Iterator Interface Extended LSM-tree-based KVSSD for Range Queries

Seungjin Lee[1], Chang-Gyu Lee[1], Donghyun Min[1], Inhyuk Park[2], Woosuk Chung[2]
Anand Sivasubramaniam[3], Youngjae Kim[1*]
[1]Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
[2]Memory System Research, SK hynix
[3]Dept. of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, USA

ABSTRACT

Key-Value SSD (KVSSD) has shown great potential for several important classes of emerging data stores due to its high throughput and low latency. When designing a key-value store with range queries, an LSM-tree is considered a better choice than a hash table due to its key ordering. However, the design space for range queries in LSM-tree-based KVSSDs has yet to be explored, despite range queries being one of the most demanding features. In this paper, we investigate the design constraints in LSM-tree-based KVSSDs from the perspective of range queries and propose three design principles. Based on these principles, we present *IterKVSSD*, an Iterator interface extended LSM-tree-based KVSSD for range queries. We implement *IterKVSSD* on OpenSSD Cosmos+, and our evaluation shows that it increases range query throughput by up to 4.13× and 7.22× for random and sequential key distributions, respectively, compared to existing KVSSDs.

CCS CONCEPTS

• **Computer systems organization** → **Firmware**; • **Information systems** → **Storage management**.

KEYWORDS

Log-structured Merge-tree, Key-Value SSD, Range Query

ACM Reference Format:
Seungjin Lee[1], Chang-Gyu Lee[1], Donghyun Min[1], Inhyuk Park[2], Woosuk Chung[2] and Anand Sivasubramaniam[3], Youngjae Kim[1]. 2023. Iterator Interface Extended LSM-tree-based KVSSD for Range Queries. In *The 16th ACM International Systems and Storage Conference (SYSTOR '23), June 5–7, 2023, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3579370.3594775

1 INTRODUCTION

Key-Value SSD (KVSSD) is a new type of SSD that uses keys to store and retrieve values, with a new set of Key-Value operations such as Put, Get, Delete, and Scan. With the new Key-Value interface directly integrated into storage, which is fundamentally different from the Logical Block Address (LBA) interface used in conventional SSDs, KVSSDs offer benefits such as a shortened host-side I/O stack by removing the Key-Value store and file system, resulting in reduced I/O to the device. These benefits of KVSSD introduce numerous KVSSD studies [2, 6, 12–16, 19, 22, 26, 28] and standardization efforts [23, 27].

Range query is a core operation used in many database applications [1, 3, 7, 29] to retrieve Key-Value pairs belonging to a specific key range. Therefore, it is vital for KVSSDs to efficiently support range queries. Typically, a range query is served via an iterator that consists of Seek() and Next(). An iterator locates the start of a certain key range using Seek() and retrieves successive Key-Value pairs by repeating Next() operations. As range queries essentially require an ordered data structure, the LSM-tree [24] was considered a natural fit, resulting in its adoption by a large number of KVSSDs [6, 12, 16, 19, 26].

The current state-of-the-art LSM-tree-based KVSSD with an iterator interface is PinK [12]. PinK proposes pinning index (SSTable) at the top k level to minimize tail latency caused by NAND flash accesses. However, PinK's index pinning technique cannot resolve long tail latency issues when NAND flash accesses to SSTables at a deeper level occur. Moreover, PinK only presents basic iterator interfaces that can cause serious problems, such as inconsistent concurrent range queries and performance issues.

In particular, range queries through an iterator interface are served over multiple I/O commands, so it is possible that some Key-Value pairs will be modified, deleted, or created in the middle of range queries. As a result of this structural modification of the LSM-tree, the iterator may return inconsistent and unexpected results for the range query, such as missing keys or even skipping them.

Additionally, PinK focuses on optimizing index reads, which is not the dominant factor for the performance of range queries. For the LSM-tree with Key-Value separation, the index size to read is much smaller than the values, so most of the time is spent accessing NAND flash to read values synchronously. Considering that NAND access time is hundreds of times slower than DRAM access time [17], even a single extra NAND access is critical for KVSSD performance. Furthermore, the index pinning technique can reduce index read times to a certain degree, but it may still require synchronous index reads because some parts of the index cannot be pinned in memory due to its immense size.

