KVRangeDB: Range Queries for a Hash-based Key–Value Device

MIAN QIN, Texas A&M University, USA
QING ZHENG and JASON LEE, Los Alamos National Laboratory, USA
BRADLEY SETTLEMYER, Nvidia, USA
FEI WEN, NARASIMHA REDDY, and PAUL GRATZ, Texas A&M University, USA

Key–value (KV) software has proven useful to a wide variety of applications including analytics, time-series databases, and distributed file systems. To satisfy the requirements of diverse workloads, KV stores have been carefully tailored to best match the performance characteristics of underlying solid-state block devices. Emerging KV storage device is a promising technology for both simplifying the KV software stack and improving the performance of persistent storage-based applications. However, while providing fast, predictable put and get operations, existing KV storage devices do not natively support range queries that are critical to all three types of applications described above.

In this article, we present KVRangeDB, a software layer that enables processing range queries for existing hash-based KV solid-state disks (KVSSDs). As an effort to adapt to the performance characteristics of emerging KVSSDs, KVRangeDB implements log-structured merge tree key index that reduces compaction I/O, merges keys when possible, and provides separate caches for indexes and values. We evaluated the KVRangeDB under a set of representative workloads, and compared its performance with two existing database solutions: a Rocksdb variant ported to work with the KVSSD, and Wisckey, a key–value database that is carefully tuned for conventional block devices. On filesystem aging workloads, KVRangeDB outperforms Wisckey by 23.7× in terms of throughput and reduce CPU usage and external write amplifications by 14.3× and 9.8×, respectively.

24

CCS Concepts: • **Information systems → Flash memory**; **key–value stores;**

Additional Key Words and Phrases: Key value stores, KVSSD, range queries

ACM Reference format:
Mian Qin, Qing Zheng, Jason Lee, Bradley Settlemyer, Fei Wen, Narasimha Reddy, and Paul Gratz. 2023. KVRangeDB: Range Queries for a Hash-based Key–Value Device. *ACM Trans. Storage* 19, 3, Article 24 (June 2023), 21 pages.
https://doi.org/10.1145/3582013

This work is supported in part by grants from Samsung, Los Alamos National Lab and NSF Grants 1439722, 1823403 and 2203033.

Authors' addresses: M. Qin, Fei Wen, N. Reddy, and P. Gratz, Texas A&M University, 3128 TAMU, College Station, TX, 77843, USA; emails: celery1124@tamu.edu, fei8wen@gmail.com, reddy@tamu.edu, pgratz@gratz1.com; Q. Zheng and J. Lee, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM, 87545, USA; emials: qzheng@lanl.gov, jasonlee@lanl.gov; B. Settlemyer, Nvidia, 11001 Lakeline Blvd, Austin, TX, 78717, USA; email: bsettlemyer@nvidia.com.

1553-3077/2023/06-ART24 $15.00
https://doi.org/10.1145/3582013

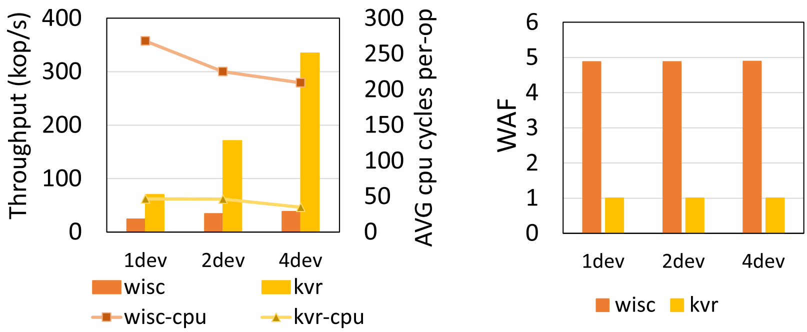
1 INTRODUCTION

As the importance of **key–value (KV)** workloads has increased, so has the sophistication of modern KV databases [9, 12–14, 22, 25, 31]. Popular key–value databases, such as RocksDB [14], are carefully optimized to extract performance from underlying flash-based SSDs. **Log-structured merge (LSM)** trees [26] are used as the core data structure for these purposes: adaptive I/O size, disk request alignment, and key–value store ordering. Mordern SSDs are usually managed in larger and aligned blocks. In spite of the significant efforts spent to improve the efficiency of storing small keys and values into larger blocks, the block-oriented interface still leads to several possible sources of inefficiency for KV workloads. First, the minimum device I/O is bound to the block size, regardless of the requested key and value size. As a consequence, latency-sensitive workloads without effective data prefetching experience large amounts of read amplification, as 4K-blocks are read to retrieve much smaller values. Second, to minimize the external read and write amplification associated with LSM compaction, existing state-of-the-art KV store only performs compaction on keys, while values are stored separately. Such storage policy provides efficient insertion and retrieval performance, at the cost of expensive garbage collection when values are frequently updated or deleted. Finally, interest is rising in computational storage devices or storage devices that support the offloaded programmed analysis and reduction functions. These devices promise much lower query latencies, as common searching and reduction functions can be performed within the storage without sending data back to the host CPU. However, when the structure and metadata describing the LSM is updated in the host memory instead of the storage device, it is typically impossible to semantically interpret the contents of a block device.

To address these three issues, researchers proposed key–value interfaces for flash-based storage called **KV solid-state disks (KVSSDs)** [1, 4]. These KVSSDs directly support the insertion, retrieval, and deletion of arbitrarily sized KV data. During this process, two competing device designs have arisen that attempt to address different workloads. Hash-based KVSSDs, such as the one produced by Samsung [18], deliver fast individual KV operations but are incapable of range-ordered iteration. In contrast, LSM-based KVSSDS [15, 16, 35] require additional on-device processing support, but they maintain the key ordering entirely within the KVSSD, thus eliminating the external read and write amplification incurred by compaction. Most real-life applications are a mix of both point and range queries [20, 28, 36, 39, 41]. While a workload containing *any* range queries can benefit from an LSM-based KVSSD, we can expect substantial performance degradation given a large portion of the workload is not range queries. Significant processing power within LSM-based KVSSDs are allocated to maintain the LSM organization, slowing down the point queries. Hence, a hash-based KVSSD with support for range queries in host applications could become an attractive solution to the dilemmas described above.

In this article we present KVRangeDB, a KV store designed to exploit the fast point operations from hash-based KVSSDs while providing support for efficient range queries. KVRangeDB is implemented on the host side in a layer between the KV applications and the KV device. We employ similar ideas of the key value separation used by Wisckey [23], i.e., using a small in-storage LSM-tree to store keys and preserve key order, while the value is stored separately using the device's KV interface. The key difference from Wisckey is that we preserve point query performance by directly accessing the device through key value interface instead of using an LSM tree index that incurs multiple I/O operations. Additionally, KVRangeDB effectively offloads the value log garbage collection required by Wisckey into the device, which significantly mitigates host side CPU usage and reduces the external write amplification.

We conduct experiments to demonstrate how KVRangeDB outperforms Wisckey with respect to aging. In these experiments, we first load 500 million records. Then we perform three rounds of aging process, consisting of multiple delete/update/insert operations. Garbage collection



(a) Performance and CPU(us) cost (b) Write Amplification Factor

Fig. 1. Record aging comparison between Wisckey, a block device key–value database, and KVRangeDB, a KVSSD key–value database.

was triggered for Wisckey at the end of each aging round. Detailed experiment setup and methodology are described in Section 4. Figure 1 presents the results of Wisckey on traditional block SSDs and KVRangeDB on KVSSDs. The results show that Wisckey experiences substantial write amplification at the host compared to our proposed KVRangeDB on key–value interfaced device. This write amplification leads to lower throughput and higher CPU costs at the host when KV stores interface with block-level devices.

