish conditions for asynchronous compaction. In contrast, the synchronous approach treats the task from compaction trigger to completion as a single process. In the asynchronous mechanism, compaction tasks are continuously generated when the conditions for triggering compaction are met. These tasks, generated during the trigger phase, must be executed. To manage them efficiently, we propose a multi-level task queue that stores compaction tasks uniformly and waits for the asynchronous manager to schedule them. To align with the structure of the LSM-tree, a compaction task queue is assigned to each level, with tasks generated during the trigger phase placed into the task queue at a level, awaiting scheduling. In Fig. 5, ProckStore employs a multi-task queue for each column family. The multi-task queue selects compaction tasks at each level based on a score value.

We implement multi-level task queues in a thread pool. Tasks in each queue are sorted in ascending order by the number of SSTables. A heap sorting algorithm is used in each task queue to ensure sorting occurs in time complexity $O(n \log n)$. The task queue is a double-ended queue, allowing compaction tasks to be allocated from both ends to the host and device sides. Since multiple tasks are scheduled in the queue, a thread pool is used to manage the pending tasks in the compaction queue.

3.3. Four-level priority scheduling

An asynchronous mechanism-based compaction procedure separates the two phases: compaction triggering and execution. To achieve this, we propose a four-level priority scheduling strategy that assigns priority levels to the four steps involved: triggering compaction tasks, generating tasks, allocating tasks, and executing tasks. This strategy ensures efficient execution of the asynchronous compaction process.

In the case of a multi-level queue, it is important to prevent the queue from becoming starved of compaction tasks. On both the host and device sides, one side may pause the compaction task while waiting for new task allocations upon completion of the task allocation. Consequently, the triggering conditions must be adjusted to trigger more frequently, ensuring a sufficient number of compaction tasks are available in the task queue. Additionally, different priorities must be set for scheduling various tasks. ProckStore assigns a score to each priority level, see Fig. 6

First-level Priority: the priority of the compaction trigger. In the stage of triggering a compaction task, the goal of ProckStore is to select the level most urgently needed to perform the compaction task. ProckStore sets the first_score to realize the prioritization of the compaction triggering stage. Due to the structure of the LSM-tree, the score in level L_0 is calculated as the ratio of the number of SSTables to the threshold value of level L_0. However, the score in other levels is calculated as the ratio of the total size of SSTables to the threshold value of the level. Thus, the calculation of first_score is also divided into level L_0 and other levels, see the following equation.

$$\text{first\_score} = \begin{cases} \frac{N_{\text{sst}} - N_{\text{no\_comp}} - N_{\text{being\_comp}}}{N_{\max}}, & \text{level i, i} = 0 \\ \frac{S_{\text{sst}} - S_{\text{no\_comp}} - S_{\text{being\_comp}}}{S_{\max}}, & \text{level } i, i > 0 \end{cases} \quad (1)$$

where N_{sst} and N_{max} denote the number of SSTables and the threshold of the number of SSTables in level L_0, respectively. N_{no_comp} and N_{being_comp} denote the number of compaction tasks in level L_0 that have been picked into the task queue to be executed and the number of compaction tasks in level L_0 that are executed and contain SSTables. S_{sst} and S_{max} denote the size of the total data volume of SSTable in level i and the threshold of the data volume of SSTable in level i, respectively. In contrast, S_{no_comp} and S_{being_comp} denote the data volume of SSTable included in the compaction task to be executed in the queue of tasks picked in level i and the current compaction task being executed. The data volume containing SSTable is being executed in the compaction task. Different from the RocksDB score, we can see that the SSTables that have been picked into the compaction queue and the SSTables that are involved in compaction tasks are subtracted to reduce the number of SSTables that are not in the level, which makes the calculation of the first_score more accurate.

When a level triggers a compaction, the generated compaction task will be put into the corresponding task queue of the level, and the compaction module will wait for its processing. See Fig. 6. In an asynchronous trigger mode, the compaction task will not be executed immediately, and the asynchronous compaction manager will have to wait for it to be scheduled. The device side triggers the compaction task according to the first_score of each level and places it into the task queue. The device triggers the compaction task according to first_score (select the maximum value) in each level and puts it into the task queue. It provides the basis for prioritizing the compaction triggering and the execution between levels, which is the first-level prioritization.

A second level of prioritization is the prioritization of SSTable picking. In the compaction task generation phase, we select some SSTables in the level and all the overlapping SSTables from the following level. These SSTables are conducted and merged in the compaction operation. ProckStore puts the compacted SSTables into the compaction_queue and then reads the first file information that needs to be compacted from the queue. The compacted SSTables in the queue perform compaction operations sequentially without considering the hot and cold data and the size of the compaction task. Thus, the information about the number of overlapping SSTables with the lower level is added to the FileMetaData of each SSTable. The second_score is set to the number of overlapping SSTables, and the meta-information is sorted in ascending order by each level following the size of the second_score, see Eq. (2), as follows

$$\text{second\_score} = Overlap_{\text{sst}}, \text{ for SSTable} \quad (2)$$

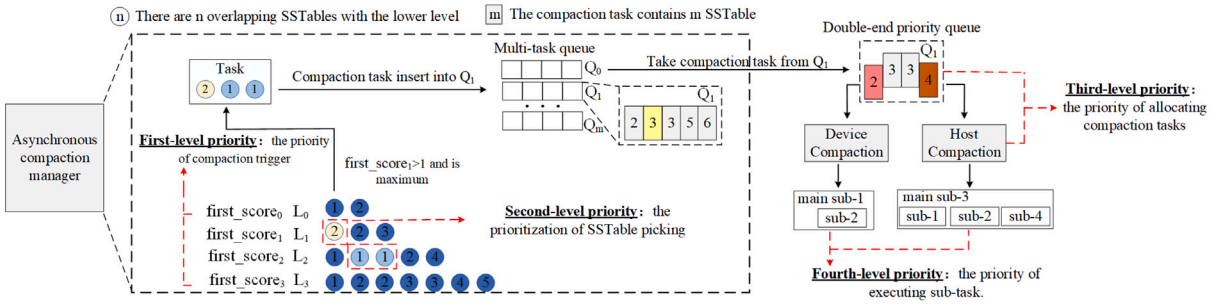

Fig. 6. Four-level priority scheduling in ProckStore. The light gold circles represent SSTables selected for compaction at the current level, while the light blue circles denote SSTables selected for compaction at the next level. The dark blue circles indicate SSTables that are not selected for compaction. The yellow rectangles represent newly generated compaction tasks, the light red rectangles signify compaction tasks assigned to the device side, and the dark orange rectangles represent compaction tasks assigned to the host side.

where $Overlap_{sst}$ denote the number of overlapping SSTables. The SSTable with the smallest number of overlapping SSTables at the lower level is prioritized to select the SSTable to be compacted. Compaction and the metadata information of SSTable are maintained using a linked list to facilitate insertion and deletion. However, we query the overlap of SSTables with the lower level and cost ***O(n)*** time complexity to maintain the order of the linked list. We can ensure that the minimum number of SSTables is selected in each compaction, reducing compaction time.

A third level of priority is the priority of allocating compaction tasks. In the stage of compaction task allocation, we consider the different computational capabilities of the host and the device sides for compaction tasks. Meanwhile, the compaction processing efficiency varies with the configurations of the host and the device and the data paths of read, write, and transmission. Therefore, it is necessary to select appropriate compaction tasks for both the host and the device. When all the SSTables involved in compaction are selected, the compaction task information is generated and inserted into the compaction task queue. The compaction task queue of each level is a double-ended priority task queue, which is sorted in ascending order according to the number of SSTables in a compaction task. The queue is heap sorted with time complexity $O(nlogn)$. Initially, the host obtains the tasks from the left side of the queue with fewer SSTables, and the device side gets the tasks from the right side with more SSTables. The host and the device sides record the compaction time.

During the compaction process, the host and the device record the time cost of five consecutive compaction tasks and data volume of compacted SSTables. The third_score is the ratio of the time taken for five consecutive compaction tasks to the data volume of the compacted SSTables, which is given as

$$\text{third\_score} = \begin{cases} \frac{S_{\text{host\_sst}}}{T_{\text{host\_comp}}}, & \text{for host} \\ \frac{S_{\text{device\_sst}}}{T_{\text{device\_comp}}}, & \text{for device} \end{cases} \tag{3}$$

where S_{host_sst} and T_{host_comp} denote the total data volume of compacted SSTables on the host and the time cost, respectively. S_{device_sst} and T_{device_comp} represent the total data volume of compacted SSTables on the device and the time cost, respectively. We use the third_score to evaluate the compaction processing capability of both the host and device sides. The side with a higher compaction processing capability handles tasks containing a large number of SSTables from the right end of the queue, while the other side handles tasks from the left end.

The larger the value of third_score[1] is, the less efficient the compaction is. Compared $third\_score_{host}$ with $third\_score_{device}$, there are three cases: (1) the score of the host side is greater than that of the device side; (2) the score of the device side is greater than that of the host side; (3) the score of the host side is equal.

In case (1), the default rules of acquiring tasks in the queue remain unchanged, and case (2) is changed into a situation in which the device side fetches the tasks from the left side of the queue, and the host obtains the tasks from the right side of the queue. In case (3), the default rules for taking tasks are still maintained, and the host side and the device side re-record and calculate the compaction time at one end, then make judgments according to the comparison results. In a running process, the configuration of the host and device sides cannot change, so the queue to get the task rules decided after the numerical comparison is no longer carried out. The decision of the task to get the rules cannot be changed in this process.

A fourth level of prioritization is the priority of executing sub-task. After selecting the SSTables, these SSTables are integrated into a complete compaction task that reaches the stage of compaction execution on the NDP device and the host. The compaction task is decomposed into multiple sub-tasks, which can be executed in parallel on the device. Notably, *sub-task* refers to as sub-compaction that are performed in the multi-threaded compaction mechanism in RocksDB. In a compaction process, the primary thread first executes a sub-task. Notably, the first sub-task is designated as the main thread for execution by default. The rest of the sub-task creates many sub-threads to be executed concurrently. Then, the primary thread merges the results and writes them back in a unified manner.

The amounts of data and execution time are different in the sub-task. The computational resources are underutilized by default. To address this issue, we prioritize the concurrent execution process of sub-task. Let us have $fourth\_score = S_{SST}$, where S_{SST} denotes the total data volume of SSTable in each sub-task. When dividing the sub-task tasks, we compare the data size of each sub-task. The sub-task containing the smallest data is set to be the highest priority. It means that the smaller the fourth_score is, the higher the priority is, and the highest-priority sub-task is placed into the primary thread for compaction. The compaction execution time can be illustrated as

$$T_{exe} = T_{pthread} + T_{subthread}, \tag{4}$$

where $T_{exe}, T_{pthread}, and\ T_{subthread}$ represent the overall execution time, primary thread execution time, and sub-thread execution time. The sub-task with the least execution time is placed into the primary thread for execution to reduce the execution time. Notably, the sub-thread execution time is determined by the sub-task with the longest execution time. This procedure cannot affect the execution time of sub-tasks, thereby reducing the overall execution time and improving the system performance.