To tackle the above problems, we propose *IterKVSSD*, an Iterator interface extended LSM-tree-based KVSSD. *IterKVSSD* suggests three techniques to resolve the aforementioned problems of the

*Y. Kim is the corresponding author.

SYSTOR '23, June 5–7, 2023, Haifa, Israel

ACM ISBN 978-1-4503-9962-3/23/06…$15.00
https://doi.org/10.1145/3579370.3594775

state-of-the-art LSM-tree-based KVSSD. For the inconsistent range query problem, *IterKVSSD* implements memory-efficient versioning via metadata pooling, which decouples the actual SSTable metadata from the summary data structure, to reduce the memory footprint used in each version. *IterKVSSD* also adopts a prefetch method for SSTables (index) and values to mitigate the synchronous NAND access costs. This takes advantage of the fact that the KVSSD is aware of the physical location of values on NAND flash memory. Furthermore, internal parallelism, due to the independently operating NAND Flash Channel Controllers (NFCs), makes index and value prefetch feasible while serving foreground I/O requests in parallel.

Overall, we make the following contributions:

- We define three design principles for iterator support in LSM-tree-based KVSSD based on design space exploration.
- We present the design of a memory-efficient versioning mechanism via metadata pooling.
- We perform a mathematical analysis of the critical I/O path in processing Seek() and Next(), focusing on the NAND flash penalty.
- Based on the analysis of the critical path, we show that index prefetch and value prefetch can effectively exploit internal parallelism and hide the NAND flash penalty in iterator operations.

We implemented *IterKVSSD* on the OpenSSD Cosmos+[17], which is an LSM-tree-based KVSSD. Through an extensive evaluation using db_bench[8], we demonstrate that *IterKVSSD* is memory-efficient and robust in handling a wide range of concurrent range queries. In particular, our evaluation results demonstrate that *IterKVSSD* achieves 4.13× and 7.22× higher range query throughput on KVSSDs with random and sequential key orders, respectively, compared to the case without prefetching.

2 BACKGROUND & PRIOR WORK

2.1 LSM-tree-based Key-Value Store

LSM-tree [24] is one of the most widely adopted data structures in modern Key-Value stores [5, 8, 11, 18] that demand high write throughput. Figure 1 depicts the architecture of an LSM-tree-based Key-Value store. LSM-tree keeps both MemTable and summary data structures in memory. In persistent media, LSM-tree manages SSTables (Sorted String Table) that essentially form the levels of the LSM-tree. Also, LSM-tree has a compaction procedure to create and organize SSTables. Note that a summary data structure manages how to construct an entire LSM-tree with SSTables.

LSM-tree handles Key-Value operations as a record, where a record is identified with its key, but its contents can be interpreted as a value for the key or the key's deletion. For simplicity, we assume only insertion. But note that there is not much difference because the record for deletion also has to be searched in the LSM-tree as other existing keys. MemTable can hold records up to a predefined size. When it gets full, the MemTable is flushed to level 0 as an SSTable, and then the LSM-tree starts over with a new empty MemTable. An SSTable is a block-based format that stores a set of records with their index. Collectively, a set of SSTables forms a level in the LSM-tree. Except for SSTables in level 0, SSTables in level i partition the key extents of level i. For example, a level 1 consisting of SSTables covering key extents $(42-56)$ and $(60-125)$,

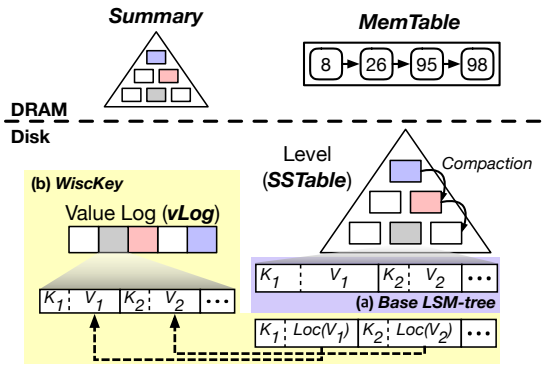


Figure 1: Overall Architecture of LSM-tree.

is a valid level. However, SSTables covering key extents $(42-56)$ and $(43-56)$ can not form level 1.

