iKnowFirst: An Efficient DPU-assisted Compaction for LSM-Tree-based Key-Value Stores

Jiahong Chen, Shengzhe Wang, Zhihao Zhang, Suzhen Wu✉ and Bo Mao
Xiamen University, Xiamen, Fujian, China
{chenjiahong, wangshengzhe, zhihaoz}@stu.xmu.edu.cn
{suzhen, maobo}@xmu.edu.cn

***Abstract*—In scenarios with write-intensive workloads, LSM-tree-based key-value stores, such as RocksDB, suffer from compaction-induced performance degradation. RocksDB provides configurable compaction options to mitigate the severe read/write amplification problems associated with compaction. The advent of the Data Processing Unit (DPU) allows us to better utilize the configurable options of RocksDB to guide the key-value store system in choosing a suitable compaction strategy with prior knowledge of the workload characteristics. This paper proposes iKnowFirst, an efficient DPU-assisted key-value store. iKnowFirst (1) sets a data buffer on the DPU and separates hot-cold data to relieve the pressure of subsequent LSM-tree compaction, (2) senses the characteristics of the workloads in advance, and dynamically guides RocksDB to choose different compaction modes or enable/disable compaction when the workloads change, to cope with the scenario of write outbreak, and (3) implements an auto-selecting interface for compaction strategies selection. Our prototype implementation and experimental results show that iKnowFirst achieves $3.2\times$ improvement compared to the original RocksDB on write-intensive and highly skewed workloads while showing acceptable performance under read-intensive workloads.**

***Index Terms*—Key-Value Store, RocksDB, Data Processing Unit, Compaction**

I. Introduction

In today's data centers, Log-Structured-Merge-tree (LSM-tree) based key-value stores [7], [9], [22], [23], [28], [29] have been widely applied in NoSQL systems, such as Google LevelDB [1], Facebook RocksDB [4], Alibaba X-Engine [13], and TiDB [12]. Unlike RDBMS (Relational Database Management Systems), LSM-tree-based key-value stores show better performance under write-intensive workloads because it converts random writes to sequential writes, significantly improving the write throughput. However, this comes at the cost of storing redundant data. LSM-tree uses background compaction threads to sort and merge stale data, but frequent compaction can cause a lot of subsequent read/write amplification or even system blocking in writer-intensive scenarios. Therefore, it is crucial to choose the appropriate compaction strategies in different scenarios. RocksDB provides a series of optional configurations that allow users to select different compaction strategies or enable/disable compaction in different scenarios, which can help improve system performance.

Recently studies have shown that real-world workloads are often highly skewed [5], [6], [27]. Therefore, under read/write-intensive and highly skewed workloads, it is important to separate hot-cold data in addition to weighing various compaction strategies. Some existing key-value stores address the hotspot issue in real-world workloads by separating hot-cold data to provide fast access and high performance [8], [15], [19], [26]. Separating hot-cold data also helps reduce the overhead of compaction, which is similar to garbage collection (GC).

The advent of the DPU allows us to better combine the choice of compaction strategies with the hot-cold data separation. DPU is a newly developed category of dedicated processors. It is the third most important computing chip in data center scenarios after CPU and GPU. It provides a computing engine for high-bandwidth, low-latency, and data-intensive computing scenarios. DPU integrates SmartNICs and mainly deals with network and Input/Output (I/O) data. As the first station of I/O, DPU can analyze the workloads before the KV requests arrive at the host. This gives us the opportunity to identify the characteristics of the workload in advance and guide the RocksDB on the host to choose different compaction strategies under different circumstances. At the same time, there is an opportunity to identify hot-cold data in advance, reducing the pressure of subsequent compaction. In addition, existing researches [18], [25], [30] attempt to offload critical networking, storage, and security tasks from the host CPU to DPU/SmartNICs in order to reduce CPU overhead and achieve performance breakthroughs. Similarly, the task of separating hot-cold data can be transferred from the CPU to DPU to reduce the consumption of CPU resources.

Based on the above analysis, this paper proposes iKnowFirst, an efficient key-value store assisted by the data processing unit. iKnowFirst utilizes DPU to assist the key-value storage system. It mainly consists of the following three techniques. (1) iKnowFirst sets up a data buffer on the DPU, which is actually a set of hot hash tables and cold hash tables, to realize the separation of hot-cold data, keeping hot data on the DPU side. (2) iKnowFirst configures a workload-aware component on the DPU to analyze the workloads' characteristics, guiding the RocksDB on the host to select a proper compaction strategy to achieve a breakthrough in system performance. (3) iKnowFirst implements an auto-selecting interface on the DPU to control the compaction strategies selection of RocksDB on the host. Generally speaking, iKnowFirst has two meanings. One is knowing the workload's characteristics first, and the other is knowing the hot-cold data first.