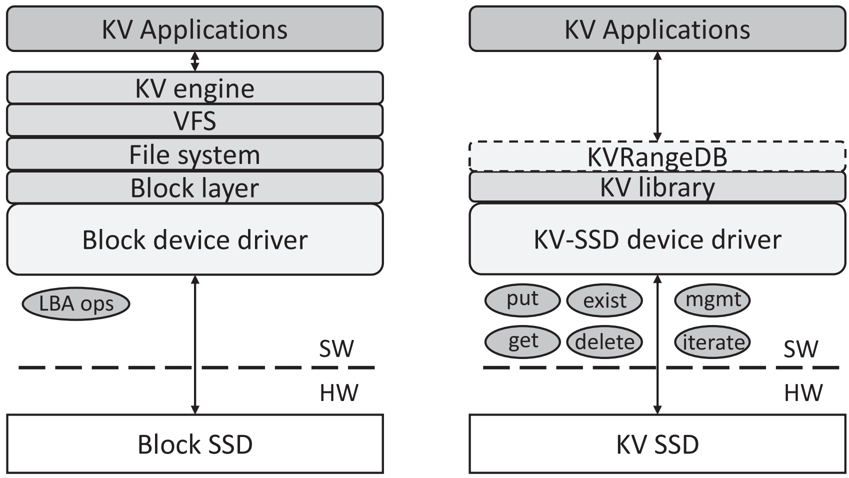
To better understand the performance characteristics of the key–value interface, we have focused on two important storage system workloads: key–value database workloads and file system metadata workloads. Both of these workloads emphasize the performance of small update, retrieval, and deletion operations that are already supported by the existing key–value device. However, key–value databases and file system metadata operations also leverage range queries that require the retrieval of sequences of consecutive key–value pairs. Such types of operations are are not natively supported by the device interface of the commercially available key–value storage device to date. To cope with this limitation, we provide an efficient software-based range-query capability to analyze the device under realistic usage scenarios.

In this article, we describe the detailed design of KVRangeDB and the performance of the only commercially available KVSSD and we attempt to answer the question whether a key–value interface for the storage device is superior to the traditional block interface. The contributions include the following:

- A detailed design of KVRangeDB that employs various novel techniques to enhance efficiency of a hash-based key–value storage device.
- Comparison of a key–value workload using a hash-based key–value storage device and a block device.
- Comparison of file system metadata workloads using a hash-based key–value device and a block device.
- A senescence/aging analysis of block-based key value databases and a KV database implemented on a key–value interface.

2 BACKGROUND

In this section, we briefly review the emerging key–value interface storage devices and the state-of-art for software key–value stores. Then we introduce how modern file systems use key–value storage to manage metadata.



(a) Software KV system stacks (b) KVSSD system stacks

Fig. 2. Comparison between (a) traditional software KV system stack and (b) KVSSD system stack.

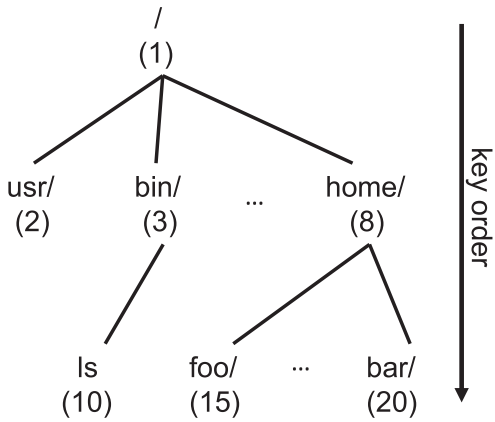
2.1 KVSSDs

Flash vendors have provided users a variety of alternative interfaces to flash-based storage devices. Open Channel SSDs [6] moved the majority of the FTL into software allowing users to manage the physical placement of blocks and access the device's internal parallelism. More recent Zone Namespace devices [1, 5] provide an interface that allows users to leverage a block-oriented page append interface and indicate to the devices groups of blocks that can be erased efficiently. Most recently, the storage industry has standardized a key–value device interface [1, 4] that simplifies the mapping of popular key–value software interfaces to the device interface [15, 16, 35]. Currently, Samsung provides KVSSD products [18] with a hash table implementation [18, 19] targeting fast put/get performance and low write amplification.

Figure 2 illustrates the system stacks for KVSSD-based systems. Traditional software KV stores involve complex key–value to file and then file to block translations done by the file system and block layer of the operating system. By contrast, KV stores based on KVSSDs leverage a thin layer of software consisting of only a device driver and a user space KV library. KVRangeDB is built on top of the KV library layer as shown in Figure 2(b). KVRangeDB can be also seen as an enhancement of the KV library layer. The KVSSD provides put, get, delete, as well as basic KV iteration operations. Its KV iteration interface allows traversing a group of keys (with the same 4B key prefix) without key ordering. For ordered key scans for arbitrary keys, we implemented a range query engine using the device iterator capable of retrieving all keys stored on a device. We then used an in-memory priority queue to store all keys from the seek position up to the scan length. The range query latency turned out to be impractically long for real-life applications, in the ballpark of tens of seconds for a 10 million records dataset.

2.2 Modern Software KV-stores

Modern KV-store applications [8, 9, 13] rely on software KV engines to translate the key–value interface to the block interface used by HDDs or SSDs. State-of-art software KV stores [3, 12, 14] use LSM-tree data structures [26] for efficient reads and writes. LSM-trees organize KV objects into multiple levels of large, sorted tables (SSTable). All writes and updates occur as out-of-place



Key	Value
<0, “/”>	{1, stat, …}
<1, “bin”>	{3, stat, …}
<1, “home”>	{8, stat, …}
<1, “usr”>	{2, stat, …}
<3, “ls”>	{10, stat, …}
<8, “bar”>	{20, stat, …}
<8, “foo”>	{15, stat, …}

Fig. 3. TableFS metadata management schema illustration.

writes to the top-level table. Reads search from the top-level table to the bottom-level tables for the most recent data. LSM-trees achieve high performance by converting small writes into large sequential I/Os that are optimal for the underlying device. However, this comes at the cost of high CPU utilization and I/O amplification as previous work shows [23, 25, 31].

LSM-trees use compaction for efficient KV scans and get performance. To reduce the write amplification overhead caused by compaction, Wisckey [23] proposes the separation of keys from values for LSM-tree-based KV stores. Wisckey stores values in a log and maintains a small LSM-tree as an index that maps keys to offsets to the value log. While improving write performance, this indirection reduces range query efficiency. The value log additionally requires garbage collection that adds complexity to the design.

2.3 File Systems Using Ordered KV-stores

Several local and distributed file systems [2, 21, 27, 32, 37, 40] used KV stores for file system metadata management. The main advantages of using KV-store for metadata management instead of traditional extent trees or B-trees is scalability and write performance. However, for efficient directory traversal, these systems often require the underlying KV-store to provide efficient ordered KV scan operations.

In this section, we describe TableFS [32] as an exemplar and briefly introduce how to use a KV-store to manage file system metadata. Figure 3 illustrates the metadata schema of TableFS. Each TableFS record stored in the KV-store corresponds to a file or directory in the file system. The variable length key consists of a 64-bit inode number of the parent directory and the name of the file. The value contains the inode number of the file and the file's various attributes (type, size, permission bits, owner information, etc.).

To resolve a full file system path name, TableFS starts searching from the root inode. Then it traverses each level of the directory tree with a search key that combines the inode number of the current directory and the next component name in the path. For file system operations such as **mkdir, mknod, unlink, lstat**, and so on, the file name is first resolved and then the corresponding `put`, `get`, `delete` KV operation is performed. For **readdir** operations, range queries are used. TableFS first resolves the target directory path name and then range queries records using the directory's inode number as the key to list all children of that directory. TABLEFS implements a light-weight locking mechanism [32, 34] to guarantee the atomicity and the correctness under concurrent accesses.