3.4. Triple-level filter compaction

The asynchronous compaction method of ProckStore improves the compaction performance; however, this method brings the write amplification problem. Therefore, we propose the mechanism of triple-level filtering compaction (see Fig. 7) . In a compaction procedure,

[1] It indicates that the compaction operation spends more time processing an SSTable.

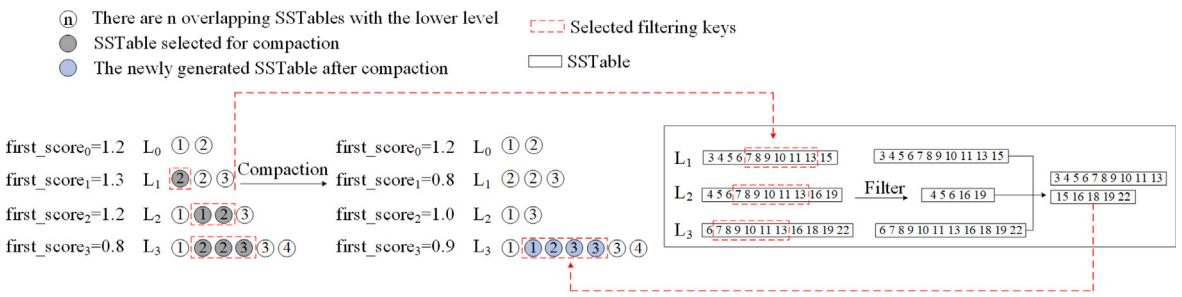

Fig. 7. The triple-level filter compaction in ProckStore.

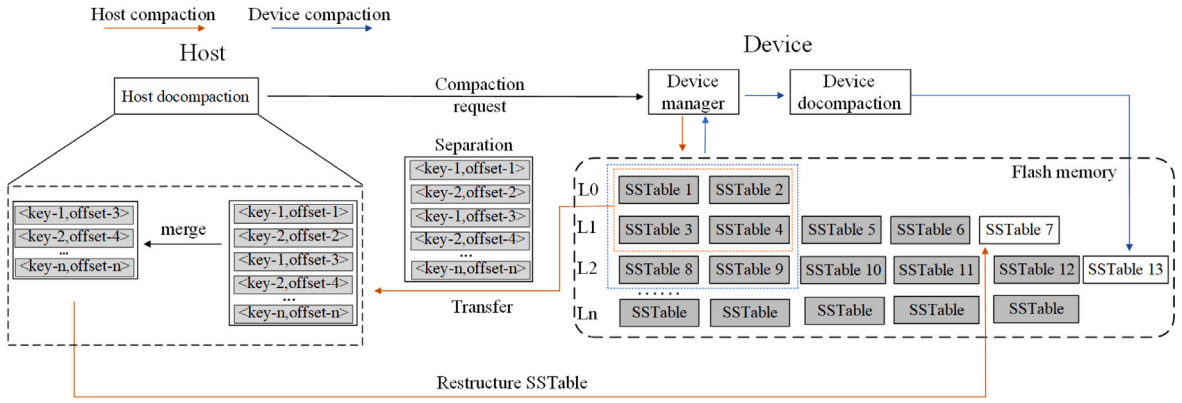

Fig. 8. The transmission module between the host and the device in ProckStore.

triple-level filtering compaction involves SSTables from three levels to remove duplicate data. The triggering of the triple-level filtering mechanism, however, requires certain conditions to be met. When performing the compaction involving SSTables in levels L_i and L_{i+1}, ProckStore first_score value of level L_{i+1}. If the value is greater than 1, the triple-level filtering compaction is triggered, involving SSTables from level L_{i+2} that overlap with those from level L_{i+1}. This mechanism helps reduce duplicate writes and alleviates write amplification.

As triple-level filtering compaction contains overlapping-key-range SSTables of levels L_i, L_{i+1}, and L_{i+2}, it causes the problem of excessive compaction data. When performing the three-level compaction, some key ranges can exist levels L_i, L_{i+1}, and L_{i+2}. This key ranges can be deleted and filtered at the intermediate level (*i.e.,* level L_{i+1}), which cannot affect the update of new keys in level L_i to L_{i+2} or the merging of old keys in level L_{i+2}. At the stage of generating compaction tasks, we mark the duplicate key range in the three levels when picking the overlapping SSTables from the three levels and filter the duplicate key ranges in the three levels out of level L_{i+1} in advance. Then, the newest keys in level L_i and the oldest keys in level L_{i+2} are retained. This approach reduces the data volume involved in compaction by reducing redundancy across the three levels, thereby alleviating the issue of excessive data in compaction operations.

As shown in Fig. 7, when level L_1 performs compaction, the score of level L_2 is greater than 1 to satisfy the condition of triple-level filter compaction. Compared the key ranges of levels L_i, L_{i+1}, and L_{i+2}, ProckStore filters out and deletes the same keys that exist across the three levels. In Fig. 7, the keys 7, 8, 9, 10, 11, and 13 are filtered out from level L_2 before performing the compaction operation. These keys are placed in the compaction queue, awaiting the asynchronous manager to initiate the compaction. According to Eq. (1), these keys are marked as S_{no_comp}, causing the first_score value in levels L_1 and L_2 to drop below 1 due to the subtraction of these keys. This results in a reduction of excess data in the level. The default compaction method in ProckStore merges the SSTables with overlapping key ranges in levels L_i and L_{i+1} and then writes new SSTables into level L_{i+1}. This process introduces write amplification. The SSTables newly written to level L_{i+1} may immediately needs to be combined with the SSTables with overlapping key ranges in level L_{i+2} to form a new compaction task. These SSTables are merged and the new data are written to level L_{i+2}, resulting in additional write amplification for data previously written to level L_{i+1}. Consequently, this procedure incurs two instances of write amplification.

The triple-level filtering compaction combines all the overlapping-key-range SSTables in the three levels to perform compaction. The data in level L_i is written to level L_{i+2}, which eliminates a compaction process and the write amplification from level L_{i+1} to L_{i+2}.

3.5. Transmission in ProckStore

In ProckStore, data is transferred between the host and device sides, as shown in Fig. 8. During compaction, a large amount of data is read from the NDP device to the host for compaction, which involves transferring many KV pairs. This results in significant data migration overhead. To address this issue, we employ key–value separation for compaction to reduce the data transfer overhead, which minimizes data migration between the host and the NDP device, reduces write amplification, and improves compaction performance. In the compaction process, only the keys of the KV pairs are read, sorted, and written to the NDP device. The key size is less than 1 KB, while the value size exceeds 1 KB. During compaction on the host side, the NDP device transmits only the keys to the host, while the device processes the values locally. Afterward, the host processes the values, sends them back to the device, and integrates them into an SSTable. This approach significantly reduces data migration between the CPU and memory on the host side and minimizes the overhead of data transfers between the device and host.

The key–value separation mechanism is implemented during host-side compaction, with the entire KV pair stored on the NDP device. The key is processed during host-side compaction, reducing both device-to-host data transfers and host-to-device compaction operations. Based

on the compaction information, the device separates the key from the value in the SSTables. The key array stores the address of each value, which is used for subsequent reorganization. The keys are then sent to the host for a sort-merge operation. Afterward, the compacted keys are sent back to the device, where they are reorganized into new SSTables based on the value addresses. There are three threads for each step: (1) the separation thread on the device, (2) the merge operation thread on the host, and (3) the key–value reorganization thread on the host. The host-side compaction task is divided into the following three steps:

▵ Step 1. The key–value separation thread in the NDP retrieves the KV pairs based on the SSTable data format. The key or value is stored in the corresponding array in the NDP device's memory. In the key array, each key records the subscript of its corresponding value, and the time complexity for searching the array is ***O(1)***. The value array is transferred to the NDP device via memory sharing and waits for the sorted key array to be fetched from the host. The key array is transferred to the host via the host-NDP interface.

▵ Step 2: The host fetches the key array, sorts the individual keys, and sends the sorted key array to the NDP device for restructuring. All these steps are organized within a single thread.

▵ Step 3: Upon receiving the new key array, the NDP device finds each key's corresponding value based on its subscript. Simultaneously, the device reconstructs the new SSTables according to the order of the keys. To minimize data transfer time between the host and device, the data volume is reduced, and a separate transfer thread ensures that the communication between the host and device remains unaffected, minimizing transmission latency.

As shown in Fig. 8, only the keys (which are reconstructed on the host side) are passed between the host and the device. When the host performs compaction, a compaction request is sent to the device, which then provides the necessary data from the NDP device. SSTables 1 and 2 from level L_0 and SSTables 3 and 4 from level L_1 are separated. The duplicate keys and offset addresses are passed to the host, which executes the compaction procedure. After deduplication, the keys are re-transmitted to the NDP device, where they are reorganized into a new SSTable (SSTable 7) in level L_1.

By reducing the transmission overhead on the host side, the device reduces the compaction task's time cost, aligning with the NDP architecture's requirements. At the fourth priority level, the host handles most of the compaction tasks, which contain more SSTables, thereby relieving the device's computational load. However, the NDP device not only processes the values for the host but also handles the KV pairs in the compaction task, which increases the device's processing pressure. To alleviate this, cross-level compaction is employed to reduce computational strain on the NDP device.

When a compaction process is triggered in level L_i and the first_score of level L_{i+1} exceeds one, cross-level compaction is initiated. This process searches for SSTables with overlapping key ranges in the subsequent level L_{i+2}. Unlike traditional compaction, cross-level compaction in ProckStore continuously searches for overlapping key-range SSTables in level L_{i+2}. Subsequently, SSTables from levels L_i, L_{i+1}, and L_{i+2} undergo compaction, and the newly generated SSTables are written to level L_{i+2}.

The trigger selection in level L_i follows the priority rules, while the selection of SSTables in levels L_{i+1} and L_{i+2} is based on their second_score(see Eq. (2)). SSTables written to level L_{i+1} in traditional compaction may be written to level L_{i+2} through cross-level compaction. This cross-level approach helps balance the SSTable distribution across levels, reducing the number of compaction operations. However, it introduces a drawback: compaction involving many SSTables increases compaction time. For the NDP device, data transmission time can be ignored, thereby reducing overall compaction time. As illustrated in Fig. 8, SSTables 1 and 2 in level L_0, SSTables 3 and 4 in level L_1, and SSTables 8 and 9 in level L_2 are involved in compaction on the NDP device, and new data is written into SSTable 13 in level L_2. With an asynchronous mechanism, priority scheduling, and optimized data transmission under the NDP-empowered KV store, ProckStore efficiently optimizes the compaction process.