The compaction procedure enforces this property except for level 0. Each level in LSM-tree has a limit on the total size of SSTables belonging to a level. The size limit typically increases about ten times as the level goes deeper, e.g., level 0 has 64 MB, level 1 has 640 MB, and so on. When level i exceeds its size limit, the compaction procedure takes action to fit level i in the size limit. The compaction procedure picks victim SSTables in level i, merges them with SSTables with an overlapped key range in level $i+1$, then creates new SSTables in level $i+1$, deleting the victim SSTables in level i. The merging during compaction is crucial for the LSM-tree because it replaces obsolete keys in level $i+1$ with more recent keys from level i. Since the MemTable flushing and compaction procedures continuously transform the LSM-tree organization over time, LSM-tree has a summary data structure that tracks which SSTables form each level and also keeps brief metadata about SSTables, which consists of the minimum and maximum keys of each SSTable.

Key-Value Separation: Meanwhile, keys that are not deleted by newer records are repeatedly copied as the record moves down to the last level. These long-lived records contribute significantly to I/O amplification during compaction, which leads to performance degradation [21]. To solve this, WiscKey [20] proposed a Key-Value separation technique that decouples actual value data from a Key-Value pair in SSTables via indirection to the value log (vLog). WiscKey appends a Key-Value pair to the end of vLog and inserts the new record into the MemTable with the key and the offset in vLog. Simple indirection using the vLog offset greatly reduces the I/O amplification during the compaction procedure since small vLog offsets are copied instead of the actual value.

2.2 Iterator Interface in LSM-tree

Iterator interface is one of the common ways to provide range queries. Because LSM-tree maintains the ordering of keys, the iterator interface can be implemented more easily on LSM-tree than other non-ordered data structures. Specifically, the LSM-tree can be considered as a tree composed of small ordered data structures because MemTable and SSTables both organize keys in numerical order. When we search from MemTable, the uppermost part of the tree, we can find the keys that reflect the most recent update while all keys are in numerical order.

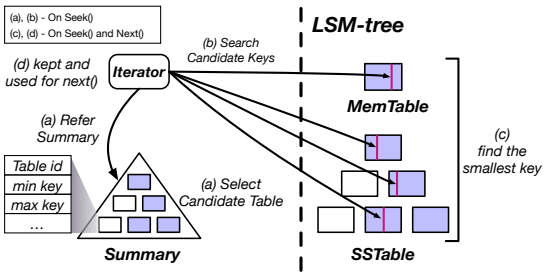


Figure 2: Description of Seek() and Next().

Seek() and Next() are the two essential operations in the iterator interface. The Seek() operation positions an iterator at the key_{start}. When LSM-tree may not have exactly key_{start}, the iterator will be positioned at the smallest key that is greater than key_{start}. To position the iterator at key_{start}, the LSM-tree uses the summary data structure. By selecting SSTables with key extent covering key_{start} from all levels, the LSM-tree can have a complete candidate set of SSTables. Then, the LSM-tree can pick the smallest key from the search result of key_{start} from the set of SSTables and the MemTable. It is possible that the same keys may be found across multiple levels from the candidate SSTables (and MemTable) because obsolete records are alive from the LSM-tree until compaction reclaims it. If the same key is found in multiple levels, the LSM-tree takes the Key-Value pair from the uppermost level as the most recent record.