Table 1. Key Challenges and Corresponding KVRageDB Design

Challenges	Design ideas
How to saturate device bandwidth for small records?	Packing multiple small records (Section 3.2)
How to implement efficient index structure?	LSM tree structure on top of native KV interfaces (Section 3.3)
How to amortize latency for separate value retrieval?	Leverage value prefetch and packing heuristic (Section 3.4)
How to improve efficiency for empty queries?	Hierarchical bloom filter for point and range queries (Section 3.5)

3 KVRANGEDB

KVRangeDB is designed to support efficient range queries on hash-based KV storage devices while retaining the native `put/get` performance benefits from the device. A critical feature is to manage an ordered key index separately from the data. For a range query, we will first check the key index and find the target keys in the queried range and then retrieve the values from the device. The idea seems straightforward, however, there are many problems to consider.

Table 1 outlines the key challenges for efficient range queries on hash-based KV storage devices and how our KVRangeDB design addresses those fundamental problems. We will detail our design choices in the following sections.

3.1 Basic API

KVRangeDB provides key–value semantics with range query support, similar to the APIs of RocksDB [14] and LevelDB [12]. An iterator interface is provided to perform range query or scan operations. We define the following APIs for our KVRangeDB (the user hint API will be discussed in Section 3.4):

- `put(k,v)`: Put new key–value pairs.
- `get(k,v)`: Retrieve value from key.
- `delete(k)`: Delete key–value pairs.
- `iterator`: Iterator for range query.
  - `seek(k)`: Moves the iterator to the first key–value pair with key greater than or equal to the seek key.
  - `next()`: Move the iterator to the next key–value pair.
  - `valid()`: Whether iterator is valid.
  - `key()`: Return the key of the current iterator.
  - `value()`: Return the value of the current iterator.
  - `hint.scan_length`: Specify the user hint for the scan length.

3.2 Packing Smaller Records

In the rest of the article, we use logical keys and user keys interchangeably as the application keys. We define physical/device keys as the actual key written to the device with the KVSSD KV interface. For smaller size records, packing multiple values into a single physical record can yield better write throughput and mitigate the performance decrease at large key counts. The logical key to physical key mapping can also serve as a key index to provide the range query capability over logical keys, accomplishing two goals with one mechanism.

Figure 4 illustrates how smaller KV records are packed into a large physical record. Samsung KVSSD shows almost flat put IOPS for records smaller than 4 KB [18, 30]. Multiple smaller

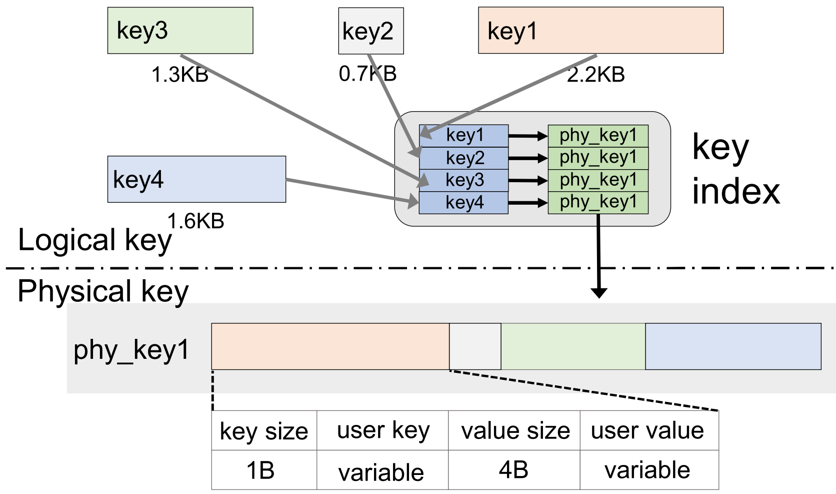


Fig. 4. Packing smaller records and translating user keys.

logical records can be packed into a single large physical record around 4 KB to yield higher write throughput and reduce the number of physical keys managed on the device. The key index keeps the logical keys to physical keys mapping for retrieving records by the logical keys, which requires a linear scan on the physical record to extract the user record. To support range query on the logical key, we use LSM tree to maintain the logical key to physical key translation. The main reason to choose LSM tree as the data structure for logical to physical key mapping instead of traditional B/B+ tree is to achieve higher write performance [12, 14, 26].

Point query processing: When performing point queries on the packed records, LSM tree key index is consulted first to find the physical key and then the value from the packed physical record is retrieved.

Range query processing: When range queries are performed, the LSM tree key index is traversed to find the physical key mapped to the target user key and retrieve the values in the queried range separately.

Update and remove operations: We propose two approaches to handle update and remove operations of the existing packed records. First, we can use *in-place update* mechanism that requires read-modify-write to the packed physical records. Alternatively, we can always assign new physical keys to the updated records (*out-of-place update*) to maintain high write throughput and apply background garbage collection to clean up the stale records as shown in Figure 5.

3.3 Building a Key Index for Range Queries

This section will describe in detail how we design the key index to support range queries on a key–value storage device. Compared to Wisckey [23], which also employs key value separation, Our design has two main advantages.

- The get operations, or point queries can be fulfilled by a single read I/O directly from the device.
- We effectively offload the value log garbage collection to the device side which significantly reduces host CPU usage and external read/write amplifications.

As we mentioned in Section 3.2, to achieve high write/put throughput performance we choose an LSM tree-based key index for logical key to physical key mapping when packing smaller records. For other records (large value records or frequently accessed records), however, we simply leave

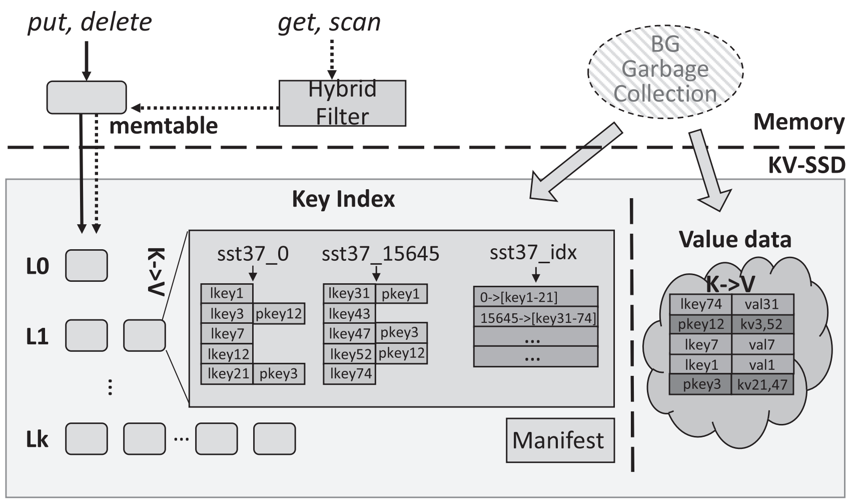


Fig. 5. LSM tree key index design for supporting range queries.

the logical keys in the LSM tree key index to achieve logical key ordering for range queries and use the logical key as the device key directly without the need for logical to physical key translation. This is a core difference when compared to Wisckey [23]. Wisckey needs to consult the key index for both point and range queries to locate and retrieve the values from the value log. However, for point queries with KVRangeDB, we can bypass the index and directly use the logical key to retrieve the value from the device with exactly one I/O for large unpacked records. For example, as shown in Figure 5, *lkey1, lkey7, lkey12*, and so on, are unpacked records that can be retrieved directly from device through the logical keys. Records *lkey3, lkey52* are packed into a physical record (physical key pkey12) and need to go through key translation to retrieve the value of the records.

To balance write and range query performance, we must carefully design the LSM tree structure. In our LSM tree index design, we use separate keys to store each data block and the index block (here block is not fixed size block in the block device, it can be any size). Figure 5 illustrates the LSM tree key index for KVRangeDB. Similarly to levelDB and rocksDB, the LSM tree index contains a memtable, multiple sorted SStables based on logical keys and manifest. The manifest uses a single KV record. For SStable storage, we use separate device KV pairs to store each data block and index block. The data block keys are the SStable number plus the offset. There is a single device KV pair for each index block using the SStable number as the key and with the value containing the key range information and offset for each data block.