4. Experimental settings

Platform. We implemented ProckStore based on RocksDB and conducted experiments to assess its performance. To evaluate ProckStore, we constructed a test platform simulating the NDP architecture, where data transfer between the host and NDP device occurs over Ethernet. Although this platform was used for validation, ProckStore is scalable to real NDP platforms. SocketRocksDB, a version of RocksDB deployed on the NDP collaborative framework, was used as the baseline. TStore, PStore, and ProckStore all share the NDP-empowered storage framework. The experimental platform comprises two subsystems: a host-side and a device-side NDP subsystem. The host system is equipped with an Intel(R) Core(TM) i3-10100 CPU (8 cores) and 16 GB of DRAM, while the NDP device runs on an ARM-based development board with four Cortex-A76 cores, four Cortex-A55 cores, 16-GB DRAM, and a 256 GB Western Digital SSD. A network cable connects the host to the NDP device, with a bandwidth of 1000 Mbps.

The host system runs Ubuntu 16.04, and RocksDB version 6.10.2 is employed. The NDP platform uses a lightweight embedded operating system. Data transfer between the host and the NDP device is facilitated by the SOCKET interface, replacing the standard POSIX interface to ensure efficient data transmission. In RocksDB, the buffer and SSTable sizes are set to 4 MB, the block size is 4 KB, and the level settings remain at default values. The number of sub-tasks on the host is limited to 4, and all other configuration parameters in RocksDB are set to default values.

Workloads. In this section, we evaluate the performance of ProckStore under realistic workloads. The details of the DB_Bench and YCSB-C workloads used in the experiments are presented in Table 1. The DB_Bench workload is used to assess random-write performance.

Table 1 presents the different workloads in the "Type" column. In addition, db_bench_1 is configured in random-write mode with a fixed value size of 1 KB and varying data sizes (10 GB, 20 GB, 30 GB, 40 GB), db_bench_2 is configured in random-write mode with multiple value sizes (1 KB, 4 KB, 16 KB, 64 KB) and two data volumes (10 GB and 40 GB). We also employ YCSB-C to measure the ProckStore's performance under mixed read–write workloads.

5. Performance evaluation

We conduct experiments to evaluate the performance of ProckStore in terms of throughput, latency, and write amplification (WA).

5.1. Performance under DB_Bench with various data volumes

In this section, we evaluate the performance of ProckStore using DB_Bench with various data volumes and a 4-KB value. Fig. 9 illustrates the impact of data volume on performance, focusing on throughput, WA, CPU utilization, and bandwidth. ProckStore delivers peak performance with 10-GB workloads, achieving up to 48% higher throughput compared to PStore, and an average improvement of 40%. Under 40-GB workloads, the WA of TStore and SocketRocksDB reaches its maximum, while ProckStore's WA remains constant at 1.4 across all cases. Under 30-GB workloads, ProckStore's throughput decreases by an average of 67% and 61% compared to TStore and SocketRocksDB, respectively. This performance decrement is attributed to the frequency of compaction operations, which consume bandwidth and degrade overall performance. PStore exhibits lower CPU utilization than ProckStore across all workloads. The multi-threaded approach in ProckStore optimizes the utilization of computing resources. In contrast, SocketRocksDB prioritizes data storage over compaction, leading to lower CPU utilization than PStore and ProckStore.

Table 1
Workload Characteristics used in the Experiment.

Workloads in DB_Bench				
Type	Feature	Fillrandom	Value Size (1 KB)	Data Size (10 GB)
db_bench_1	100% writes	✓	4×	1×, 2×, 3×, 4×
db_bench_2	100% writes	✓	1×, 4×, 16×, 64×	1×, 4×

Workloads in YCSB-C					
Type	Feature	Data Size		Record Size	Distribution
		Load (10 GB)	Run (10 GB)	(1 KB)	
A	50% Reads, 50% Updates	1×, 2×	1×, 2×	1×	Zipfian
B	95% Reads, 5% Updates				Zipfian
C	100% Reads				Zipfian
D	95% Reads, 5% Inserts				Latest
E	95% Range Queries, 5% Inserts				Uniform
F	50% Reads, 50% Read-Modify-Writes				Zipfian

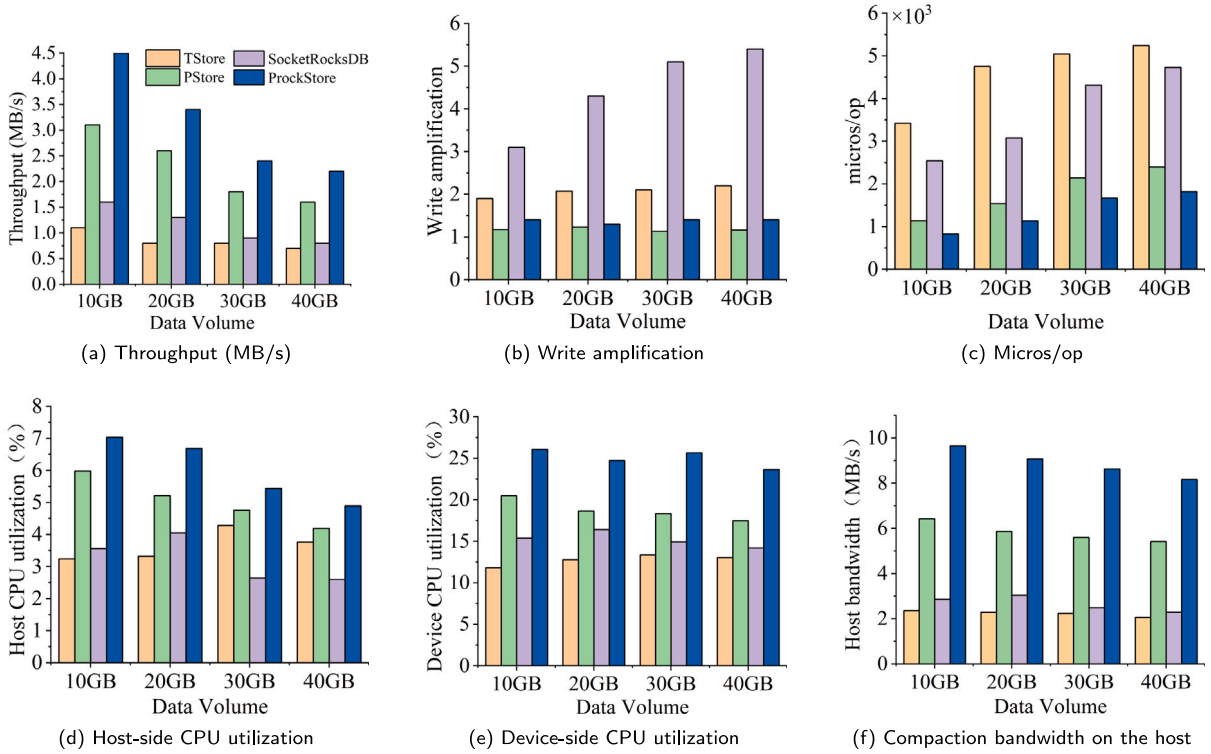

Fig. 9. The results of TStore, PStore, SocketRocksDB, and ProckStore under Fillrandom DB_Bench with various data volumes.

5.1.1. Write amplification (WA)

A large WA indicates significant duplication of write operations, which degrades system performance. In SocketRocksDB, WA is primarily caused by write-ahead log and compaction on the host. WA increases with the amount of data, as the number of compaction operations is proportional to the data size. As shown in Fig. 9(b), under a 10-GB workload, the WA of TStore and PStore is reduced by 39% and 62%, respectively, compared to SocketRocksDB, which performs all compaction tasks on the host. By offloading a portion of the compaction tasks to the NDP device, TStore and PStore reduces WA. Notably, ProckStore exhibits a 55% reduction in WA. A similar trend is observed under 20- and 30-GB workloads. For a 40-GB workload, WA is reduced by 36.4%, and 72.0% for ProckStore, respectively, compared to TStore, and SocketRocksDB.

5.1.2. Throughput

In Fig. 9(c), the operation time of ProckStore ranges from 828.35 micros/op to 1819.18 micros/op. The operation time of ProckStore is lower than that of SocketRocksDB because it takes less to execute write and read operations.ProckStore reduces operation time by 72.8% compared to TStore, under a 20-GB workload. At the same time, under a 40-GB workload, ProckStore reduces the operation time by 24.0% and 61.5%, compared to PStore and SocketRocksDB. In Fig. 9(a), with a 40-GB dataset, the throughput of ProckStore is 4.15× and 1.47× higher than that of TStore and PStore. Meanwhile ProckStore achieves a throughput of 2.75× higher than SocketRocksDB. Under a 10-GB workload, ProckStore achieves 2.81× write throughput of SocketRocksDB through the multi-threaded asynchronous approach. In addition, with a 10-GB dataset, the throughput of ProckStore is 45.2% higher than PStore.

Other KV stores (excluding SocketRocksDB) leverage collaborative strategies between the host and NDP device to accelerate compaction,

thereby enhancing throughput. ProckStore optimizes resource allocation with a multi-threaded asynchronous approach, which improves performance. Its throughput exceeds 4.57 MB/s, achieving a 48% improvement over PStore.

5.1.3. CPU utilization

CPU utilization refers to the proportion of CPU resources consumed by the KV stores under different workloads. TStore utilizes a single-threaded approach on both the host and device, leading to low CPU utilization (see Figs. 9(d) and 9(e)). As a result, TStore's CPU utilization is lower than that of SocketRocksDB. Despite leveraging multi-threaded concurrency, SocketRocksDB faces a transmission bottleneck between the host and the device. During task processing, the host quickly performs merge operations; however, there is significant latency during read and write operations. By offloading a portion of tasks to the NDP device, ProckStore reduces CPU idle time and improves CPU utilization on the host. Compared to SocketRocksDB, ProckStore achieves improvements of 97% and 89% in CPU utilization under 10-GB and 40-GB workloads, respectively. ProckStore demonstrates the highest host-side CPU utilization, peaking at 7.03% under a 10-GB workload.