Next() is used to continue traversing Key-Value pairs in a sequential manner starting from key_{start}. To this end, the LSM-tree keeps the set of SSTables (and MemTable) collected from the previous Seek() operation to search for the next key. More specifically, the LSM-tree keeps the exact position in each index of SSTables and MemTable so that it can simply move to the next item and compare to get the appropriate key following key_{start}.

Figure 2 describes how an LSM-tree works in the case of a Seek() followed by a Next(). (a) Handling Seek() starts with selecting candidate SSTables and MemTable (simply candidate tables hereafter) by referring to the minimum and maximum key of each SSTable in the summary data structure. (b) Candidate tables are searched for the starting key or the smallest key greater than the specified starting key. Note that MemTable and SSTables are all ordered data structures, and along with the keys Seek() found (candidate keys), it also keeps their position as indexes in each table. (c) Next, the smallest key is selected from the candidate keys. When identical keys are found in multiple levels, the key from the uppermost level is selected. In this case, MemTable is considered as the upper level of level 0 (i.e., level -1) since MemTable always holds the most recent records. (d) Then with Next(), the iterator utilizes previous candidate tables to find the next key following the starting key. Using candidate tables, the candidate keys are again selected and compared by searching each table for the smallest key greater than the starting key.

2.3 LSM-tree-based Key-Value SSD

For the past years, KVSSDs have been proposed to reduce I/O amplification resulting from host I/O stack [6, 12, 13, 16, 19, 25, 28]. Due to their range query capabilities, LSM-tree-based KVSSDs[1] gain attention among those KVSSDs. Furthermore, given the limited memory capacity of KVSSDs, Key-Value separation has also been actively discussed to reduce memory footprints for compaction operations and further reduce I/O amplification [12, 16, 19].

KVSSD shares most of the design shown in Figure 1, except for the way persistent components, such as SSTables, are stored. Since the file interface is no longer available inside the SSD, KVSSDs must take on a portion of the file system's roles to determine the physical layout of persistent components, so that SSTables and vLog can be safely persisted and loaded. The summary data structure plays a critical role for this, which we refer to as simply the *Summary* from here on. Each entry in the *Summary* stores NAND page addresses that indicate the physical location of SSTable, along with other attributes such as the minimum and maximum keys. Therefore, the LSM-tree can be reconstructed on device reset by loading the *Summary* and SSTables in order.

2.4 State-of-the-art KVSSD with Iterator Interface and its Limitations

PinK is a state-of-the-art LSM-tree-based KVSSD that implements an iterator interface. In order to reduce tail latency caused by loading indexes and filters, PinK proposed pinning indexes at the top few levels (L0, L1, L2 SSTables) in memory. However, PinK only provided a basic iterator interface without considering multiple issues that may lead to serious problems, such as inconsistent range queries, tail latency, and poor performance issues.

Range queries via iterator are typically composed of one Seek() and multiple Next() calls. Since a range query is performed across multiple calls that can be interleaved with other Key-Value operations, it is possible that Put() or Delete() operations followed by compaction may change the structure of the LSM-tree. This structural modification includes the change of *Summary*, and consequently, the iterator created before compaction may not be able to find SSTables (Index) that were deleted by the compaction process. Since iterator objects in most database applications [8, 11] are intended to see the database at its creation time, it is essential for KVSSDs to support versioning for consistent range queries. However, we believe that PinK fails to address this inconsistent range query problem. In this regard, we believe that PinK fails to address this inconsistent range query problem.

Furthermore, while pinning high-level SSTables can eliminate tail latency caused by NAND flash reads for those SSTables, the pinning approach still requires NAND flash accesses for reading deeper-level SSTables. This means that even with the pinning technique, synchronous index reads can still occur, potentially resulting in tail latency issues during the execution of range queries.

Lastly, PinK uses huge amounts of memory to pin the index in memory, but using too much memory for pinning or caching indexes is undesirable for range query performance. It is because PinK adopts Key-Value separation where the size of the index to read for Seek() and Next() is much smaller compared to the size of the value. As we will show in Section 5, the key to optimizing the overall performance of range queries is to avoid synchronous NAND flash access for value reads as much as possible. However, PinK did not consider optimizing synchronous NAND flash access for value reads, which may result in poor range query performance.