We conduct a series of experiments to evaluate iKnowFirst with a variety of benchmarks. The experimental results show

DOI 10.1109/ASAP57973.2023.00022

that iKnowFirst achieves 3.2× improvement compared to the original RocksDB on write-intensive and highly skewed workloads. Moreover, iKnowFirst did not suffer significant side effects in subsequent read-intensive workloads.

In summary, this paper makes the following contributions:

- We propose iKnowFirst, an efficient key-value storage system using a new DPU-Host framework. It takes advantage of DPU to efficiently combine hot-cold data separation with RocksDB's configurable compaction while reducing the CPU's overhead.
- We design and implement an auto-selecting interface on the DPU for compaction strategy selection.
- We conduct a series of experiments to show that iKnowFirst achieves high performance on write-intensive and highly skewed workloads. Moreover, our components also can be deployed to other systems with RocksDB and DPU.

The rest of this paper is organized as follows. Section II describes the background and motivation. Section III introduces the design and implementation of iKnowFirst. The experimental results are shown in Section IV. Section V discusses related work and Section VI concludes this paper.

II. Background and Motivation

In this section, we first discuss the read/write amplification due to LSM-tree compaction and the benefits of hot-cold data separation. Then we introduce the NVIDIA BlueField-2 DPU [3] and its functional features. Finally, we elaborate on our motivation for using DPU to assist the RocksDB auto-selecting compaction.

A. LSM-tree Compaction

LSM-tree-based key-value stores are widely used in production environments, such as LevelDB [1] and RocksDB [4]. They are known for high write performance, but they are sub-optimal for reads [20] and suffer from performance bottlenecks due to compaction. Existing studies [11], [21] have proposed a number of optimizations for compaction to alleviate performance bottlenecks. Besides, RocksDB provides configurable options for compaction, and users can dynamically select the compaction strategies or even temporarily disable compaction as needed. Fig. 1 shows the data organizations and compaction strategies of LSM-Tree. For example, when performing a data write operation, the key-value pairs will be first written to a MemTable in the host memory, and when a MemTable is full, it will be converted to an immutable MemTable. Finally, the key-value pairs in the immutable MemTable will be dumped to a persistent storage medium such as SSD. The data is organized as SSTables on SSD, and there is no overlap between SSTables at the same level (except Level-0). In order to bridge the performance gap between memory and disks, the key ranges in SSTables in Level-0 overlap with each other. The two main compaction strategies' processes are as follows.

Leveled compaction strategy: In leveled compaction, a threshold is set for the total size of SSTables at each level. Higher levels have larger thresholds. When the total amount

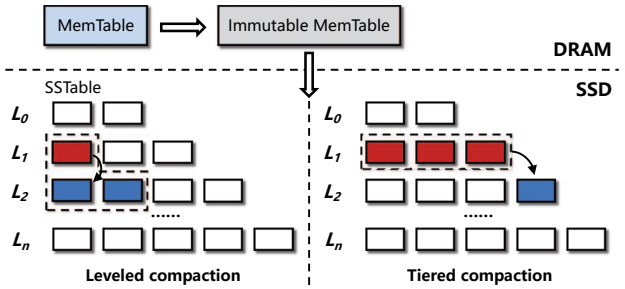


Fig. 1. The data organizations and compaction strategies of LSM-Tree.

of data at a level exceeds the threshold, one SSTable is selected and merged into one or more SSTables at a higher level. The selection of the SSTables involved in the high-level layer depends on the distribution of the data. All the SSTables in the high-level layer after the merger are in overall order, i.e., there is no overlap of data in the same layer. Leveled compaction is a strategy to minimize space amplification. However, compaction requires reading multiple SSTables and then writing multiple updated SSTables, resulting in greater write and read amplification. However, read amplification during data lookup is reduced by reducing the number of SSTables that need to be looked up.

Tiered compaction strategy: In tiered compaction, when the number of SSTables at a certain level reaches the threshold, multiple SSTables at the same level are merged into a new SSTable and placed in the high level. Tiered compaction focuses on controlling write amplification, reduces space amplification, and improves data query performance to a certain extent. However, there is still data overlap in the same layer of SSTables, and the data query performance is not as good as that of leveled compaction. It is suitable for scenarios where the frequency of data modification is low and the frequency of recently written data queries is high.

In summary, different compaction strategies have their own advantages and disadvantages and applicable scenarios. In some cases, incorrect compaction strategies may lead to performance bottlenecks and resource waste and even fast consume the SSD service life due to read, write, and space amplification. We should choose different compaction strategies in different situations according to the actual needs. In addition, since Level-0 is unordered, frequent compaction from Level-0 to Level-1 can cause write stalls when write bursts occur. Therefore, if necessary, we can temporarily disable the compaction and restore it after the writing pressure decreases.