ProckStore's multi-threaded method on the host further enhances CPU utilization. As shown in Fig. 9(e), PStore employs a single-threaded, asynchronous method, offering greater flexibility than traditional scheduling models. Furthermore, reduced compaction time increases the device-side CPU utilization of PStore by over 20.49%, a 73% improvement compared to TStore under a 10-GB workload. In ProckStore, device-side CPU utilization is further enhanced through cross-level compaction. This metric increases by 27%, 33%, 40%, and 35% compared to PStore under 10-, 20-, 30-, and 40-GB workloads, respectively.

5.1.4. Compaction bandwidth

The compaction bandwidth unveils the compaction performance of a KV store. In this paper, the term "compaction bandwidth" refers to the host-side compaction bandwidth, as our proposed ProckStore primarily focuses on optimizing host-side performance. For instance, the four-level priority scheduling in Section 3.3 prioritizes four steps—triggering, task generation, task allocation, and task execution on the host—to perform asynchronous compaction efficiently. The triple-level filter compaction in Section 3.4 combines two compaction procedures into one, thereby improving host-side compaction performance. Therefore, we define compaction bandwidth as the ratio of compaction time to the amount of compacted data on the host side. SocketRocksDB performs compaction tasks on the host, while other KV stores provide compaction bandwidth on the host and NDP device. In Fig. 9(f), the single-threaded TStore fails to fully leverage the multi-core computational capabilities of the host, resulting in an average bandwidth of 2.35 MB/s.

In contrast, SocketRocksDB uses the multi-threaded method to enhance the bandwidth to 2.86 MB/s, which outperforms other KV stores. This is because the host handles all the tasks, resulting in much total data. The collaborative solution improves processing efficiency on the host. Under 40-GB workloads, ProckStore's bandwidth improves by 3.56× and 1.51× over SocketRocksDB and PStore, respectively.

5.2. Performance under DB_Bench with various value sizes

We configured the workloads with various value sizes and two data volumes (10 GB and 40 GB). The large-sized value increases the compaction overhead while improving the throughput under workloads with a fixed-size data volume. ProckStore maintains optimal performance under workloads with different value sizes and two data volumes (see Figs. 10 and 11). ProckStore's throughput increases on average by 63.1% and 77.7% compared to PStore and SocketRocksDB in the case of 1-KB value (see Fig. 10(a)). The performance increases at 64 KB because large-value workloads trigger more frequent compaction and shorter running time. ProckStore has the best performance in terms of bandwidth, with an average improvement of 1.67× compared to PStore and 2.32× compared to SocketRocksDB across all different value sizes. Meanwhile, ProckStore achieves the highest host- and device-side CPU utilization. In Fig. 11(e), the device-side CPU utilization of PStore and ProckStore are similar due to task stacking on the device under large data volumes.

5.2.1. Write amplification (WA)

With increasing value sizes, the amount of data on the host grows, exacerbating WA in TStore and SocketRocksDB. In Fig. 11(b), the WA of TStore and SocketRocksDB is the lowest (2.18 and 5.2) with a 16-KB value. Under 1-KB value workloads, WA increases to 2.39 and 6.1, respectively. ProckStore's WA is unaffected by host-side compaction. Under 1-KB and 64-KB workloads, ProckStore reduces WA by 76.2% and 75.1%, respectively, compared to SocketRocksDB. However, it increases to 76.4% and 77.6% under 10-GB workloads. This improvement is due to ProckStore's triple-filter compaction on the host, which reduces compaction operations and the volume of compacted data.

5.2.2. Throughput

In Figs. 10(a) and 11(a), ProckSotre's average throughput ranges from 3.8 MB/s to 5.1 MB/s and 2.7 MB/s to 4.0 MB/s under 10-GB and 40-GB workloads, respectively. It is worth noting that ProckStore's throughput increases compared with PStore, indicating lower response times to foreground requests. Compared with SocketRocksDB, ProckStore improves by 2.04× and 2.1× under 40-GB workloads with 1-KB and 16-KB values, respectively. Compared with PStore, ProckStore improves throughput by 1.51× and 1.58× (see Fig. 11(a)). In particular, compared with TStore, ProckStore archives 4.1× and 2.68× improvement under workloads with 4-KB and 64-KB values, respectively.

5.2.3. CPU utilization

Large-sized values increase compaction overhead and host-side CPU utilization, peaking under workloads with a 64-KB value. ProckStore's host-side and device-side CPU utilization reach 10.83% and 29.11%, respectively (see Figs. 10(e) and 11(d)), while SocketRocksDB's values are 8.34% and 18.28%. Additionally, ProckStore's CPU utilization on both sides is 8.27% and 25.39% under 40-GB workloads with 1-KB values. On average, ProckStore's CPU utilization is 3.35× and 4.1× higher than TStore and SocketRocksDB, respectively, and outperforms PStore in both host- and device-side CPU utilization under all workloads.

5.2.4. Compaction bandwidth

In Figs. 10(f) and 11(f), the compaction bandwidth of the KV stores varies. TStore's device-side bandwidth peaks at 3.14 MB/s, while ProckStore shows an average improvement of 4.29× and 1.61× over TStore and PStore, respectively. SocketRocksDB leverages multi-threaded parallelism to enhance computation and reduce processing time, leading to superior bandwidth performance under 40-GB workloads across all value sizes. However, PStore achieves higher bandwidth than SocketRocksDB under 10-GB workloads. ProckStore outperforms all other stores in terms of bandwidth across all workloads, achieving a 3.54× improvement over SocketRocksDB under workloads with a 64-KB value.

5.3. Performance under YCSB-c

YCSB-C provides real-world workloads, which we use to evaluate the compaction performance of TStore, PStore, SocketRocksDB, and ProckStore. We configure this workload with two types of data volumes: 10 GB and 20 GB in the Load and Run phases. We define the configuration with 10 GB Load and 10 GB Run as small data volumes, and 20 GB Load and 20 GB Run as large data volumes. We use six types of workloads in the experiment.

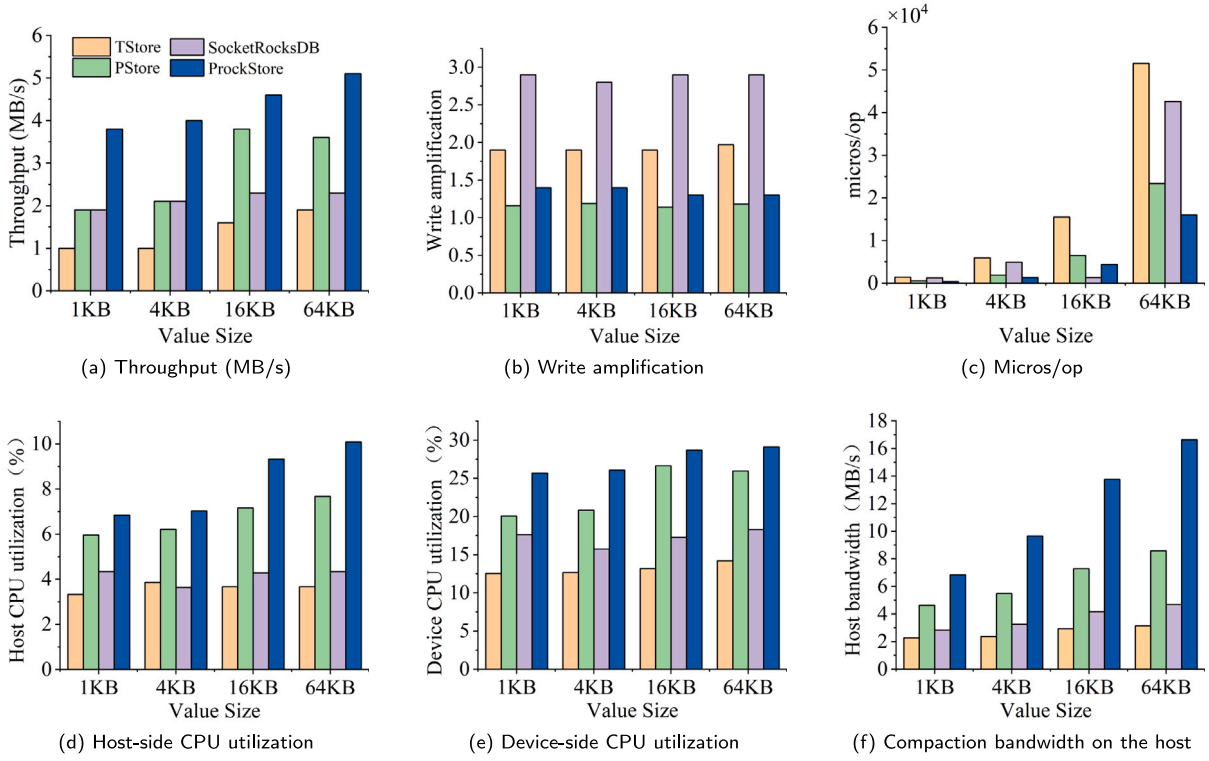

Fig. 10. The results of TStore, PStore, SocketRocksDB, and ProckStore under Fillrandom DB_Bench with 10-GB data volume and various value sizes.

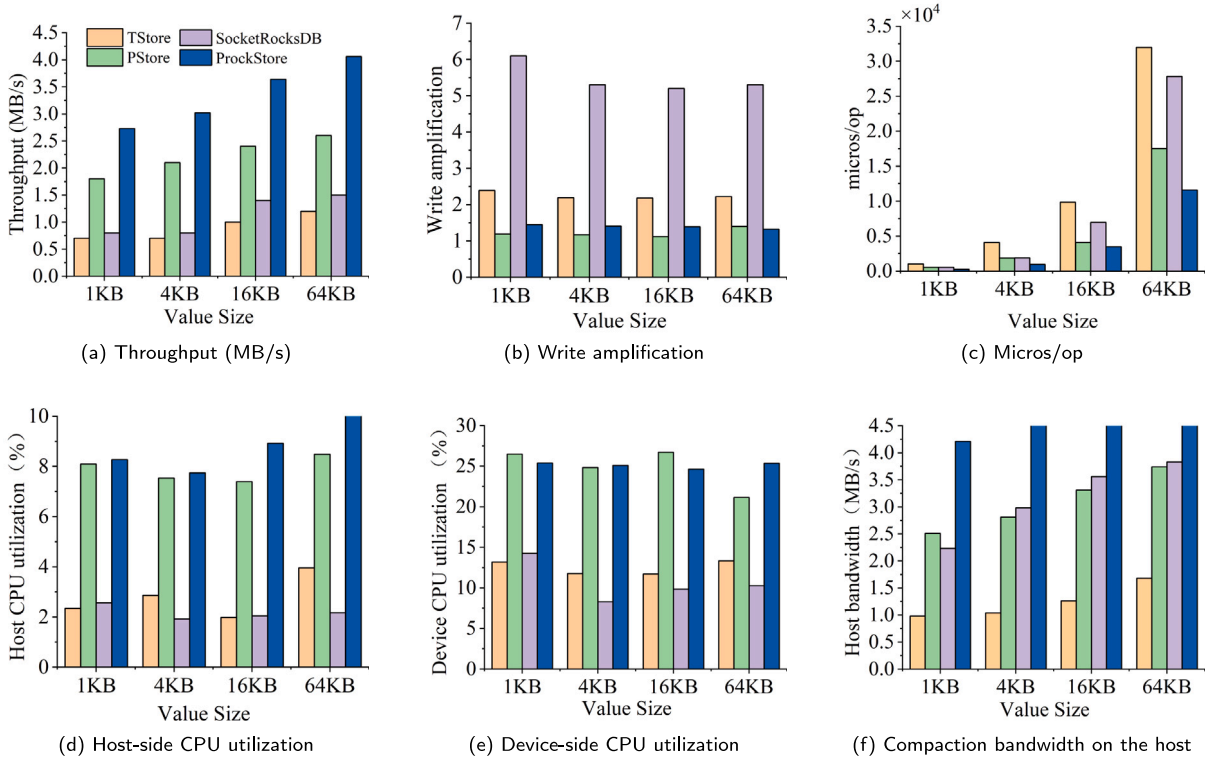

Fig. 11. The results of TStore, PStore, SocketRocksDB, and ProckStore under Fillrandom DB_Bench with 40-GB data volume and various value sizes.