It is not practical to expect that users always know in advance whether a logical key is in a packed record or directly stored as a physical key. To cope with that, we implement a scheme called hybrid key translation. Hybrid key translation provides an efficient mechanism to determine if the key index must be consulted or can be bypassed for a value retrieval. In the context of *get* operations processing, it requires checking the key index to make sure whether the queried key is translated or not. In our design, we leverage a small bloom filter [7] to reduce the overhead of key index checking when

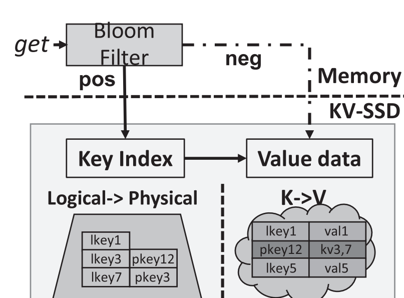


Fig. 6. Bypassing index checking for hybrid key translations.

the keys are not translated and can be directly retrieved from the device with the logical/user keys.

As illustrated in Figure 6, when we process the *get* operations, we consult the bloom filter. If the filter returns negative (dashed arrow), then the queried key is definitely not translated, and we directly retrieve the value from the device with the logical key. Otherwise (solid arrow), we will consult the key index to find the physical key for the value. In a false-positive case, i.e., logical key are not translated to physical key, we will still go to the key index for consultation (solid arrow). Since the key index has the global view of all the logical keys mappings (if logical keys are not translated, the physical key counterpart will be null as shown in Figure 6). Then we go to the device to retrieve the value with the logical key. Since the false-positive rate is relatively low, the overhead of extra key index checking can be neglected.

3.4 Value Prefetching and Caching for Range Queries

The key index introduces two problems that hinder efficient query processing. First, unlike LSM-based software KV stores [12, 14] that pack key and values together, the range query in our design will need to consult the index first to determine the target keys in the queried range. Then, we need to separately issue I/Os to retrieve the values associated with the target keys if the user also asks for values. This requires additional I/Os to satisfy the range query. Second, the record packing introduces key translation from logical keys to physical keys, and the value may not be directly retrieved with the logical key for simple point queries. In such a case, the point query performance will be impacted by the additional I/Os for index look up.

We propose two approaches to resolve these issues. The first approach is to leverage user hints for prefetching the values to overlap the value retrieval latency. We implemented two additional read options for range query, i.e., *scan length* and *upper bound key*. Since users may have prior knowledge of the queries (e.g., during a table scan, what is the approximate number of entries in the table; or in a query for events between two timestamps, what is the end timestamp, etc.), by applying those hints, we can prefetch the values in advance to hide the latency for accessing the values separately. Besides user domain knowledge, proper profiling can be also used to help extract hinting information to better leverage our hint interface. We also design a prefetch throttling mechanism to prevent too many in-flight prefetch requests that may increase the device queueing time and affect the other I/O requests.

The second approach is to leverage the temporal locality of the packed records. Application may write the adjacent records (in defined key order) together and may get packed into a single physical record. When range queries are performed, the LSM tree key index is traversed to find the physical key mapped to the target user key. We will cache the other records packed in the same physical records. When following ***next()*** and ***value()*** is called, we can examine small cache and on a hit, return the value directly without issuing I/O to the device. In the worst-case scenario, we still perform the same number of I/Os as no packing. In Section 4.3, we will demonstrate how real-world applications can leverage packing for range queries.

3.5 Range Filter for Empty Queries

Given typical key sizes compared to the typical number of records stored within a key–value database, most stores have only a small portion of the key space occupied. As a result, queries may result in empty/negative replies, and thus we need an efficient mechanism for deciding that keys do not exist. In KVRangeDB, we design hybrid filters as an auxiliary structure to filter out empty/negative queries for both point and range queries. (For example, "Is key 7 in the store?" or "Do any keys between 3 and 100 exist in the store?"). Filters are compact/compressed structures that can be completely stored in memory. For a typical data store with 1 billion keys, the key index

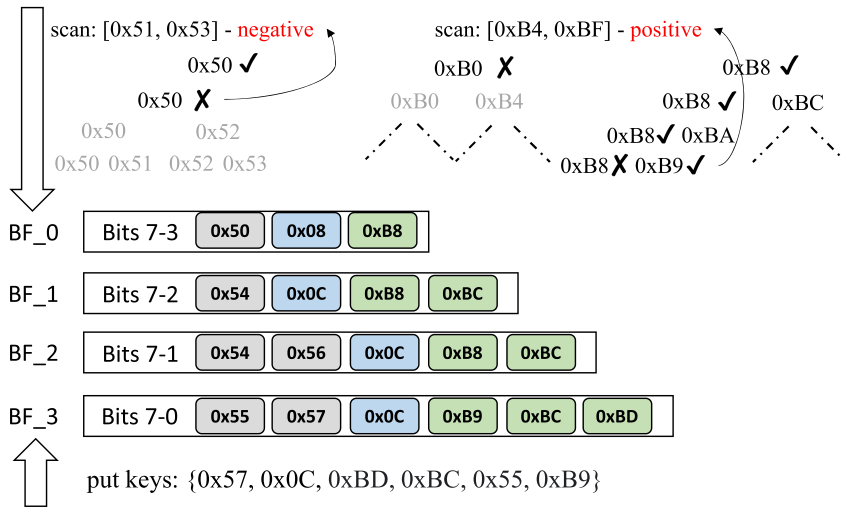


Fig. 7. Hierarchical bloom filter for range queries filtering.

size may be tens of gigabytes that may exceed the available memory resources dedicated to the database and an alternate filter must instead be designed as an in-storage data structure. Typical filters only require very small memory footprint (one to two gigabytes per billion keys) and can fit into small memory budgets successfully.

Bloom filters are well studied [7] and have been deployed in various software key–value stores [12, 14]. As mentioned earlier, KVRangeDB also uses a bloom filter for our hybrid key translation algorithm. However, simple bloom filters are not efficient to handle range queries ("Do any keys between 3 and 100 exist in the store?"). We could query every possible key by accessing bloom filters multiple times (from key 3 to key 100) to determine whether the queried range exists. However, such a method suffers from high CPU cost and false-positive rate.

Recently, more advanced filters were proposed [24, 39] with similar purpose for range queries especially for those short-range queries with high probability of being empty. Unlike the prior work [24, 39] that is designed for block-based range filters targeting LSM-tree-based KV stores, we propose a lightweight unified in-memory range filter for accelerating both empty point queries and empty range queries. As opposed to storing filters for each sorted-run, we do not store any filter data in storage but build the filter on the fly when opening the database (we can also persist the filter data on the devices). There are two main reasons for this design. First, unlike LSM-tree-based KV-stores that need to scan the entire database to retrieve all keys in the database, our KVRangeDB separates the sorted key index from the value store and can retrieve the keys efficiently. Second, building the range filters on the fly is more flexible to accommodate fast shifting workloads by altering the filter designs. For example, some workloads that frequently result in empty point queries may only need a simple bloom filter with lower memory costs, and workloads that rarely encounter empty queries may simply discard the range filter altogether.

In our design, we extend the idea of prefix bloom filter [14] and use multiple layers of a prefix bloom filter, each with different sizes of prefix to enable efficient range filtering. When building the filter, each key in the database stores various length prefixes into each level of bloom filters (the bottom level stores the full key bloom filter that also works for point queries). Range queries that consult the filter will be broken down to multiple prefix sets according to the top-level bloom filter prefix length. For each prefix set covered by the top-level prefix length, it can then recursively probe the lower-level bloom filters to determine if there are potential keys in the checked range. As long as there is one possible key existing in the queried range, the filter will return positive and requires checking the key index in storage to satisfy the query. However, if the filter returns negative, then the queried range is definitely empty, and we can directly return and save the I/O cost of checking the key index.