B. Hot-Cold Data Separation

Recently years, the hotspot issue become the focus of attention in the real scene. Researches show that in a real-world highly-skewed workload, only a small portion of items will be frequently accessed [5], [6], [27]. There are many solutions to address cluster-wide hotspots, such as consistent hash, data migration, and front-end data caching [15], [19]. Besides, many schemes have been proposed to solve the

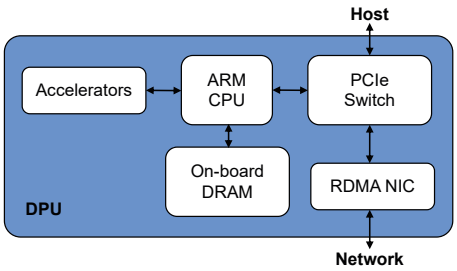


Fig. 2. System overview of the NVIDIA BlueField-2 DPU.

TABLE I
MAIN FEATURES OF THE NVIDIA BLUEFIELD-2 DPU

CPU	8 ARMv8 A72 cores (64-bit) @ 2.5 GHz, 1MB L2 cache per 2 cores, 6MB L3 cache
DRAM	16GB on-board DDR4-1600
Storage	eMMC flash memory
Network	Ethernet or InfiniBand: dual ports of 10/25/50/100 Gb/s, or a port of 200Gb/s
Accelerators	RegEX, SHA 256-bit hardware acceleration, True random number generator, and more
PCIe	Gen 4.0 × 16
OS	Ubuntu 20.04 (kernel 5.4.0-1007-bluefield)

single-node hotspot issue. Most of them utilize the hierarchical layout of computer architecture (e.g., disk, RAM, CPU cache), caching hot data in the low-latency storage medium. Low-level storage systems such as LevelDB [1] and RocksDB [4] use in-memory key-value stores (KVSes) to manage hot data. In KVSes, the hash index is the most popular structure because it provides fast lookup capabilities, especially when range queries are not required by upper-level applications.

It seems that using the hash index to cache hot data for RocksDB is an optimal solution. Is there a better solution or a new architecture that can further reduce CPU overhead and memory footprint while separating hot-cold data? We think the answer is coming with the advent of DPU.

C. Data Processing Unit

The data processing unit (DPU) is a new heterogeneous processor, which is mainly used to reduce CPU load and energy consumption. The NVIDIA BlueField-2 DPU [3], [17] can carry out a series of services such as network, storage and management, and provide each host with the acceleration ability of uninstallation and innovation. Fig. 2 illustrates the system overview of the NVIDIA BlueField-2 DPU.

Table I lists the main features of the NVIDIA BlueField-2 DPU. It integrates an NVIDIA Mellanox ConnectX-6Dx intelligent NIC and ARM processors to provide users with programmability. Besides, the NVIDIA DOCA SDK [2] integrates industry-standard APIs, making it easier for users to develop on the DPU. Users can move and secure data on it, reducing CPU overhead and latency, and leading to significant performance breakthroughs.

DPU plays an essential part in the I/O path. The cores of the DPU are located on the path of data transmission, and they can manipulate incoming and outgoing data. In addition, unnecessary PCIe access can be avoided by utilizing onboard DRAM to store data when processing data requests from remote networks. Moreover, the DPU can serve as the CPU offloading engine. Relevant studies show that a single BlueField-2 DPU can provide data center services equivalent to those provided by consuming 125 CPU cores. By offloading host CPU computing or storage tasks to the DPU, ones can significantly reduce the CPU overhead, freeing up CPU cores to run various other applications.

D. Our Motivation

In this article, we take advantage of the DPU's critical position on the data critical path, as well as the DPU's ability to perform storage and computing offloading. Specifically, we use DPU to assist RocksDB in auto-selecting compaction strategies. The KV requests from applications will arrive in the DPU first, and DPU can sense the workloads' characteristics and guide the RocksDB to select different compaction strategies. Meanwhile, we set up a data buffer on the DPU to realize the hot-cold data separation. Compared with the traditional architecture, we offload hot-cold data separation calculation tasks to the DPU, reducing the CPU overhead and memory occupation on the host.

III. DESIGN AND IMPLEMENTATION

In this section, we first present the architecture overview of the iKnowFirst framework. We then elaborate on the design of iKnowFirst in detail, including the hot-cold data separation and auto-selecting compaction. We finally describe the data operation of iKnowFirst.

A. Overview

Fig. 3 shows the architecture overview of iKnowFirst. iKnowFirst sets up a data buffer on the onboard DRAM of the DPU for the separation of hot-cold data. Besides, a workload analyzer is configured to analyze the characteristics of the workloads (e.g., read/write ratio, etc.), and an auto-selecting interface is implemented to dynamically recommend different compaction strategies to the RocksDB on the host.