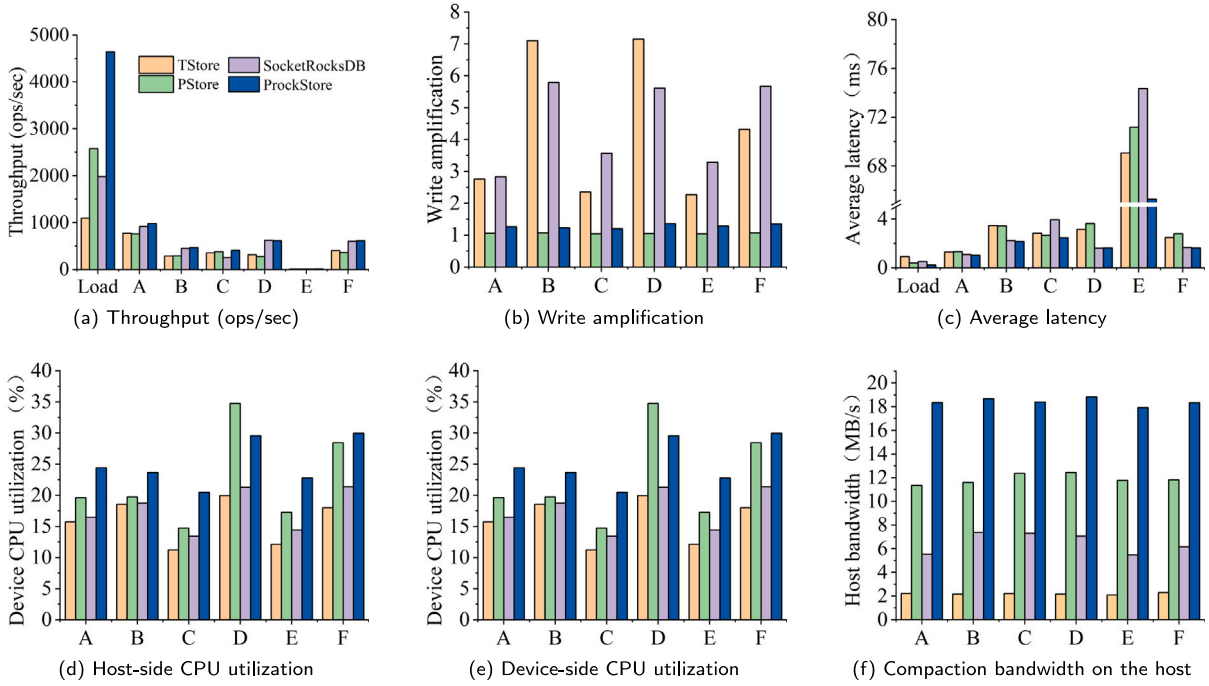

Fig. 12. The results of TStore, PStore, SocketRocksDB, and ProckStore under YCSB-C with load 10 GB and run 10 GB data volume.

5.3.1. Case 1: Load 10 GB and run 10 GB

Load. In YCSB-C, the workload Load is write-intensive, resulting in frequent compaction. ProckStore optimizes compaction under various workloads (Fig. 12). Its throughput outperforms SocketRocksDB by a factor of 2.3×. TStore benefits from time-aware dynamic task scheduling, which reduces the performance gap compared to ProckStore. PStore's asynchronous compaction improves performance, ProckStore's multi-threaded execution further enhances the performance of the asynchronous compaction strategy. Consequently, ProckStore's throughput is 4.24× and 1.80× higher than that of TStore and PStore, respectively. During the Run phase, ProckStore's collaborative mode improves performance under write-intensive workloads. Workloads A and F exhibit the highest write ratios at 29.0%. Under workload A, ProckStore's throughput is 28.2% and 29.0% higher than that of TStore and PStore, respectively. Under workload F, ProckStore's throughput surpasses TStore and PStore by 36.4% and 71.3%, respectively. However, when the write percentage is low, ProckStore's throughput shows minimal variation compared to other KV stores. Additionally, under read-intensive workloads, ProckStore achieves the maximum throughput improvement of 60.7%, 59.8%, 122.4%, and 9.2% under workloads B, C, D, and E, respectively. In contrast, TStore's read performance suffers due to the excessive number of SSTables, which increases the query operation overhead.

Throughput and Latency. Throughput and latency are critical metrics for KV stores. As KV stores are widely deployed in real-world applications, these metrics significantly impact response time. ProckStore maintains its performance advantage in the Load phase when the data size increases from 10 GB to 20 GB under a workload with the same amount of data. Its throughput is 4.24× that of TStore, see Fig. 12(a). Compared with SocketRocksDB and PStore, ProckStore's throughput improves by 2.33× and 1.8×, respectively, under the same workload. The advantage of ProckStore becomes even more pronounced under workloads A and F which involves a higher percentage of writes. Latency results further demonstrate the flexibility of ProckStore's scheduling method. Under workloads D and F, ProckStore has 55.1% and 42.5% lower latency than PStore (see Fig. 12(c)). Compared with TStore and PStore, ProckStore has 20.8% and 22.1% lower latency under workload A, respectively, see Fig. 12(c). Under read-intensive workload C, ProckStore's average latency is 13.4% and 37.5% lower than that of TStore and SocketRocksDB, respectively. ProckStore exhibits similar trends under workloads B and D. The average latency of ProckStore under write-intensive workloads is 5.72% and 8.45% lower than that of TStore and PStore under workload E, respectively. Moreover, the throughput of ProckStore is not lower than other KV stores.

Write Amplification (WA). ProckStore achieves lower WA than both TStore and SocketRocksDB (see Fig. 12(b)). WA is reduced by approximately 1.2 compared to SocketRocksDB. ProckStore's host-side multi-threaded method further decreases WA by an average of 62.3% compared to TStore. The minimum WA of ProckStore is 1.20 under workload C. WA in ProckStore is influenced by the write-ahead log and host-side compaction, its compaction frequency is higher, resulting in greater WA than PStore. Under workloads C and D, WA is 1.20 and 1.36, respectively. However, ProckStore's triple-level filter compaction mechanism mitigates WA compared to SocketRocksDB.

CPU Utilization and Compaction Bandwidth. Figs. 12(d) and 12(e) show the host-side CPU utilization of the KV store. Notably, TStore runs on a single thread. The bandwidth limits the data transfer between the host and the device. Overall, CPU utilization patterns for SocketRocksDB and ProckStore are similar on both sides, which can be attributed to the reduction in total processing time, accompanied by a reduction in compaction time. Fig. 12(f) shows compaction bandwidth in the Load and Run phases. ProckStore achieves the highest bandwidth on the host. Its bandwidth is 8.64× and 3.32× improvement than TStore and SocketRocksDB, respectively, under workload A. This improvement increases to 7.97× and 2.99× under workload F. Nevertheless, ProckStore improves the bandwidth by exploiting multi-threaded parallelism. The average bandwidth of ProckStore is 56.8% higher than PStore due to its efficient task scheduling, which leverages the computational capabilities of both the host and the device.

5.3.2. Case 2: Load 20 GB and run 20 GB

In the Load phase, the throughput of ProckStore surpasses SocketRocksDB and TStore by 3.44× and 3.73×, respectively, see Fig. 13(a). Although the asynchronous approach of PStore enhances performance, the multi-threaded method of ProckStore integrates with the asynchronous compaction mechanism. Consequently, the throughput of

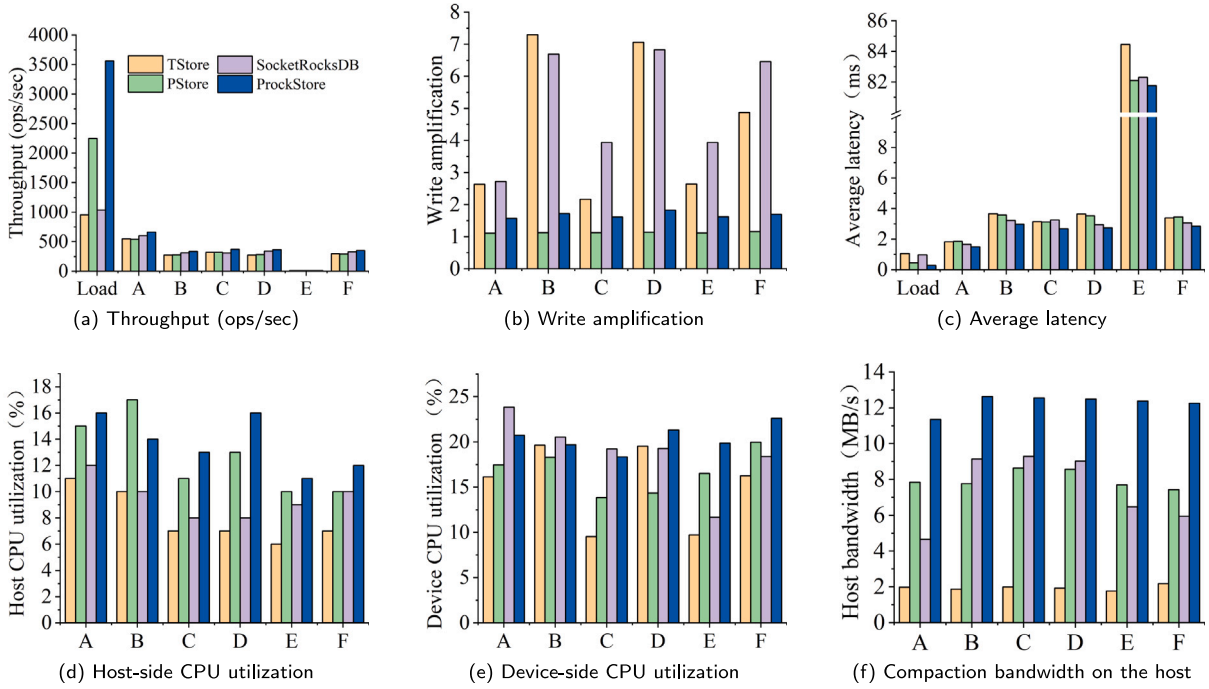

Fig. 13. The results of TStore, PStore, SocketRocksDB, and ProckStore under YCSB-C with load 20 GB and run 20-GB data volume.