Figure 7 illustrates how our hierarchical range filter works, through an example. Consider we store records with the keys shown at the bottom (the key size is 2 bytes, 0x57...). The hierarchical range filter is designed as four levels of bloom filters (*BF_0* to *BF_3*). The top-level bloom filter *BF_0* is constructed using the 5 MSB bits prefix (bits 7-3) of all keys in the store. Each element in the top-level bloom filter covers a range of eight keys. A negative result of *BF_0* filter check means there is definitely no key existing in this key range. Each lower level filter uses 1 more bit prefix and covers half of the key range than the upper-level filter. The bottom level stores the full key bloom filter that also works for point queries. As an example, consider range query “scan [0x51, 0x53]” as shown in the top left of Figure 7. We first consult the top-level filter whether 0x50 prefix (covering key range from 0x50 to 0x57) exists. The top-level filter returns “yes,” and it keeps consulting the next level with 0x50 (*BF_1* prefix 0x50 covers key range from 0x50 to 0x53). *BF_1* returns “no,” which means the query key range (0x50 to 0x53) is definitely empty in the store. A range query may break into multiple prefix checks, each covering a smaller range as shown in the query example on the top right. As long as there is one positive result when we reach the bottom level, a key possibly exists in the queried range.

One critical component of a hierarchical bloom filter design is the amount of memory to dedicate to each level of the data structure. Intuitively, higher-level filters may contain less distinct keys due to shorter prefix lengths and may require less memory for the filter. In our design, we use a simple strategy to allocate the memory footprint as follows, which has thus far worked well,

$$M_i = M\frac{i+1}{\sum_{n=1}^{N} n}, i = 0, 1 \ldots N-1,$$

where N is the number of levels for the Hierarchical bloom filter, M is the total memory budget for the range filter, and M_i is the memory budget for the ith-level filter (i start from 0 to N).

4 EVALUATION

This section presents the experimental results of **Yahoo! Cloud Serving Benchmark (YCSB)** [11] and TableFS [32, 34], a real-world KV application that relies on range queries. We compare KVRangeDB against two other solution: Wisckey [23], the state-of-art software KV-store on block devices, and RocksDB [14], the industry counterpart, ported to KVSSD. We analyze how each optimization technique presented here contributes to the overall performance improvement and how they impact different collections of KV operations.

4.1 Methodology

4.1.1 Experiments Setup. Table 2 lists the detailed hardware information. Block SSD and KVSSD use the same SSD hardware device, except that the firmware is different.

Since the complete Wisckey source code is not disclosed to public, we implemented Wisckey according to the article for this evaluation. Instead of using LevelDB to store the user key to <log offset, value size> mapping in the original article, we use RocksDB [14], which has better overall performance. To make our comparisons using the same memory budget and exclude page cache effects for the block device, we use direct I/O mode for the Wisckey implementation, including the RocksDB index and value log operations. The evaluation configurations within our experiments are listed as follows:

Table 2. Hardware Specification

Component	Description
CPU	Intel Xeon Silver 4216 @ 2.1 GHz, 16 cores
Memory	96 GB DDR4 @ 2133MHz
SSD	PM983 3.84TB x4, (~580k 4 KB read IOPS)
KVSSD	PM983 3.84TB x4, (~200k 4 KB read IOPS)
Memory	128 GB DDR4
OS	Linux version 4.15

- **Wisckey:** Wisckey implementation on a conventional block SSD. The values are packed in a contiguous log file with 1MB log buffer. The key to log offset mapping for each record is stored in RocksDB.
- **RocksKV:** RocksDB implementation ported to the KVSSD. It uses the key–value interface instead of a file system interface to store the SSTable files and metadata files. For SSTable files, we store each data block with a separate record using the combined SSTable file number and block offset as the key. Manifest files are stored as a monolithic record.
- **KVR:** Baseline KVRangeDB implementation without hybrid record packing.
- **KVR-PF:** Baseline KVRangeDB optimized with value prefetching for range queries.
- **KVR-PK:** Baseline KVRangeDB optimized with hybrid record packing.
- **KVR-PK-PF:** Baseline KVRangeDB optimized with hybrid record packing and value prefetching.

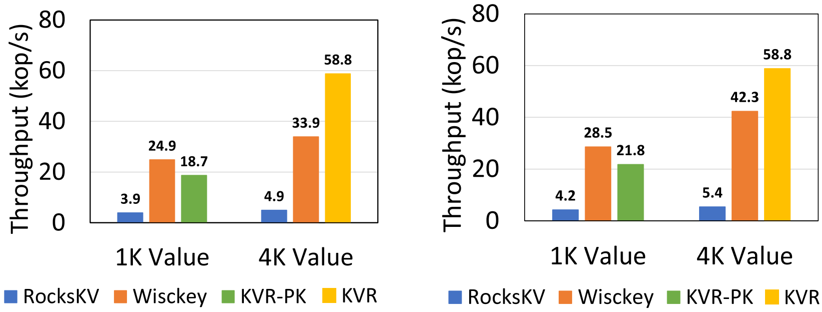
4.1.2 Workloads. We conducted two categories of experiments to evaluate the above systems.

- To measure the KVRangeDB performance, we run comprehensive micro-benchmarks including scan operations of various length, with/without retrieving values, as well as simple `put`, `get`, and `seek` operations under YCSB. Quantitative description for each query workload is explained in the following sections.
- File system applications under TableFS, which utilizes a KV-store as its metadata management engine. This application uses a large real-world directory tree, executes find commands, lists file/directory contents, and list metadata that are all composed of mixed `put`, `get`, and range queries, and we emulate the file system aging process with multiple rounds of updating, removing, and inserting files/directories.
  TableFS only uses KV-store to store the file metadata. The file data blocks are stored separately. In our experiments, we use KVRangeDB to replace TableFS's KV-store (LevelDB) and only examine the metadata operations that are the main bottleneck of the filesystem workloads [32]. The actual file data blocks are not included in the given filesystem tree and our experiments.

For micro-benchmarks (YCSB), we use single SSD/KVSSD. To better emulate real-world TableFS application, we use four devices in RAID0 mode. We use linux md to configure RAID0 for block SSDs. For the four KVSSD array, we spread the records through hashing the key [29].

4.2 Results for YCSB

We use two datasets for YCSB experiments: The first dataset of 250 million large records (with 16-B key and 4000-B value size) does not leverage packing; second dataset of 1 billion small records (with 16-B key and 1000-B value size [38]) can leverage packing (we pack four logical records to form a physical records). For all of our experiments, we first load all the data on the device (the index is written with the data). We then run different query workloads to examine



(a) Throughput with no cache (b) Throughput with 1GB cache

Fig. 9. YCSB Get performance (16 threads).

the performance of KVRangeDB, RocksKV, and Wisckey. For KVRangeDB, the bloom filter filter described in Section 3.3 is constructed during the loading phase and persisted to the KVSSD when database is closed. However, we bypass the bloom filter checking in the get workload, since it is either fully packed or unpacked.

Write performance. Figure 8(a) demonstrates the throughput performance of loading data onto the device. For smaller records, packing can be useful in improving the overall write throughput and reducing the number of keys managed by the device as we discussed in Section 3.2. The loading throughput of KVR-PK outperforms RocksKV by 14× and Wisckey by 1.3×. It is also worth noting that RocksKV requires greater compaction I/O, since it packs keys and values together. Packing more records into a physical record yields higher write throughput, and thus it enhances the data loading efficiency. KVR-PK is beneficial for write-heavy use cases that contain lots of small records. For 4000-B value size, KVR can achieve 18.8× better performance compared to RocksKV. KVR performs slightly (∼15%) worse than Wisckey in terms of write operations, as Wisckey leverages large sequential I/O for writes. However, Wisckey's implementation suffers on removes and updates (which require host-side garbage collection), contrary to KVRangDB that can directly remove and update records from device through the user key. We evaluate remove performance as part of the file system workloads in Section 4.3.