[1] Hereafter, we will use the terms LSM-tree-based KVSSD and KVSSD interchangeably.

To resolve these problems, a memory-efficient design must be applied for KVSSDs in order to minimize synchronous NAND accesses as well as version control. In this paper, we formulate the design principles for an iterator interface inside the device, and then we show an example design strictly abiding by the principles in the following sections.

3 DESIGN PRINCIPLE

To address the aforementioned problems in designing an iterator interface in KVSSDs, we conceive the following design principles: **P1**, **P2**, **P3**.

P1. Memory-efficient Versioning Support To enable versioning inside the device, an iterator needs to manage the minimum information about the version at its creation time. It is also important for KVSSDs to ensure that the corresponding index and value remain available until the iterator is destroyed. To address these requirements, *IterKVSSD* manages LSM-tree's state with *Summary*, which provides access to SSTable metadata. Since multiple iterators with the same version share most of the SSTable metadata, we propose a metadata pooling method that decouples the SSTable metadata from the summary data structure in memory. Furthermore, *IterKVSSD* keeps track of reference counters of each iterator, which ensures SSTables used by range queries are alive until the end of each query.

P2. Minimize Cost of Index (SSTable) Read There are two possible cases where NAND flash is accessed to read index (SSTables). First, when processing the Seek() command, an iterator is required to read some of the tables covering the smallest key at each level, which may incur multiple NAND flash reads. This process is unavoidable unless all candidate SSTables are cached in device memory. The second case happens when processing the Next() command. While the iterator moves forwards it is possible that the iterator reaches the end of the buffered SSTables. At this moment, Next() is responsible for loading the next SSTable with the succeeding key range from the same level. Consequently, the iterator may encounter a potential NAND read during the process of Next().

To minimize memory footprint without having much cache for index, *IterKVSSD* exploits the sequential key access pattern of range queries and also the parallelism of NAND flash controllers (NFC) running independently. One thing to note is that Key-Value separation alleviates the NAND access for index read by Next(). Because Key-Value separation reduces the size of each entry in an SSTable by replacing the value to vLog offset, the number of keys in a single SSTable can be greatly increased compared to a KVSSD without Key-Value separation in an SSTable of the same size. This increases the expected number of Next() required to encounter the above situation which incurs a potential NAND read.

P3. Minimize Cost of Value Read All Seek() and Next() commands need to return a value to the host from the vLog. However, values of adjacent keys may not be written contiguously in vLog due to Key-Value separation. Without Key-Value separation, both keys and values are sorted and written in NAND flash together during compaction. This is not the case with the Key-Value separation, as values are not stored with their corresponding keys, and it is, therefore, hard to expect spatial locality in NAND flash, even though iterators access sequential keys. This randomness of value location in vLog woefully deteriorates the throughput of the range query.

In this case, an extensive read cache may seem to be a solution, but due to the device's memory restriction. *IterKVSSD* adopts a prefetching method to read values by exploiting the sequential access pattern of iterators to tackle this problem. Additionally, *IterKVSSD* makes full use of the internal parallelism from independently working NAND flash controllers to prefetch vLog entries.

4 ITERATOR INTERFACE FOR KVSSDS

In this section, we propose *IterKVSSD*, an LSM-tree-based KVSSD strictly following the three design principles. We first explain the overall architecture of *IterKVSSD* and address the solutions to meet each design principle.