The data buffer on the DPU is divided into three parts: $Data\ Set$, $Hot\ Set$, and $Cold\ Set$. Each part is composed of a set of hash tables. The structure of the hash table is shown in Fig. 4. A hash table contains several hash buckets, each with four slots, to ease hash conflicts and improve the load factor of the hash table. A key hashed into a hash bucket can select an empty slot in that bucket for insertion then set the $flag$ to 1 to indicate that the data is valid. When the load factor of a hash table in $Data\ Set$ exceeds a certain value (e.g., 0.9), the new key-value pairs will be inserted into a new hash table, and the old one will wait for a background thread to separate hot-cold data in it and flush the cold key-value pairs to the RocksDB on the host. At the same time, the workload analyzer analyzes the characteristics of the workload

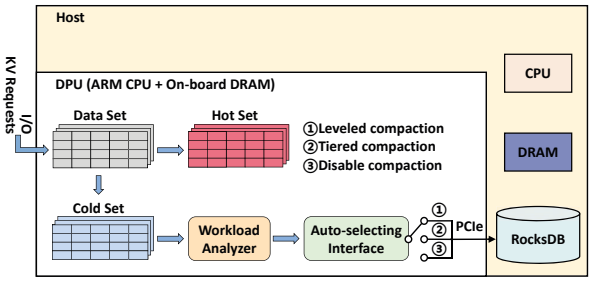


Fig. 3. iKnowFirst architecture overview.

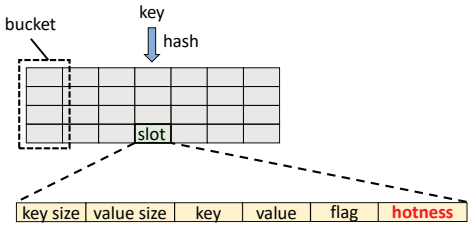


Fig. 4. The structure of the hash table.

and recommends different compaction strategies for RocksDB based on the workload type. After key-value pairs arrive at the host, the data operation follows the default RocksDB data manipulation process. We also provide a redo log option on the DPU to ensure data consistency, and users can choose according to their needs.

The design objective of iKnowFirst is to utilize the nature of DPU proximity data to offload data, prioritizing data perception and analysis. It follows two core design principles:

a) ***Setting a data buffer on the on-board DRAM of DPU and separating hot-cold data****:* We maintain several sets of hash tables on the DPU, which form a data buffer to cache and separate the hot-cold data. This allows us to reduce CPU resource overhead while keeping hot data on the DPU side for quick access.

b) ***Configuring a workload analyzer on the DPU and guiding the RocksDB to select a proper compaction strategy****:* We take advantage of the characteristics of DPU proximity data to offload data on the DPU, learn the features of workloads, and guide the RocksDB on the host to select an appropriate compaction strategy.

B. Seperating Hot-Cold Data

iKnowFirst leverages hot-cold data separation to accelerate access to hot data on the DPU, which is conducive to easing the pressure of subsequent LSM-tree compaction under write-intensive and highly skewed workloads. By exploiting the locality of workload access, the performance of key-value stores can be significantly improved. The specific process of hot-cold data separation is as follows.

As shown in Fig. 4, iKnowFirst maintains a $hotness$ for each key-value pair to collect statistics on the heat of the corresponding key-value pair. When a key-value request arrives, we first check the $Hot\ Set$ for the data. If there is such data, we carry out the operation and update the hotness. If not, we perform the operation in $Data\ Set$ and $Cold\ Set$. Each access to the data increases the hotness of the data by 1 in $Hot\ Set$ and $Data\ Set$. We count the $hotness$ of the key-value pairs within a period of time and set a threshold (e.g., 4). When the $hotness$ of a certain data exceeds this threshold, the data will be judged as hot data. When the load factor of a hash table in $Data\ Set$ reaches a specified threshold, it is transformed into an immutable hash table, which is then scanned by a background thread. The background thread inserts the key-value pairs in the hash table that exceeds the threshold into $Hot\ Set$, and the hash table is then added to $Cold\ Set$, waiting to be flushed to RocksDB on the host. In order to maintain data read consistency, the hot data in the $Cold\ Set$ will not be invalidated until it is flushed. Besides, the background thread periodically scans the hot set and weeds out outdated hot data. In this way, we can keep the hot data in $Hot\ Set$ on the DPU to support fast access to them.

As shown in Fig. 7(a), the task of hot-cold data separation is assigned to the ARM cores of the DPU, which can reduce the overhead of host CPU resources and reduce power consumption. Besides, the data update operation uses an in-place update strategy for the hash table. However, for the RocksDB KV store, the data update operation adopts an append update strategy. For some frequent hotspot data update operations, we save the hot data in the hash table on the DPU side and adopt an in-place update, which reduces the append update operation to the RocksDB on the host and the overhead of subsequent RocksDB compaction.

C. Auto-Selecting Compaction

iKnowFirst deploys a workload analyzer on the DPU and an interface to automatically select a compaction strategy based on the read/write intensity of the workload, and calls the interface to recommend appropriate compaction strategies to the RocksDB on the host.