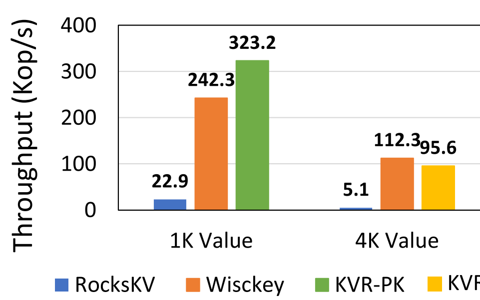
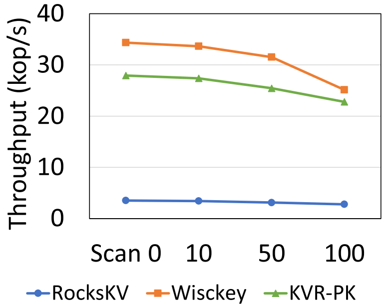


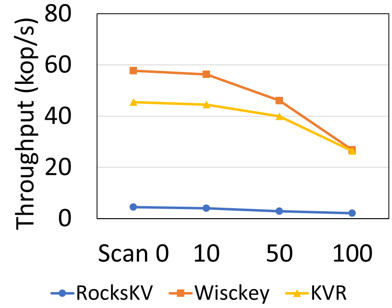
Fig. 8. YCSB write performance (16 threads).

Point query. For RocksKV, a `get` operation requires examining several sorted-runs in each level of the LSM-tree to finally retrieve the records, introducing multiple I/Os. Wisckey needs to look up the LSM-Tree for the log offset of a record based on user key before retrieving the value from the log. In contrast, KVRangeDB without packing (KVR) can fulfill the `get` request by a single I/O using the user key through the KV interface provided by the device. Similarly to Wisckey, KVR-PK only requires traversing a small LSM-tree to translate the logical key to physical key and then retrieve the value from the device using the physical key. Hence, a small index cache is enough to help reduce the I/O overheads from index lookup.

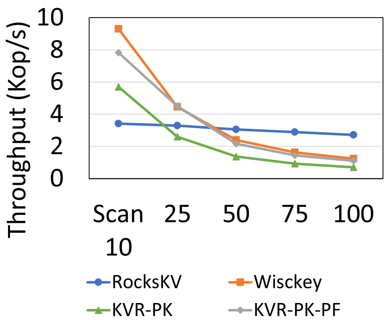
Figure 9(a) and (b) demonstrates the performance of simple `get` (or *point query*) workload. KVR exhibits a large advantage over RocksKV for both no cache and 1-GB cache scenarios. KVR outperforms Wisckey for large records by 73% (no cache) and 39% (1 GB cache). KVR-PK provides slightly



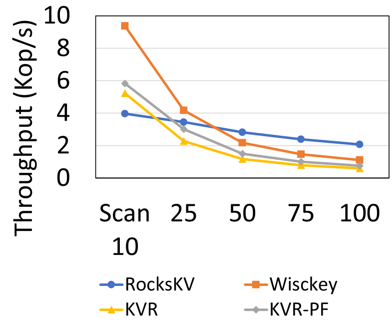
(a) Scan keys throughput for 1K value (16 threads)



(b) Scan keys throughput for 4K value (16 threads)



(c) Scan keys&values throughput for 1K value (16 threads)



(d) Scan keys&values throughput for 4K value (16 threads)

Fig. 10. YCSB range query performance.

lower performance than Wisckey with 1000 B value size, because the block device provides better read performance compared to KVSSD.

Scan keys. For the scan key workload, KVRangeDB only needs to traverse a relatively small LSM tree only containing keys. By contrast, RocksKV's LSM-Tree comprises both keys and values, which may require more I/Os. KVR-PK/KVR achieve much better performance, ~8× better compared to RocksKV with 1 GB cache as shown in Figure 10(a) and (b). KVR-PK/KVR perform slightly worse compared to Wisckey due to the device read performance disadvantage of KVSSD (Wisckey also only needs single I/O to retrieve value after locating the log offset).

Some may wonder if scanning the keys only (without retrieving values) makes sense in real-world applications. Here is an example of a typical file system workload (more details in Section 4.3): Consider the command line utility **ls**, which lists files and sub-directories. In TableFS, a **ls -l $path** command translates to a scan on the target directory that needs to retrieve value (calling both `key()` and `value()`) for parsing stats in the inode. However, a simple **ls $path** command only needs to iterate on the keys without reading the value (inode information).

Scan keys and values. On the flip side, KVRangeDB does not perform equally well with range queries that retrieve values, since it costs a separate I/O for each `value()` operation. As shown in Figure 10(c) and (d), when the scan length passes 40, KVR-PK-PF/KVR-PF perform worse than RocksKV. The optimization of value prefetch with user hints improves the performance to some extent (~56%). From the analysis of real key–value workloads [41], the average scan length is less than 20. Therefore, it may not be worth packing key and value together like RocksKV, which mostly benefits longer scans with value retrieval (`value()` operation).

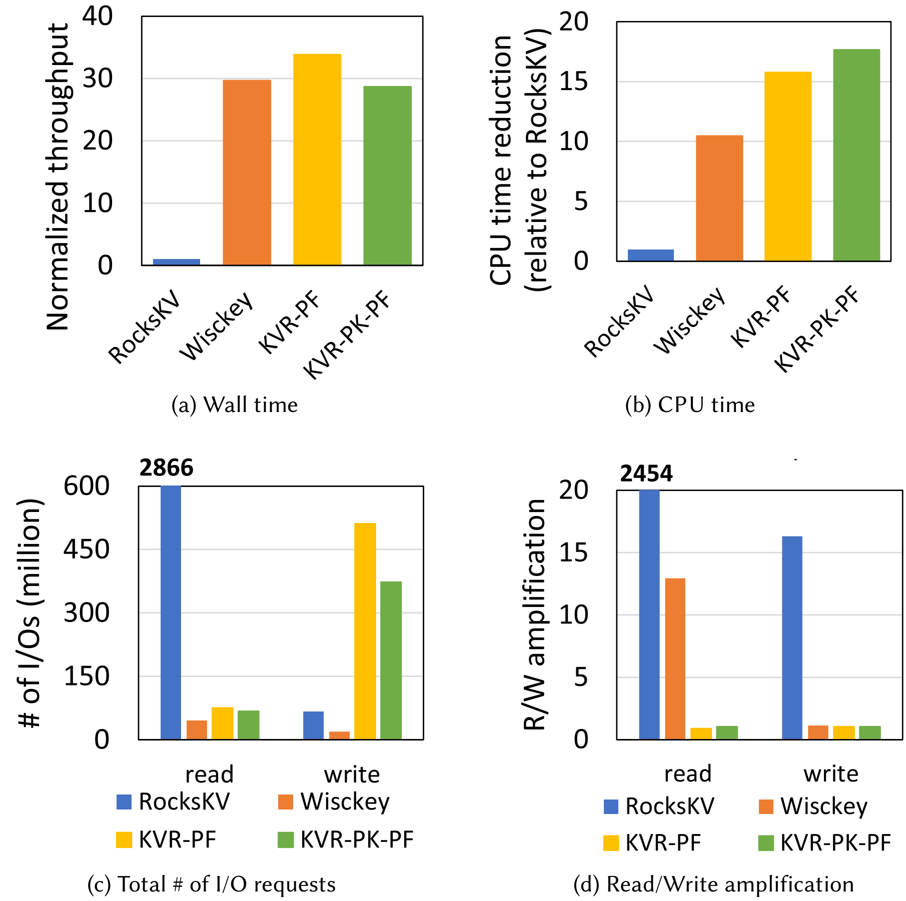


Fig. 11. Results for loading file system tree to TableFS.