ProckStore reaches 1.59× of PStore, respectively. In the Run phase, the multi-threaded asynchronous mode improves the performance of ProckStore under write-intensive workloads A and F, where half of the operations are writes. Specifically, under workload A, ProckStore's throughput exceeds that of TStore and PStore by 21.0% and 23.1%, respectively. Similarly, under workload F, ProckStore achieves 19.1% and 21.3% higher throughput than TStore and PStore, respectively.

Workloads A and F involve large-sized data volumes. Under these workloads, the data volume increased from 10-GB to 20-GB, and the throughput of ProckStore decreased by 32.2% and 42.7%, respectively. In addition, the throughput of ProckStore is optimized under read-intensive workloads. In ProckStore, we focus on optimizing compaction, and the read performance improvement is small. For read-intensive workloads, ProckStore achieves 20.4%, 16.3%, and 28.5% improvement under B, C, and D, compared with PStore, respectively. For workloads with small-sized data volumes, ProckStore decreases by 27.8%, 8.4%, and 40.6% under workloads B, C, and D, respectively.

Throughput and Latency. With a data size of 20 GB, ProckStore maintains its performance advantage in the Load phase. Compared with SocketRocksDB and PStore, ProckStore's throughput is improved by 3.44× and 1.58×, respectively, in the Load phase. Both average latency and throughput of ProckStore have the highest performance in Figs. 13(a) and 13(c). Under read-intensive workloads such as B and C, ProckStore outperforms SocketRocksDB by about 9.1% and 21.4%, respectively. This improvement is attributed to the triple-filtering compaction, which reduces execution time in the run phase, thereby increasing throughput.

As shown in Fig. 13(c), under write-intensive workload A, the average latency of ProckStore is 17.6% and 18.9% lower than TStore and PStore, respectively. ProckStore has a similar trend under workload F. In addition, ProckStore's latency also reduces by 16.9%, 14.1%, and 22.2% under read-intensive workloads B, C, and D, compared with SocketRocksDB, respectively. However, compared with 10 GB data volume, the latency increases due to the additional compaction operations and the associated lookup costs.

CPU Utilization and Compaction Bandwidth. When the data volume increases from 10 GB to 20 GB, Figs. 13(d) and 13(e) illustrate the changes in CPU utilization for ProckStore under various workloads. ProckStore increases host- and device-side CPU utilization by 18.1% and 32.6%, respectively, compared with PStore under workload C. Under mixed read–write workloads, such as A and F, ProckStore increases host-side CPU utilization by 6.7% and 20.0% and device-side CPU utilization by 12.2% and 13.2%, respectively. Fig. 13(f) shows the compaction bandwidth of the Run phase. ProckStore achieves the highest bandwidth under all workloads. In workload C, ProckStore's bandwidth is 45.4% and 35.1% higher than that of PStore and SocketRocksDB, respectively. This improvement is attributed to ProckStore's utilization of multi-threaded parallelism. However, with large-size data volumes, ProckStore's bandwidth decreases by 17.7% compared to a 10-GB data volume. Under workload D, ProckStore's average bandwidth is 6.47× and 1.38× higher than TStore and SocketRocksDB, respectively. In comparison to a 10-GB data volume, the CPU utilization decreases by 38.1%, 32.3%, 17.7%, 33.7%, 30.9%, and 33.1% under workloads A, B, C, D, E, and F, respectively.

5.3.3. Tail latency

We analyzed the tail latency of ProckStore, including P90, P99, and P999 latencies, and compared it with TStore, SocketRocksDB, and PStore under workloads of different data volumes (10 GB, 20 GB) and a 1-KB value size. The experimental results are shown in Figs. 14 and 15.

The results demonstrate that ProckStore outperforms other key-value stores, exhibiting lower tail latency. SocketRocksDB's P90 and P99 tail latencies are notably lower than those of other KV stores, due to its multi-version management mechanism in RocksDB. ProckStore's P90 and P99 tail latencies are lower than those of other KV stores thanks to its asynchronous allocation method, which reduces tail latency. Under a 10 GB workload, the most significant reduction in latency occurs when ProckStore lowers P90 latency by 94.07% and 93.89% compared to TStore and PStore under workload E. This improvement is attributed to ProckStore's superior range query performance, while TStore and PStore are not optimized for range queries. Similarly, ProckStore achieves the lowest P99 latency. Fig. 14(b) shows that ProckStore achieves the most significant P99 latency reduction under workload E, lowering P99 by 79.4% and 79.2% compared to TStore and PStore, respectively. It also shows substantial improvements under workload B,

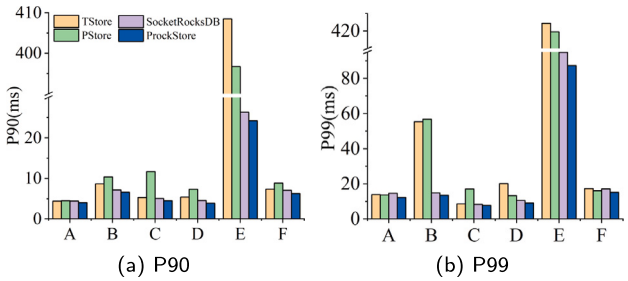

Fig. 14. The tail latency of ProckStore under YCSB-C with load 10 GB and run 10 GB data volume.

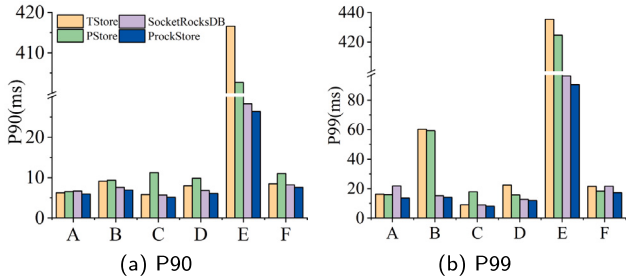

Fig. 15. The tail latency of ProckStore under YCSB-C with load 20 GB and run 20 GB data volume.

with ProckStore reducing P99 by 75.6% and 76.2% compared to TStore and PStore.

In Fig. 15, the differences in tail latency become more pronounced under a 20 GB workload. ProckStore reduces P90 latency by 9.32% and 31.06% under workloads A and F, respectively, compared to PStore. Under workload E, ProckStore reduces P90 latency by 93.46% and 93.68% compared to PStore and TStore, respectively. ProckStore's four-level priority scheduling mechanism prevents low-priority requests from blocking high-priority writes, reducing extreme write latency often blocked by flush or compaction in TStore and SocketRocksDB. Similarly, ProckStore reduces P99 tail latency under workloads A and F by 28.9% and 17.9%, respectively, compared to SocketRocksDB and TStore. Under workload C, ProckStore reduces P99 tail latency by 54.9% and 9.45% compared to PStore and SocketRocksDB, respectively. Under workload D, ProckStore reduces P99 tail latency by 23.5% and 6.0% compared to the same alternative KV stores. ProckStore performs best under workload E, reducing P99 tail latency by 79.22%, 78.69%, and 6.23% compared to TStore, PStore, and SocketRocksDB, respectively.

The FIFO scheduling used by traditional KV stores like SocketRocksDB can cause high-priority requests to be blocked, leading to increased tail latency. In contrast, ProckStore's multi-level queue scheduling mechanism enables compaction tasks to be executed in priority order, with high-priority compaction tasks executed first, thereby reducing tail latency.

6. Extended experiment

In this section, we study the impact of multi-threaded and the number of subtasks on ProckStore performance. The results demonstrate the effectiveness of ProckStore under multi-threaded and verify its performance under multiple subtask numbers. The environment of the extended experiment is the same as the experimental configuration in Section 4.

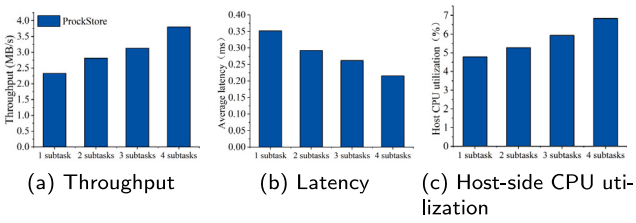

Fig. 16. Write performance of ProckStore under DB_Bench with different numbers of subtasks.

6.1. Impact of number of subtasks

To validate the fourth-level prioritization, we conducted experiments to evaluate the impact of various subtasks on the write performance of ProckStore. The extended experiments replicate the configuration from Section 4. We configured DB_Bench with a 10 GB dataset and a 1-KB value. Specifically, we examine the impact of the number of subtasks on the fourth-level prioritization in ProckStore by configuring four types of subtasks on the host. The experimental results are shown in Fig. 16.

As shown in Fig. 16(a), the throughput of ProckStore increases significantly with the number of subtasks. The throughput is 2.33 MB/s, 2.81 MB/s, and 3.13 MB/s for one, two, and three subtasks, respectively, and subsequently stabilizes. With four subtasks, ProckStore achieves a peak throughput of 3.8 MB/s. The average latency shows a similar trend, where ProckStore achieves the lowest latency (0.22 ms) with four subtasks, showing a 17.1% improvement from three to four subtasks. The host-side CPU utilization also reflects ProckStore's performance with different numbers of subtasks, as multi-core CPUs enable parallel execution of multiple threads.

As shown in Fig. 16(c), CPU utilization increases with the number of subtasks, allowing the CPU to utilize its computational resources fully. CPU utilization is 4.78% (the lowest) with one subtask, improving by 10.3% with two subtasks. The highest CPU utilization (6.84%) occurs with four subtasks. However, as the number of subtasks increases, the performance improvements in CPU utilization, throughput, and average latency become less pronounced. This is because while parallel execution of multiple threads reduces compaction execution time, the overhead from thread creation and synchronization increases. As the number of threads grows, this additional CPU overhead impacts ProckStore's CPU utilization.