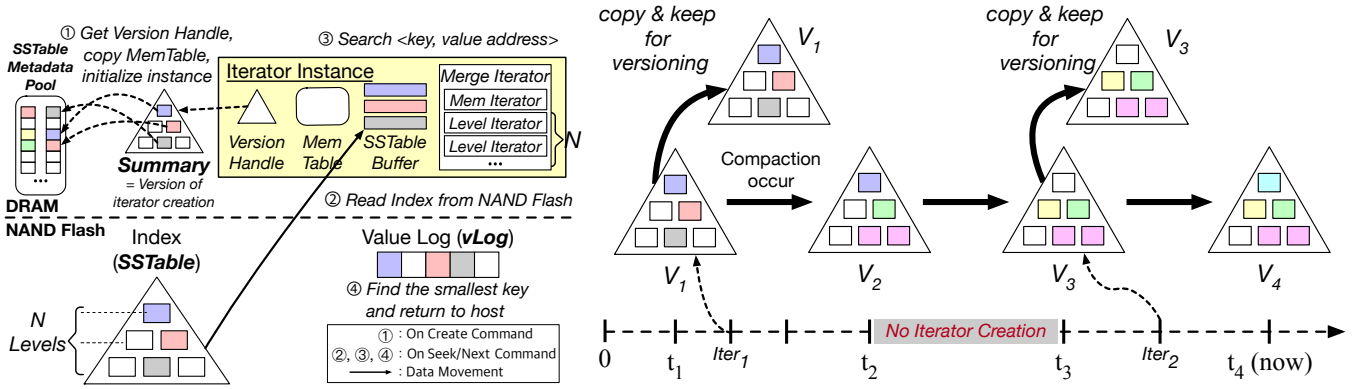
4.1 Design of Iterator Interface

Figure 3(a) shows the main data structure for an iterator instance. When *IterKVSSD* receives a range query request, *IterKVSSD* creates an iterator instance and stores the version information of the LSM-tree in DRAM. The iterator instance consists of six components as follows (yellow rectangle in Figure 3(a)).

- **Version Handle:** The pointer to *Summary* at the time the iterator was created.
- **MemTable:** The copy of the MemTable at the time the iterator was created.
- **Mem Iterator:** Mem Iterator points to the key currently being traversed within the MemTable.
- **Level Iterator Array:** Each array element points to the current key being traversed within the SSTable of each level in the LSM-tree. Each level iterator is updated with a new key when iterating to the next.
- **Merge Iterator:** Store the smallest key among all the internal Iterators (Mem Iterator and Level Iterators).
- **SSTable Buffers:** Buffer for SSTables. By default, one SSTable buffer is allocated for each level.

Version handle and the copy of the MemTable serve as version information for range queries. The versioning mechanism will be described in detail in Section 4.2. These two data structures ensure that iterators see the KVSSD version as of their creation time, even when the LSM-tree changes due to upcoming compactions. Additionally, an iterator instance also has a working space (SSTable Buffer) for searching and traversing indexes.

Figure 3(a) depicts the operation flow of the iterator for a range query. A range query can send commands such as Create(), Seek(), Next(), Destroy() to *IterKVSSD*. ① When *IterKVSSD* receives a Create() command, *IterKVSSD* creates an iterator instance with the aforementioned data structures. After completing the creation of the iterator instance, *IterKVSSD* sends a completion command to the host with an iterator-specific ID. ② Afterwards, the host can initiate a range query by issuing a Seek() command with the iterator ID and the start key of the range ($\geq key_{start}$). On Seek(), *IterKVSSD* first finds the candidate SSTables covering the search range by consulting its version of *Summary*, and *IterKVSSD* reads the first SSTables of the candidate tables at each level. *IterKVSSD* can determine which SSTable contains the first key in the search



(a) Description of the data structure and the behavior of an iterator instance on *IterKVSSD*

(b) Description of management of Summary that occurs when versioning is supported in *IterKVSSD*

Figure 3: Description of the architecture of an iterator instance on a *IterKVSSD*.

range of each level and find out the physical address of the table on NAND flash from *Summary*. ③ Each of the level iterator array and Mem Iterator begin to search for sorted keys to find the Key-Value address pair with the smallest key within the search range. ④ After all internal iterators find its minimum key, the Merge Iterator compares them to find the Key-Value address pair with the smallest key. The Merge Iterator advances the iterator with the smallest key, pointing to the next Key-Value pair. Finally, *IterKVSSD* reads the value of the Key-Value pair found by the Merge Iterator from the vLog, returns the value to the host, and completes the Seek() command.