4.3 Results for TableFS

For the file system workloads [2, 33, 37, 40], we use a real file system trace from Los Alamos National Lab that contains approximately 500 million files and directories (~20 million directories and ~480 million files), and ~90% files are marked as "cold," which can leverage our hybrid packing technique described in Section 3.2. The loading and aging phase consists of multiple file operations such as path resolve, opendir, mkdir, mkmod, unlink, chmod, and so on, which translates into a combination of `put`, `get`, `delete` workloads to the KV-store. At the end of each aging round, we perform a value log garbage collection for Wisckey (around 25% difference between real metadata capacity and actual storage usage). Value prefetching is enabled for range queries for both Wisckey and KVRangeDB variants.

For KVR-PK-PF, we selectively pack multiple file inode records (which are marked as cold set) under the same directory into a single physical record as described in Section 3.2. Since the files in the same directory are loaded together, such packing can benefit range queries as discussed in Section 3.2. For the remaining ~10% hot files, we do not perform packing, and the values (inode information) can be directly retrieved from the device through logical/application keys.

Load file system tree. Figure 11 presents the results of loading the file system tree into TableFS. KVR-PK-PF yields a 33.9× speedup over Rockskv and 1.14× over Wisckey, respectively. Besides,

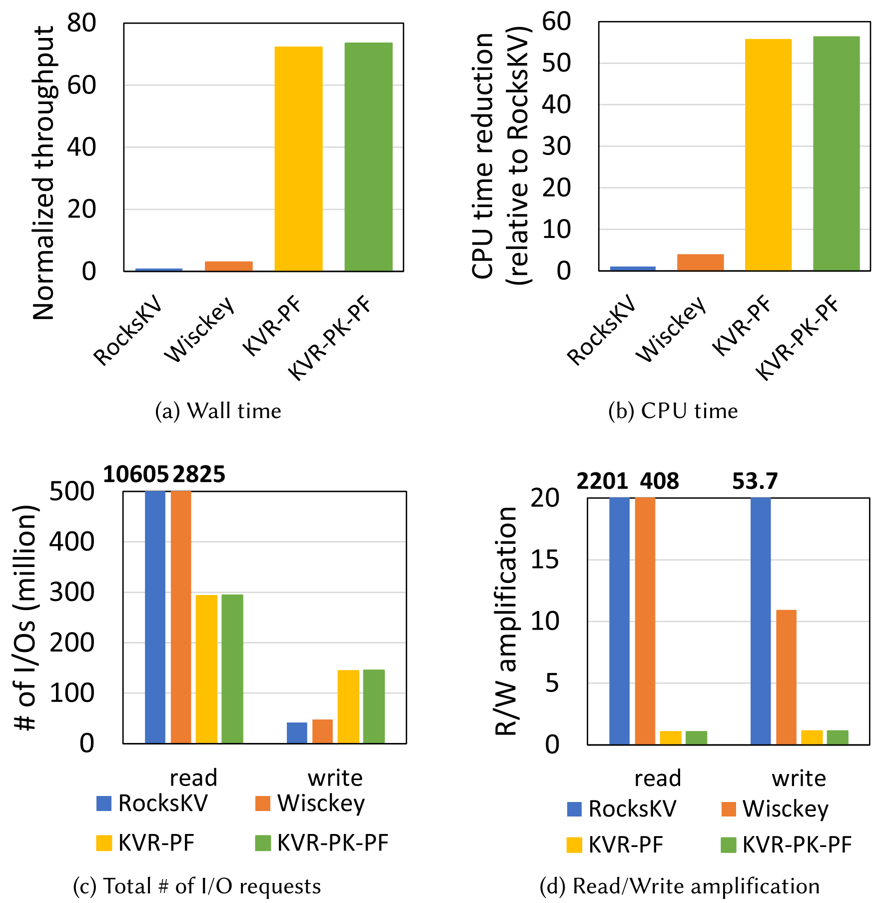
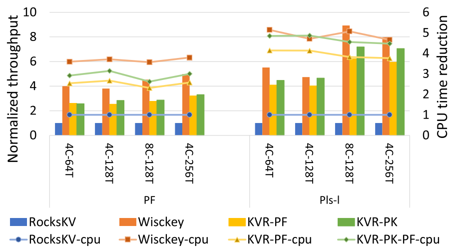


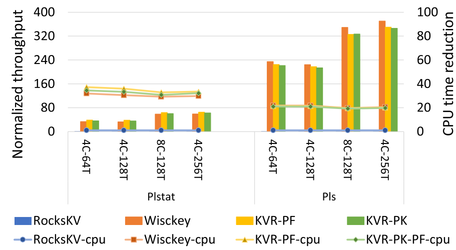
Fig. 12. Results for aging TableFS file system tree.

KVR-PK-PF also reduces CPU consumption by 15× and 1.5×, respectively. We also collect the number of I/O requests and read/write amplifications from/to the device. RocksKV incurs significantly larger write amplification, 15.7× worse than KVR-PK-PF, due to constant compaction of the sorted-runs. KVR-PK-PF also mitigates the read amplification enormously, specifically over 2000× fewer than RocksKV and 14× fewer compared to Wisckey, from the direct ***get*** interface on the device. KVR-PK-PF performs slightly worse than KVR-PF; however, it reduces CPU cost by 12% (due to less number of write I/Os).

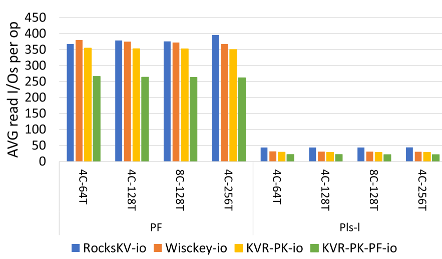
Aging the file system. Figure 12 demonstrates the results of aging the TableFS file system tree. KVR-PK-PF outperforms RocksKV and Wisckey by 72× and 23.7×, respectively. Moreover, KVR-PK-PF also saves CPU cost by 55.6× and 14.3×, respectively. The main negative factor of Wisckey is the value log garbage collection caused by records update [15, 23]. Wisckey issues a larger number of read I/Os, because it needs to lookup the key to log offset mapping for every get operation (check file path existence) and also performs garbage collection after removes and updates of the records. KVR-PK-PF greatly reduces read and write amplification by 385× and 9.8× compared to Wisckey. This advantage is mainly attributed to using the direct key value interface on the KV devices to store values that effectively offloads the value log garbage collection from the host to the device.



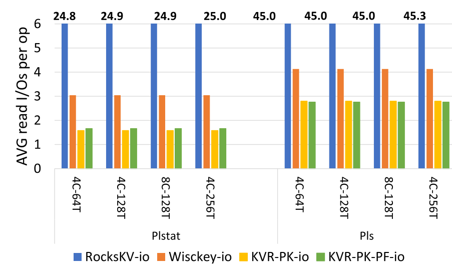
(a) Parallel find and parallel ls -l.



(b) Parallel lstat and parallel ls.



(c) Parallel find and parallel ls -l.



(d) Parallel lstat and parallel ls.

Fig. 13. Performance, CPU, and read I/Os for TableFS workloads (C and T on the x-axis denotes physical core and total thread counts).

Metadata-intensive operations. Figure 13 shows the performance and read I/O results for metadata-intensive file system workloads. We use a limited number of CPU resources (four and eight physical cores) to emulate the resource competition common in multi-tenant scenarios. We assign 16/32 client threads for each physical core.

Parallel **find** workloads perform traversal of the files/directories in a breadth first search fashion. These workloads contain path lookup and readdir operations that translate to `get` and range queries. KVR-PK-PF yields ∼5.1× better performance on average compared to RocksKV and reduces CPU cost by a factor of 3.9×. This is because in a real file system directory tree, there are lots of directories with very few sub-directories and files (leading to short scans). Wisckey outperforms KVR-PK-PF by ∼30%, simply because current block SSD has much higher read IOPS performance (∼3×) as shown in Table 2 and better latency characteristics [15].

Parallel "ls -l" contains path lookup and readdir operations that translate to `get` and range queries with both `key()` and `value()` operations with various scan lengths (depending on the number of files and sub-directories within a queried directory). KVR-PK-PF yields ∼5× better performance on average compared to RocksKV and reduces CPU cost by 4×. KVR-PK-PF slightly improves performance and reduces CPU consumption, since it reduces get I/O operations (∼10%) when the queried directory file inodes are packed.