The workload analyzer analyzes the frequency of requests (f) that are flushed from $Cold\ Set$ to RocksDB over a recent period of time, as well as the write ratio (w) of application requests. We take both these features into consideration. Specifically, when f is low but w is high, there are many write requests hitting the $Data\ Set$ and $Hot\ Set$, compaction of RockDB is not expected to be heavy, and the read ratio is low, so leveled compaction is recommended. When f is high and w is high, we think that RocksDB would face a lot of pressure for subsequent compaction and recommend disabling compaction. At the same time, we increase the size of $level0_file_num_compaction_trigger$ to ease write stall. When w is low, the workload is read-intensive, we chose leveled compaction, which can handle the compaction while achieving better read performance. When f and w are in the middle range, we will subdivide different situations to consider whether to choose leveled compaction or tiered compaction. We set corresponding thresholds for f and w to assist the

Algorithm 1 Auto-selecting compaction

Input: f, w and the corresponding thresholds Fl, Fh, Wl, Wh

if ($(f < Fl$ && $w > Wh)$ || $w < Wl)$ **then**
  $Choose\ leveled\ compaction$

else if $(f > Fh$ && $w > Wh)$ **then**
  $Choose\ disabled\ compaction$
  $resize\ the\ level0_file_num_compaction_trigger$

else if $(f >= Fl$ && $f <= Fh$ &&$w >= Wl$ && $w <= Wh)$ **then**
  $Choose\ leveled\ compaction\ or\ tiered\ compaction$
  $depending\ on\ the\ specific\ values\ of\ f\ and\ w$

end

interface in judging. We will constantly analyze the current status of application requests on the DPU and dynamically adjust the compaction strategy of RocksDB. Algorithm 1 shows our auto-selecting compaction process.

With auto-selecting compaction, iKnowFirst dynamically adjusts RocksDB's compaction strategy in real-time, which helps to improve the performance of the system.

D. Data Operations

Every item in a key-value store goes through the basic insert, update, search, and delete operation cycle. Fig. 5 shows the process of iKnowFirst's data operations.

Insert: For the data insertion process, the data is inserted into the $Data\ Set$ on the DPU and then waits for the background thread to process. If the load factor of the hash table in the $Data\ Set$ exceeds a specified threshold when the data is inserted, the data will be inserted into a new hash table.

Update: For the data update process, we first check if the key-value pair to update is in the $Hot\ Set$, if it is, we update it. Otherwise, we check if it is in the $Data\ Set$. And if it is, we update it. Otherwise, we insert it into the $Data\ Set$. We also update the $hotness$ of the data. Data in $Cold\ Set$ will not be updated in place since they will be flushed to RocksDB later. As described above, updates to data in $Cold\ Set$ are inserted into the $Data\ Set$, and they will be flushed to RocksDB later, ensuring data consistency.

Search: For the data search process, We first search the $Hot\ Set$. If the Data does not exist, we search in the $Data\ Set$. If the data does not exist, we search in the $Cold\ Set$. Besides, if the data is in $Hot\ Set$ or $Data\ Set$, we will update the $hotness$ of it. If the data is not on the DPU side, we search for it on RocksDB on the host.

Delete: The data deletion process is similar to the data search process. On the DPU side, we follow the steps of the search process. If the key-value pair to be deleted can be found, its $flag$ will be set to 0 for local deletion. If the key-value pair cannot be found on the DPU, we will append a delete tag to RocksDB in the default way.

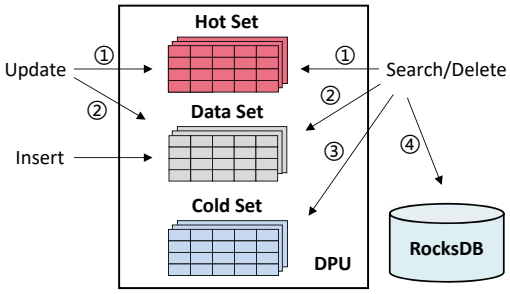


Fig. 5. Data operations of iKnowFirst.

iKnowFirst builds on the data operations illustrated above to ensure data consistency in a system that combines hash index and LSM-Tree.

IV. Experimental Results

In this section, we use a series of experiments to evaluate iKnowFirst. After describing our experimental setup, we conduct experiments to show the performance of iKnowFirst under the YCSB workloads with different read/write ratios and different value sizes. Besides, we test the performance of subsequent read-only workloads for various scenarios. Last, we conduct experiments to evaluate the CPU overhead and hash table's performance differences between CPU and DPU.