A user may continue the range query by issuing Next() with the iterator ID. When *IterKVSSD* receives a Next() command, it checks the Level and MemTable Iterators of the iterator instance and finds the Key-Value address pair with the smallest key. After that, it behaves similarly as described in Seek() above. However, if any of the level iterators reaches the end of an SSTable, the next SSTable has to be read from NAND flash to point to the next Key-Value address pair. Finally, *IterKVSSD* reads the value from vLog with the Key-Value address pair found by the Merge Iterator and sends it to the host, completing the Next() command. When the range query is done, the user can release the Iterator by specifying its ID with the Destroy() command.

4.2 Memory Efficient Versioning Support

As mentioned in Section 3, range queries with the iterator interface can be interleaved with Put() and Delete() commands, which may trigger compaction and change the state of the LSM-tree. However, an application that created an iterator instance before even compaction cannot be aware of what happens inside the device. Therefore, to give iterator instances inside the device their own view of the entire database, we propose a memory-efficient versioning method.

Keeping a specific version of *Summary* is crucial in guaranteeing a consistent view of the LSM-tree that matches the creation time of an iterator. However, since *Summary* contains every SSTable metadata of the entire database, keeping the size of *Summary* as small as possible is the key to minimizing the memory footprint of the version. Moreover, copying SSTable metadata on every iterator creation results in duplication since compaction modifies only a fraction of the many SSTables that compose the LSM-tree.

To minimize the size of *Summary*, *IterKVSSD* decouples actual SSTable metadata from *Summary* and keeps them in a pool. In the pool, each SSTable metadata stores its table ID, physical location, reference counter, and attributes, such as its minimum and maximum key. This way, *Summary* only needs to keep pointers to each entry in the pool.

Figure 3(b) shows how versioning works inside *IterKVSSD* in a memory-efficient way. First, *IterKVSSD* keeps one global *Summary* (V_1) for the up-to-date LSM-tree. If an iterator instance ($Iter_1$) is created, the instance will have the Version Handle to V_1. If compaction happens while V_1 is referenced by $Iter_1$, V_1 is copied and kept for $Iter_1$. Afterwards *IterKVSSD* continues managing the up-to-date global *Summary* (V_2) reflecting the changes made by compaction. The reference counter in each SSTable metadata keeps track of how many versions are using the corresponding SSTable. This mechanism prevents the corresponding SSTable from being discarded.

We now show an example of a memory footprint analysis for an iterator instance. In our implementation, the size of the MemTable is 48 KB. Since each internal iterator only stores the value offset, size, and key, the total memory space required for all internal iterators is only tens of bytes. Therefore, the size of the iterator instance is mainly bound to the *Summary* and SSTable buffers. For a more detailed analysis, suppose a situation where the DB/LSM-tree is populated with 25 M ×4 KB value Key-Value pairs (100 GB). To keep things simple, let us say SSTables are 48 KB in size for all levels. In this case, one SSTable can store 4 K (key, value address) tuples (12 B for each), and the LSM-tree requires a total of 6,400 SSTables to store 100 GB data. Considering that *Summary* is a tree of pointers (4 B) to metadata entries in the pool, the total size of *Summary* is about 25 KB. To sum up, for a DB of 100 GB, the storage space size for LSM-tree is 300 MB (6,400 ×48 KB), while *Summary* only requires 25 KB, making the size of *Summary* about 0.008% of the LSM-tree. We consider this ratio a reasonably small memory footprint, even if we take into account the size of the pool.

4.3 Critical Path and Serialization Points

We now examine the critical I/O paths of Seek() and Next() to minimize the NAND access penalty. *IterKVSSD* executes the following steps when processing Seek() and Next() commands.