6.2. Impact of number of threads

In Section 2.2, we studied the performance of PStore with different numbers of threads. For the multi-threaded comparison experiment of ProckStore, we extended the analysis by comparing its throughput with that of PStore under varying thread counts. The experimental results are shown in Fig. 17.

Fig. 17(a) shows the throughput of ProckStore and PStore under workloads with 4 KB values and a 10 GB data volume. As the number of threads increases, the throughput of PStore does not increase exponentially, and its performance is poor during multi-threaded writes. Specifically, the throughput increases by only 1.58% when the number of threads rises from 8 to 12. In contrast, the throughput of ProckStore increases significantly with the number of threads. At 12 threads, the throughput reaches 7.86 MB/s, which is 10.9% higher than that at 8 threads. ProckStore's throughput is 144.1% higher than PStore's, as its multi-threaded execution efficiently processes the large data volume written by multiple threads, avoiding the computational limitations of single-thread execution in PStore.

As shown in Fig. 17(b), the average latency of PStore decreases with the increase in threads under a 10 GB data volume workload.

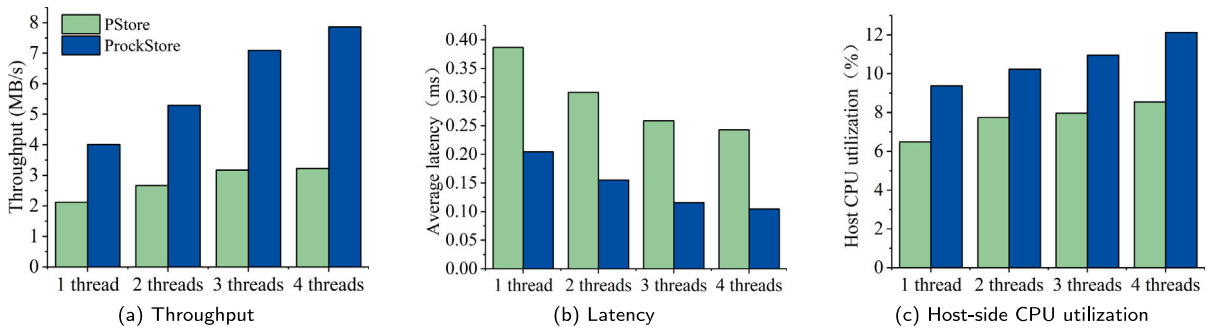

Fig. 17. Write performance of ProckStore under DB_Bench with different number of threads.

However, the decrease in latency is more significant when the number of threads is low, such as from 1 to 4 threads, where the latency drops by 27.8%. For ProckStore, as the number of threads increases, performance improves steadily. At 4 threads, the average latency of ProckStore is 0.154 ms, and at 12 threads, it reduces to 0.104 ms with a 32.3% reduction. Fig. 17(c) shows that when the number of threads reaches 12, the host-side CPU utilization for PStore and ProckStore is highest, at 8.74% and 11.62%, respectively, with ProckStore showing a 41.92% increase over PStore. Additionally, the CPU utilization for both systems increases by 7.28% and 10.7%, respectively, when the number of threads increases from 8 to 12. As the number of threads decreases, CPU utilization also drops. For 1 thread, the CPU utilization is at its lowest – 6.48% for PStore and 9.37% for ProckStore.

7. Related work

LSM-tree has become a popular data structure in key-value storage systems, offering an alternative to traditional structures by efficiently handling write-intensive workloads and large-scale datasets. Although KV stores manage data through compaction operations, these processes come at the cost of performance. Consequently, several studies have sought to mitigate the performance impact of compaction in KV stores.

LSM-tree structure. PebblesDB [13] introduces the FLSM data structure, which alleviates the limitations of non-overlapping key ranges within a level, thereby delaying the compaction process and reducing WA. WiscKey [14] separates keys and values to minimize WA during compaction but increases garbage collection overhead. To address this issue, HashKV [15] employs hash partitioning and a hot/cold partitioning strategy, while DiffKV [16] separates keys based on the size of key-value pairs to balance performance. FenceKV [17] enhances HashKV by incorporating a fence-value-based partitioning strategy and key-range-based garbage collection, optimizing range queries. FGKV [18] and Spooky [19] reduce WA by adjusting the data granularity in compaction. FGKV introduces a fine-grained compaction mechanism based on the LSpM-tree structure, minimizing redundant writes of irrelevant data. Spooky partitions the data at the largest level into equal-sized files and partitions the smaller levels according to file boundaries for fine-grained compaction.

For compaction strategies, TRIAD [20] improves LSM-tree performance by optimizing logs, memory, and storage. Work [21,22] optimize the traditional top-level-driven compaction of LSM-trees by shifting to a low-level-driven approach, decomposing large compaction tasks into smaller ones to reduce granularity. WipDB [23] utilizes a bucket-sort-like algorithm that minimizes merge operations by writing KV pairs in an approximately sorted list. Although these studies enhance compaction efficiency, they primarily focus on a single device and fail to address the competition for CPU and I/O resources between foreground requests and background tasks. In contrast, NDP devices expand computational resources to process tasks internally, reducing data transfer and resource contention.

Storage architecture. ListDB [24] employs a skip-list as the core data structure at all levels within non-volatile memory (NVM) or persistent memory (PM) to mitigate the WA problem by leveraging byte-addressable in-place merge ordering. This approach reduces the gap between DRAM and NVM write latency and addresses the write stall issue. HiKV [25] utilizes the benefits of hash and B+Tree indexes to design the KV store on hybrid DRAM-NVM storage systems, where hash indexes in NVM are used to enhance indexing performance. In a hybrid NVM-SSD system, WaLSM [26] tackles the WA problem through virtual partitioning, dividing the key space during compaction. Additionally, a reinforcement-learning method is applied to balance the merging strategy of different partitions under various workloads, optimizing read and write performance. TrieKV [27] integrates DRAM, PM, and disk into a unified storage system, utilizing a tri-structured index for all KV pairs in memory, enabling dynamic determination of KV pair locations across storage hierarchies and persistence requirements. Moreover, ROCKSMASH [28] utilizes local storage for frequently accessed data and metadata, while cloud storage is employed for less frequently accessed data.

Computing architecture. Heterogeneous computing [29] (e.g., GPUs, DPUs, and FPGAs) alleviates the computational burden on the CPU. Sun et al. [30] propose an accelerated solution for key-value stores by offloading the compaction task to an FPGA. Similarly, the FPGA-accelerated KV store [31] offloads the compaction task to the FPGA, minimizing competition for CPU resources and accelerating compaction while reducing CPU bottlenecks. LUDA [32] employs GPUs to process SSTables using a co-ordering mechanism that minimizes data movement, thereby reducing CPU pressure. gLSM [33] separates keys and values to minimize data transfer between the CPU and GPU, thereby accelerating compaction. dCompaction [34] leverages DPUs to accelerate the compaction and decompaction of SSTables, offloading compaction tasks to the DPU according to a hierarchical structure, relieving CPU overload. Despite these advances, heterogeneous computing still requires data transfer from host-side memory to the computing units, which can impact overall system performance.

Near-data processing (NDP), which offloads computational tasks from the CPU to the data location, is an emerging computing paradigm. Previous studies [35] investigated storage computing and propose frameworks for storage- and memory-level processing. Biscuit [36] introduces a generalized framework for NDP. RFNS [37] examines the advantages of reconfigurable NDP-driven servers based on ARM and FPGA architectures for data- and compute-intensive applications. λ-IO [38] designs a unified computational storage stack to manage storage and computing resources through interfaces, runtime systems, and scheduling. HuFu [39] is an I/O scheduling architecture for computable SSDs that allows the system to manage background I/O tasks, offload computational tasks to SSDs, and exploit the parallelism and idle time of flash memory for improved task scheduling. Li et al. [40] addresses the resource contention problem between user I/O and NDP requests, using the critical path to maximize the parallelism of multiple requests, thereby improving the performance of hybrid NDP-user I/O workflows.

ABNDP [41] leverages a novel hardware-software collaborative optimization approach to solve the challenges of remote data access and computational load balancing without requiring trade-offs.

In addition, hosts and NDP devices employ distinct task scheduling policies to collaborate on compaction tasks [9,10,42]. The nKV [43] defines data formats and layouts for computable storage devices and designs both hardware and software architectures to optimize data placement and computation.

KV-CSD [44] builds NDP architectures using NVMe SSDs and system-on-chip designs to reduce data movement during queries by offloading tasks. Research such as OI-RAID [45] introduces an additional fault tolerance mechanism by adding an extra level on top of the RAID levels, enabling fast recovery and enhanced reliability. KVRAID [46] utilizes logical-to-physical key conversion to pack similar-sized KV pairs into a single physical object, thereby reducing WA, and applies off-site update techniques to mitigate I/O amplification. Distributed storage systems, such as EdgeKV, have also been explored [47]. A sharding strategy is used to distribute data across multiple edge nodes, while consistent hashing ensures balanced data distribution and high availability. ER-KV [48] integrates a hybrid fault-tolerant design combining erasure coding and PBR, providing fault tolerance to ensure system reliability and high availability. Additionally, Song et al. [49] coupled each SSD with a dedicated NDP engine in an NDP server to fully leverage the data transfer bandwidth of SSD arrays. MStore [50] extends an NDP device to multiple devices, utilizing them to perform compaction tasks.

Although NDP devices can handle host-side computational tasks, their resources remain limited. Consequently, it is critical to optimize the use of these resources on the NDP device. The multi-threaded asynchronous method in ProckStore addresses this challenge by fully utilizing computation on both the host and device sides, avoiding resource wastage while ensuring sufficient computational capacity on the NDP device.

8. Conclusions

In this paper, we present ProckStore, an NDP-empowered KV store, to improve compaction performance for large-scale unstructured data storage. In ProckStore, the multi-threaded and asynchronous mechanism leverages computational resources within storage devices, reducing data movement and enhancing compaction efficiency. ProckStore optimally schedules compaction tasks across the host and NDP device by implementing a four-level priority scheduling mechanism. This separation of compaction stages provides parallel processing without interference, achieving efficient resource utilization. In addition, ProckStore uses key-value separation to reduce data transfer between the host and NDP device, minimizing transmission time. Experimental results unveil that ProckStore outperforms existing synchronous and single-threaded asynchronous NDP-empowered KV stores, achieving up to 4.2× higher throughput than the baseline KV store. ProckStore also reduces WA, compaction time, and CPU utilization.

CRediT authorship contribution statement

Hui Sun: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Chao Zhao:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Yinliang Yue:** Validation, Supervision, Software. **Xiao Qin:** Supervision, Resources, Methodology, Formal analysis, Data curation.