A. Experimental Setup

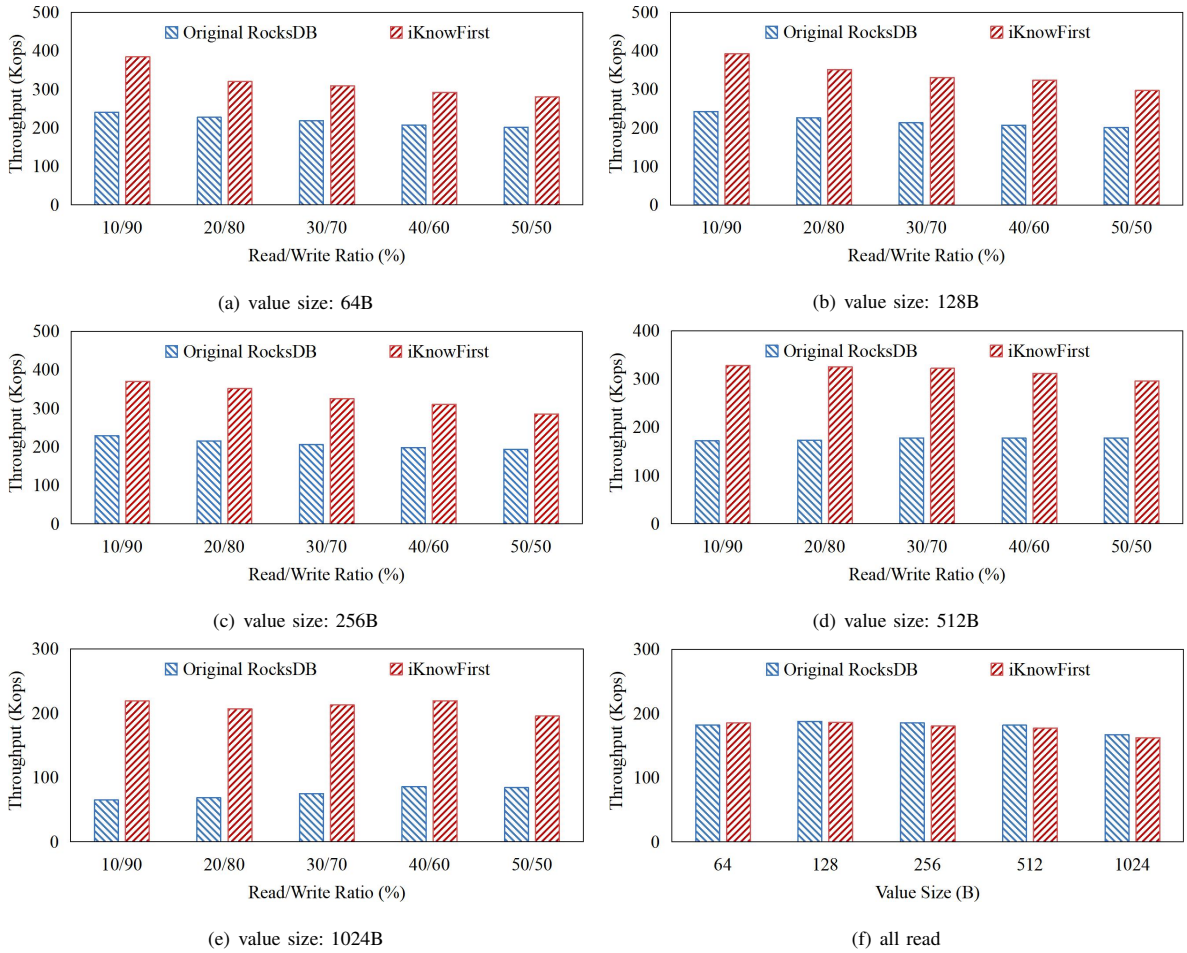
Experimental Platform: All experiments are performed on a Linux server with kernel version 5.4.0 and two Intel(R) Xeon(R) Gold 5318Y 2.10GHz CPUs. It has 128GB of DDR4 DRAM and an NVIDIA BlueField-2 DPU [3]. The NVIDIA BlueField-2 DPU comes with a Linux kernel version 5.4.0 and an 8-core ARMv8 A72 2.50GHz CPU. It is equipped with 16GB on-board DDR4-1600 DRAM. (See Table I for details.) The server also has a high-performance 500GB Samsung SSD 980 as the primary persistent storage.

Benchmark Tools: We use YCSB [10] to evaluate our scheme. YCSB is a tool developed by Yahoo to perform fundamental benchmarking of cloud services. Since the target of iKnowFirst is hot-cold separation, so in our evaluations, we use the YCSB to generate some skewed workloads (Access to 5% of data accounts for 95% of total access). The size of the Key is 8B, while the size of value varies from 64B-1024B, which is consistent with the previous works [4], [26], [29].

B. Overall Performance

We first compare the throughput of iKnowFirst and original RocksDB to prove the efficiency. In this experiment, The original RocksDB uses the default settings except that the Bloom filter is turned on. We evaluate iKnowFirst and original RocksDB with skewed workloads. Fig. 6(a)-(e) illustrate the throughput of the two schemes under different read-write ratios and different value sizes.

Overall, iKnowFirst outperforms the original RocksDB by 1.4×-3.4× with an average of 1.8×. Under different value



(a) value size: 64B

(b) value size: 128B

(c) value size: 256B

(d) value size: 512B

(e) value size: 1024B

(f) all read

Fig. 6. Evaluation results of YCSB benchmarks.