Parallel **lstat** workload consists of `get` operations only. Compared to RocksKV and Wisckey that require multiple I/Os per `get` operation (RocksKV needs to examine multiple sorted-runs or SSTable files, Wisckey needs to lookup the log offset from user key before retrieving the value from the log), KVR only requires a single I/O per `get` through the KV device interface. Thus, KVR reduce 15.9× and 1.9×, respectively, compared to RocksKV and Wisckey. Besides, KVR outperforms RocksKV and Wisckey by 51× and 1.12× and reduces CPU usage by 30× and 1.15×. The file system workloads showcase the advantages of KVR, even with the current KVSSD read performance being relatively low compared to similar hardware block SSD. Despite the fact that KVR-PK-PF requires more than one I/O per `get` when keys need to be translated, its performance is barely affected under these workloads. To understand that, we analyze the workloads and found that most **lstat** operations are performed on hot file set whose keys do not need translation (application key equals physical key), thus KVR-PK-PF performs similarly to KVR-PF.

For simple parallel "ls" without "–l," which is converted to a range query without `value()` operation, KVR-PF performs 21× better compared to RocksKV. The cause of RocksKV's poor performance is that the SSTable packs key and value together, thus the cost of range queries only calling `key()` is similar to range queries that calls both `key()` and `value()`. KVR-PF, KVR-PK-PF, and Wisckey have similar performance, since they separate keys and values.

5 RELATED WORK

Key–value device interfaces has been a frequent topic of research, since the release of the first key–value device prototype [18]. Several researchers have explored command set extensions for key–value devices [15, 19, 35] and key–value device interfaces have also been explored as possible interfaces to SmartNICs [22] and persistent memory [10, 17]. Some recent work explored how to implement conventional block oriented storage features such as redundancy to key–value interfaced devices [29, 30].

The design of KVRangeDB, our range query facility for key–value devices, extends several techniques developed for LSM-based key–value databases to the key–value device interface. Wisckey [23] proposed the idea of separation of key and values to reduce write amplification during compaction and by storing values separately in a log. Zhang et al. proposed SuRF [39], which uses a compact trie structure as range query filter to accelerate range query performance on LSM tree-based software KV stores. KVRangeDB applies these techniques without requiring the use of an LSM tree or value-log for key–value storage devices.

6 CONCLUSION

In this article, we proposed and implemented KVRangeDB to support efficient range query capability on hash-based KVSSDs. Our design leverages a secondary key index based on log structure merged tree. With that we can optionally pack records through logical to physical key translation, so as to mitigate the key management overhead in the KVSSD device. We also employ user hints for value prefetching to accelerate scans with value retrieval. Moreover, we leverage the state-of-the-art range filter to efficiently improve empty range/point queries.

We evaluated our design with a series of real-world applications. Our results show that KVRangeDB provides faster *put*, *get*, short scans performance, and lower host CPU utilization compared to the state-of-art software KV engine (Wisckey) on conventional block SSD, although it may not be the optimal choice for workloads with extremely long scans with value retrieval.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. We appreciate Memory Solution Lab (MSL) in Samsung Semiconductor Inc. for granting us the access of the hardware to carry out the experiments.

REFERENCES

[1] 2020. NVMe Command Set Specifications. https://nvmexpress.org/.
[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Association for Computing Machinery, New York, NY, 353–369. https://doi.org/10.1145/3341301.3359656
[3] Apache. 2013. HBase. Retrieved from https://hbase.apache.org/.

[4] Jens Axboe. 2020. Key Value Storage API Specification—SNIA. Retrieved from https://www.snia.org/keyvalue.
[5] Matias Bjørling. 2020. Zone Append: A new way of writing to zoned storage. In *Linux Storage and Filesystems Conference (VAULT'20)*. USENIX Association, Santa Clara, CA.
[6] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Berkeley, CA, 359–374. https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.
[7] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692
[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX, Berkeley, CA, 49–60. https://www.usenix.org/conference/atc13/technicalsessions/presentation/bronson.
[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA.
[10] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, New York, NY, 1077–1091. https://doi.org/10.1145/3373376.3378515
[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 143–154. https://doi.org/10.1145/1807128.1807152
[12] J. Dean and S. Ghemawat. 2017. LevelDB: Google's fast key value store library. Github Release 1.2.
[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *SIGOPS Operat. Syst. Rev.* 41, 6 (October 2007), 205–220. https://doi.org/10.1145/1323293.1294281
[14] Facebook. 2015. Rocksdb. Retrieved from https://rocksdb.org/.
[15] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, Berkeley, CA, 173–187. https://www.usenix.org/conference/atc20/presentation/im.
[16] Y. Jin, H. Tseng, Y. Papakonstantinou, and S. Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 373–384. https://doi.org/10.1109/HPCA.2017.15
[17] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Berkeley, CA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet.
[18] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*. ACM, New York, NY, 144–154. https://doi.org/10.1145/3319647.3325831
[19] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. 2019. Transaction support using compound commands in key-value SSDs. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/kim.
[20] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic.
[21] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. 2021. Modernizing file system through in-storage indexing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. USENIX Association, Berkeley, CA, 75–92. https://www.usenix.org/conference/osdi21/presentation/koo.
[22] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, New York, NY, 137–152. https://www.microsoft.com/en-us/research/publication/kv-direct-high-performance-memorykey-value-store-programmable-nic/.

[23] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Berkeley, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

[24] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Association for Computing Machinery, New York, NY, 2071–2086. https://doi.org/10.1145/3318464.3389731

[25] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, Berkeley, CA, 207–219. https://www.usenix.org/conference/atc15/technical-session/presentation/marmol.

[26] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[27] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, Berkeley, CA, 217–231. https://www.usenix.org/conference/fast21/presentation/pan.

[28] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquín, and Donald Kossmann. 2017. Fast scans on key-value stores. *Proc. VLDB Endow.* 10, 11 (August 2017), 1526–1537. https://doi.org/10.14778/3137628.3137659

[29] Rekha Pitchumani and Yang-Suk Kee. 2020. Hybrid data reliability for emerging key-value storage devices. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, Berkeley, CA, 309–322. https://www.usenix.org/conference/fast20/presentation/pitchumani.

[30] Mian Qin, A. L. Narasimha Reddy, Paul V. Gratz, Rekha Pitchumani, and Yang Seok Ki. 2021. 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 (SYSTOR'21)*. Association for Computing Machinery, New York, NY, Article 3, 12 pages. https://doi.org/10.1145/3456727.3463781

[31] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 497–514. https://doi.org/10.1145/3132747.3132765

[32] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, 145–156. https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren.

[33] K. Ren, Q. Zheng, S. Patil, and G. Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*. 237–248. https://doi.org/10.1109/SC.2014.25

[34] CMU/PDL File Systems. 2013. Fast and Efficient Filesystem Metadata through LSM-Trees. Retrieved from https://github.com/pdlfs/tablefs/.

[35] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article 16, 14 pages. https://doi.org/10.1145/2592798.2592804

[36] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based ordered key-value store using remote learned cache. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, Berkeley, CA, 117–135. https://www.usenix.org/conference/osdi20/presentation/wei.

[37] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 307–320. http://dl.acm.org/citation.cfm?id=1298455.1298485.

[38] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, Berkeley, CA, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang.

[39] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the International Conference on Management of Data (SIGMOD'18)*. Association for Computing Machinery, New York, NY, 323–336. https://doi.org/10.1145/3183713.3196931

[40] Qing Zheng, Charles D. Cranor, Danhao Guo, Gregory R. Ganger, George Amvrosiadis, Garth A. Gibson, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2018. *Scaling Embedded In-Situ Indexing with DeltaFS*. IEEE Press.

[41] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, Berkeley, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao.

Received 21 June 2022; revised 11 October 2022; accepted 9 January 2023