Declaration of competing interest

The authors declare that there is no conflict of interests regarding the publication of this article.

References

[1] Z. Zhang, Y. Sheng, T. Zhou, et al., H2o: Heavy-hitter oracle for efficient generative inference of large language models, in: Advances in Neural Information Processing Systems, vol. 36, 2024.

[2] H. Lin, Z. Wang, S. Qi, et al., Building a high-performance graph storage on top of tree-structured key-value stores, Big Data Min. Anal. 7 (1) (2023) 156–170.

[3] S. Pei, J. Yang, Q. Yang, REGISTOR: A platform for unstructured data processing inside SSD storage, ACM Trans. Storage (TOS) 15 (1) (2019) 1–24.

[4] IDC, IDC innovators: Privacy-preserving computation, 2023, [EB/OL]. (2023-09-20). https://www.idc.com/getdoc.jsp?containerId=prCHC51469323.

[5] P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, The log-structured merge-tree (LSM-tree), Acta Inform. 33 (4) (1996) 351–385.

[6] Google, Leveldb, 2025, https://leveldb.org/.

[7] Facebook, Rocksdb: a persistent key-value store for fast storage environments, 2016, http://rocksdb.org/.

[8] A. Acharya, M. Uysal, J. Saltz, Active disks: Programming model, algorithms and evaluation, Oper. Syst. Rev. 32 (5) (1998) 81–91.

[9] H. Sun, W. Liu, J. Huang, et al., Collaborative compaction optimization system using near-data processing for LSM-tree-based key-value stores, J. Parallel Distrib. Comput. 131 (2019) 29–43.

[10] H. Sun, W. Liu, Z. Qiao, et al., Dstore: A holistic key-value store exploring near-data processing and on-demand scheduling for compaction optimization, IEEE Access 6 (2018) 61233–61253.

[11] H. Sun, et al., Asynchronous compaction acceleration scheme for near-data processing-enabled LSM-tree-based KV stores, ACM Trans. Embed. Comput. Syst. 23 (6) (2024) 1–33.

[12] Isaac Kofi Nti, et al., A mini-review of machine learning in big data analytics: Applications, challenges, and prospects, Big Data Min. Anal. 5 (2) (2022) 81–97.

[13] P. Raju, R. Kadekodi, V. Chidambaram, et al., Pebblesdb: Building key-value stores using fragmented log-structured merge trees, in: Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 497–514.

[14] L. Lu, T.S. Pillai, H. Gopalakrishnan, et al., Wisckey: Separating keys from values in SSD-conscious storage, ACM Trans. Storage (TOS) 13 (1) (2017) 1–28.

[15] H. H. W. Chan, C. J. M. Liang, Y. Li, et al., HashKV: Enabling efficient updates in KV storage via hashing, in: 2018 USENIX Annual Technical Conference, USENIX ATC 18, 2018, pp. 1007–1019.

[16] Y. Li, Z. Liu, P. P. C. Lee, et al., Differentiated key-value storage management for balanced I/O performance, in: 2021 USENIX Annual Technical Conference, USENIX ATC 21, 2021, pp. 673–687.

[17] C. Tang, J. Wan, C. Xie, Fencekv: Enabling efficient range query for key-value separation, IEEE Trans. Parallel Distrib. Syst. 33 (12) (2022) 3375–3386.

[18] H. Sun, G. Chen, Y. Yue, et al., Improving LSM-tree based key-value stores with fine-grained compaction mechanism, IEEE Trans. Cloud Comput. (2023).

[19] N. Dayan, T. Weiss, S. Dashevsky, et al., Spooky: granulating LSM-tree compactions correctly, in: Proceedings of the VLDB Endowment, Vol. 15, (11) 2022, pp. 3071–3084.

[20] O. Balmau, D. Didona, R. Guerraoui, et al., TRIAD: Creating synergies between memory, disk and log in log structured key-value stores, in: 2017 USENIX Annual Technical Conference, USENIX ATC 17, 2017, pp. 363–375.

[21] Y. Chai, Y. Chai, X. Wang, et al., LDC: a lower-level driven compaction method to optimize SSD-oriented key-value stores, in: 2019 IEEE 35th International Conference on Data Engineering, ICDE, 2019, pp. 722–733.

[22] Y. Chai, Y. Chai, X. Wang, et al., Adaptive lower-level driven compaction to optimize LSM-tree key-value stores, IEEE Trans. Knowl. Data Eng. 34 (6) (2020) 2595–2609.

[23] X. Zhao, S. Jiang, X. Wu, WipDB: A write-in-place key-value store that mimics bucket sort, in: 2021 IEEE 37th International Conference on Data Engineering, ICDE, 2021, pp. 1404–1415.

[24] W. Kim, C. Park, D. Kim, et al., Listdb: Union of write-ahead logs and persistent SkipLists for incremental checkpointing on persistent memory, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, pp. 161–177.

[25] F. Xia, D. Jiang, J. Xiong, et al., HiKV: a hybrid index key-value store for DRAM-NVM memory systems, in: 2017 USENIX Annual Technical Conference, USENIX ATC 17, 2017, pp. 349–362.

[26] L. Chen, R. Chen, C. Yang, et al., Workload-aware log-structured merge key-value store for NVM-SSD hybrid storage, in: 2023 IEEE 39th International Conference on Data Engineering, ICDE, 2023, pp. 2207–2219.

[27] H. Sun, et al., TrieKV: A high-performance key-value store design with memory as its first-class citizen, IEEE Trans. Parallel Distrib. Syst. (2024).

[28] P. Xu, N. Zhao, J. Wan, et al., Building a fast and efficient LSM-tree store by integrating local storage with cloud storage, ACM Trans. Archit. Code Optim. (TACO) 19 (3) (2022) 1–26.

[29] Hao Zhou, Yuanhui Chen, Lixiao Cui, Gang Wang, Xiaoguang Liu, A GPU-accelerated compaction strategy for LSM-based key-value store system, in: The 38th International Conference on Massive Storage Systems and Technology, 2024, pp. 1–11.

[30] X. Sun, J. Yu, Z. Zhou, et al., Fpga-based compaction engine for accelerating lsm-tree key-value stores, in: 2020 IEEE 36th International Conference on Data Engineering, ICDE, 2020, pp. 1261–1272.
[31] T. Zhang, J. Wang, X. Cheng, et al., FPGA-accelerated compactions for LSM-based key-value store, in: 18th USENIX Conference on File and Storage Technologies, FAST 20, 2020, pp. 225–237.
[32] P. Xu, J. Wan, P. Huang, et al., LUDA: Boost LSM key value store compactions with GPUs, 2020, arXiv preprint arXiv:2004.03054.
[33] H. Sun, J. Xu, X. Jiang, et al., gLSM: Using GPGPU to accelerate compactions in LSM-tree-based key-value stores, ACM Trans. Storage (2023).
[34] C. Ding, J. Zhou, J. Wan, et al., Dcomp: Efficient offload of LSM-tree compaction with data processing units, in: Proceedings of the 52nd International Conference on Parallel Processing, 2023, pp. 233–243.
[35] E. Riedel, G. Gibson, C. Faloutsos, Active storage for large-scale data mining and multimedia applications, in: Proceedings of 24th Conference on Very Large Databases, 1998, pp. 62–73.
[36] B. Gu, A.S. Yoon, D.H. Bae, et al., Biscuit: A framework for near-data processing of big data workloads, ACM SIGARCH Comput. Archit. News 44 (3) (2016) 153–165.
[37] X. Song, T. Xie, S. Fischer, Two reconfigurable NDP servers: Understanding the impact of near-data processing on data center applications, ACM Trans. Storage (TOS) 17 (4) (2021) 1–27.
[38] Z. Yang, Y. Lu, X. Liao, et al., λ-IO: A unified IO stack for computational storage, in: 21st USENIX Conference on File and Storage Technologies, FAST 23, 2023, pp. 347–362.
[39] Y. Wang, Y. Zhou, F. Wu, et al., Holistic and opportunistic scheduling of background I/Os in flash-based SSDs, IEEE Trans. Comput. (2023).
[40] J. Li, X. Chen, D. Liu, et al., Horae: A hybrid I/O request scheduling technique for near-data processing-based SSD, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (11) (2022) 3803–3813.
[41] B. Tian, Q. Chen, M. Gao, ABNDP: Co-optimizing data access and load balance in near-data processing, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 3, 2023, pp. 3–17.
[42] H. Sun, W. Liu, J. Huang, et al., Near-data processing-enabled and time-aware compaction optimization for LSM-tree-based key-value stores, in: Proceedings of the 48th International Conference on Parallel Processing, 2019, pp. 1–11.
[43] T. Vincon, A. Bernhardt, I. Petrov, et al., nKV: near-data processing with KV-stores on native computational storage, in: Proceedings of the 16th International Workshop on Data Management on New Hardware, 2020, pp. 1–11.
[44] I. Park, Q. Zheng, D. Manno, et al., KV-CSD: A hardware-accelerated key-value store for data-intensive applications, in: 2023 IEEE International Conference on Cluster Computing, CLUSTER, 2023, pp. 132–144.
[45] N. Wang, Y. Xu, Y. Li, et al., OI-RAID: a two-layer RAID architecture towards fast recovery and high reliability, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, 2016, pp. 61–72.
[46] M. Qin, A.L.N. Reddy, P. V. Gratz, et al., KVRAID: high performance, write efficient, update friendly erasure coding scheme for KV-SSDs, in: Proceedings of the 14th ACM International Conference on Systems and Storage, 2021, pp. 1–12.
[47] K. Sonbol, Ö. Özkasap, I. Al-Oqily, et al., EdgeKV: Decentralized, scalable, and consistent storage for the edge, J. Parallel Distrib. Comput. 144 (2020) 28–40.
[48] Y. Geng, J. Luo, G. Wang, et al., Er-kv: High performance hybrid fault-tolerant key-value store, in: 2021 IEEE 23rd International Conference on High Performance Computing & Communications; 7th International Conference on Data Science & Systems; 19th International Conference on Smart City; 7th International Conference on Dependability in Sensor, Cloud & Big Data Systems & Application, HPCC/DSS/SmartCity/DependSys, 2021, pp. 179–188.
[49] X. Song, T. Xie, S. Fischer, A near-data processing server architecture and its impact on data center applications, in: High Performance Computing: 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20, 2019, Proceedings 34, Springer International Publishing, 2019, pp. 81–98.
[50] H. Sun, Q. Wang, Y.L. Yue, et al., A storage computing architecture with multiple NDP devices for accelerating compaction performance in LSM-tree based KV stores, J. Syst. Archit. 130 (2022) 102681.