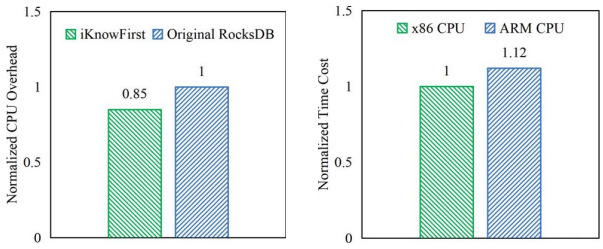
sizes, iKnowFirst has different advantages over the original RocksDB. For example, under workloads with a 64Byte value size, iKnowFirst outperforms original RocksDB by 1.4×-1.6×, but under workloads with a 1024Byte value size, iKnowFirst outperforms original RocksDB by 2.3×-3.7×. When processing small-value workloads, it is faster to write data into the hash table and RocksDB, and both DPU and host CPU need to process a large amount of data. In this case, both DPU and host CPU will be overloaded, so the effect is less than large-value workloads.

Under different read-write ratios, iKnowFirst also has different advantages. As the write ratio increases, the advantages of iKnowFirst gradually increase. This is because iKnowFirst does not specifically optimize the read operation. After a hot key-value pair is found in host RocksDB, iKnowFirst doesn't fetch them to DPU. So as the read ratio increases, most of the read operations may end up being hit in Host RocksDB, which weakens the performance advantage of iKnowFirst. The reason why iKnowFirst does not fetch read hot data to the DPU is that we believe that the buffer space of the DPU is more important for write operations. If a large amount of hot data is fetched from host RocksDB to the DPU hash table, the buffer for write operations will be reduced and flushes will be triggered frequently. In addition, fetching data from Host RocksDB to the DPU will also cause additional write amplification, occupying the bandwidth of the write operations.

We also evaluate the read performance of iKnowFirst and RocksDB in the full read scenario with different value sizes after a write burst scenario. As shown in Fig. 6(f), the performance of iKnowFirst is basically close to the original RocksDB. This shows that iKnowFirst will not have a greater impact on the performance of read operations.

C. Overhead Analysis

CPU Overhead: One of the main goals of iKnowFirst is to save Host CPU overhead. We compared the CPU overhead of iKnowFirst's Host RocksDB and the CPU overhead of the original RocksDB during the evaluations of Section IV-B. As shown in Fig. 7(a), the CPU overhead of RocksDB in iKnowFirst has been reduced by 18%, which proves that iKnowFirst successfully reduces the CPU overhead of the host



(a) The CPU overhead (b) x86 CPU vs. ARM CPU

Fig. 7. Overhead analysis of iKnowFirst.

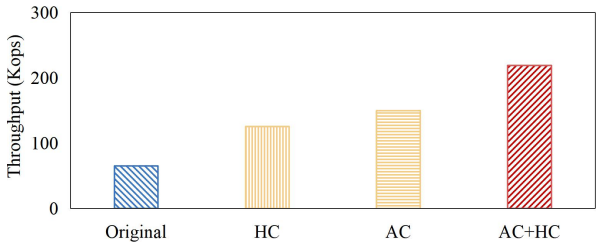


Fig. 8. The throughput when adding different design decisions under the write-intensive and highly skewed workload. (Original: original RocksDB; HC: RocksDB with hot-cold data separation; AC: RocksDB with auto-selecting compaction.)

CPU. There are two reasons: (1) The hot-cold data separation of the DPU reserves part of the hot data access on the DPU, thus reducing the processing pressure on the Host CPU. (2) The auto-selecting compaction on the DPU enables RocksDB on the Host to adopt a better compaction strategy, thereby reducing unnecessary CPU calculations.

Processing Power: Since it is generally believed that the x86 CPU on the host will be much stronger than the ARM CPU on the DPU, it is very important whether the DPU is capable of operating the hash table. To figure this out, we compared the performance between using a hash table on the Host side and using a hash table on the DPU side. As shown in Fig. 7(b), the ARM CPU using the hash table is only 12% slower than the x86 CPU. This is because the capacity of the hash table on the DPU is very small. Although the speed of the ARM CPU is relatively slow, the DPU can still quickly process this small amount of operations. In addition, since the DPU is at the first station of the I/O stack, compared with the data transmission time from the DPU to the host(tens of microseconds), the performance gap between the ARM CPU and the Host CPU seems insignificant. This result along with the above analysis proves that the processing power of the ARM CPU is sufficient to separate hot and cold data by operating a hash table.

D. Effectiveness of Each Design

Fig. 8 shows the improvements made by each design. Our experiment is under a write-intensive and highly skewed workload, and we use 1024 bytes keys. The result shows that both the hot-cold data separation (HC) and auto-selecting compaction (AC) designs improve RocksDB's performance to a certain extent, with the best results achieved when the two designs are combined.

V. Related Work

A. Optimizations of LSM-Tree

There are many solutions proposed to improve the efficiency of LSM-Trees from the software layer. Unikv [29] employs a hashtable in DRAM to efficiently index the unsorted data in Level-0 SSTables of LevelDB, which leverages the data locality and improves the performance. SplinterDB [9] employs a novel data structure STBϵ-Tree to replace the SSTables, which exposes I/O and CPU concurrency and reduces write amplification without sacrificing query performance. SpanDB [7] places the upper levels of LSM-Tree in fast NVMe SSDs and places the bottom levels in slow SATA HDDs. It also designs different access interfaces for different devices, thus achieving low cost and efficiency.

Some solutions try to enhance the LSM-Trees with hardware accelerations. X-Engine-FPGA [30] utilizes FPGA to accelerate the compactions of LSM-Trees. PinK [14] employs FPGA to LSM-Tree-based KVSSD to accelerate the sorting and the building of bloom filter in compactions.

B. Hot-Cold Separation in Key-Value Store

There are many solutions to address the hotspot issue. NetCache [15] exploits the capabilities of ASICs to efficiently detect, index, cache, and serve hot key-value items. DistCache [19] co-designs cache allocation with cache topology and query routing. The key idea is to partition the hot objects with independent hash functions between cache nodes in different layers and to adaptively route queries with the power of two choices. HotRing [8] is optimized for massively concurrent access to a small portion of items. It is based on an ordered-ring hash index structure, which provides fast access to hot items by moving head pointers closer to them. It also applies a lightweight strategy to detect hotspot shifts at runtime. Pacman [26] leverages hot-cold data separation to reduce the amount of valid data copying on persistent memory and corresponding reference updates during compaction. It uses a hotspot set to distinguish hot and cold objects and stores them in different segments to facilitate compaction.

C. DPU and SmartNIC Offloading in Key-Value Store

After the emergence of programmable SmartNICs and DPU, there are many solutions that focus on how to offload suitable operations from the host CPU to SmartNICs and DPU. KV-Direct [16] offloads the operations of the memory key-value store (e.g., Redis) to the SmartNICs by using RDMA to access the host DRAM and using the on-board FPGA processing complex operations, which achieves zero access to the host CPU and high efficiency. SKV [24] offloads replication operation and failure detection of the distributed KV store system to off-path multi-core SoC SmartNICs.

VI. Conclusion

We present iKnowFirst, which places a hash table set on DPU to absorb the hot accesses and senses the characteristics of workloads in advance. With the characteristics of workloads, iKnowFirst implements an auto-selecting interface to choose different compaction strategies for host RocksDB. Our experiments show that iKnowFirst achieves up to 3.2× improvement compared to the original RocksDB and also reduces the CPU overhead of the host.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No. U22A2027, No. 61972325, and No. 61872305, Open Research Projects of Zhejiang Lab (No. 2021DA0AM01/002), and Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2021WNLOKF011. Suzhen Wu is the corresponding author.

References

[1] LevelDB. https://github.com/google/leveldb.

[2] NVIDIA DOCA SDK. https://docs.nvidia.com/doca/sdk.

[3] NVIDIA DPU. https://www.nvidia.com/enus/networking/products/data-processing-unit.

[4] RocksDB. https://github.com/facebook/rocksdb.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, pages 53–64, 2012.

[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 209–223, 2020.

[7] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 17–32, 2021.

[8] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 239–252, 2020.

[9] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*, pages 49–63, 2020.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC'10)*, pages 143–154, 2010.

[11] Niv Dayan and Stratos Idreos. Dostoevsky: Better Space-Time Trade-Offs for LSM-tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 505–520, 2018.

[12] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[13] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-Engine: An Optimized Storage Engine for Large-Scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, pages 651–665, 2019.

[14] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind Arvind, and Sungjin Lee. Pink: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*, pages 173–187, 2020.

[15] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 121–136, 2017.

[16] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable Nic. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 137–152, 2017.

[17] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. Performance Characteristics of the BlueField-2 SmartNIC. *arXiv preprint arXiv:2105.06619*, 2021.

[18] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*, pages 363–378, Jul. 2019.

[19] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, volume 19, pages 143–157, 2019.

[20] Thomas Lively, Luca Schroeder, and Carlos Mendizábal. Splaying Log-Structured Merge-trees. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 1839–1841, 2018.

[21] Feng-Feng Pan, Yin-Liang Yue, and Jin Xiong. dCompaction: Speeding up Compaction of the Lsm-tree via Delayed Compaction. *Journal of Computer Science and Technology*, 32:41–54, 2017.

[22] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, pages 69–82, 2022.

[23] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment*, pages 2216–2229, 2021.

[24] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. SKV: A SmartNIC-Offloaded Distributed Key-Value Store. In *Proceedings of the 2022 IEEE International Conference on Cluster Computing (Cluster'22)*, pages 1–11. IEEE, 2022.

[25] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores. In *Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE'20)*, pages 1261–1272, May 2020.

[26] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22)*, pages 773–788, 2022.

[27] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 93–111, 2021.

[28] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proceedings of the VLDB Endowment*, 14(10):1872–1885, 2021.

[29] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. Unikv: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE'20)*, pages 313–324. IEEE, 2020.

[30] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 225–237, Feb. 2020.