- **(a) Memory Reference and Computation:** For Seek(), an iterator finds the candidate SSTables, and for Next(), the iterator checks if the internal iterator reaches the end of the buffered SSTables.
- **(b) Index (SSTable) read:** If an internal iterator points to the end of an SSTable, the iterator reads the next table from NAND flash.
- **(c) Increment and compare:** Moves the iterator's cursor to the next. Then compares the key found in the MemTable with the keys found from each level iterator to select the minimum key.
- **(d) Value read:** Reads the value from vLog with the value address of the key found in Step(c).
- **(e) Return to host:** Returns the value read from vLog to the host.

In the whole process, Steps(b)&(d) are the main causes of a bottleneck when NAND flash accesses happen synchronously. To minimize the NAND flash access penalty, we propose a prefetching method by utilizing the following two device characteristics. The first is the fact that Key-Value semantic is enabled in the KVSSD. This means that *IterKVSSD* knows exactly which Key-Value pair (index and value) to read and where it is stored. The second one lies in the device's internal parallelism from multiple independent NAND flash controllers (NFCs). Since each NFC is running independently of the device CPU, it can access NAND flash while the CPU processes other tasks. Through these two characteristics, *IterKVSSD* minimizes synchronous NAND flash access time by fetching index and value into memory before they are used.

4.4 Index Prefetch

Figure 4(a) depicts the synchronous NAND access problem due to Step(b) on Next(). In Figure 4(a), the CPU in KVSSD executes Step(a). If it finds out that the end of the cached SSTable has been reached, it requests NFC1 to read the next SSTable. Here, in KVSSD, the CPU will block until the next SSTable read by NFC1 is completed.

To avoid blocking, *IterKVSSD* reads the SSTable ahead of time (Index Prefetch). Figure 4(b) describes the Index Prefetch process. When the CPU requests NFC1 to read the value in Step(c), the CPU checks whether to read the next SSTable in advance and, if necessary, requests NFC2 to read the next table. As a result of this index prefetch, the NAND flash access penalty associated with synchronous index reading is virtually eliminated. Of course, the NFC which is in charge of prefetching may be busy. In that case, the effect of Index Prefetch will decrease due to channel conflict. This channel conflict problem is fundamentally a data allocation problem which we leave as future work.

Effect of Index Prefetch: We now analyze the performance gain through Index Prefetch. Suppose a user creates an iterator for range query, calls Seek() once and Next() N times, and then destroys the iterator after the range query.

Since each command is performed sequentially, the execution time of range query (RQ) T_{RQ} can be expressed in the form of a linear combination of the time required for each operation: $T_{RQ} = T_{create} + T_{seek} + N \cdot T_{next} + T_{destroy}$. T_{next} can vary with different execution flows depending on whether Step(b) is executed or not. Suppose the frequency of occurrence of Step(b) during the execution of range query is f. Then, $T_{next} = T_{Step(a)} + \frac{1}{f} \cdot T_{Step(b)} + T_{Step(c)} + T_{Step(d)} + T_{Step(e)}$. f is bound to the number of (key, value address) tuples stored in a SSTable at the i level and the height of the LSM-tree. In other words, if more Key-Value pairs are buffered in memory, Step(b) is less likely to happen during the range query.

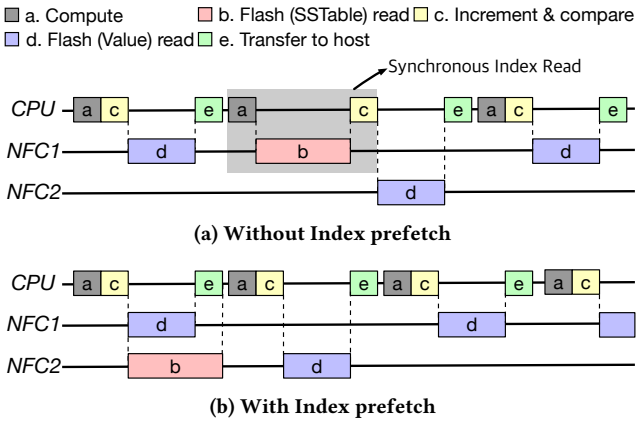


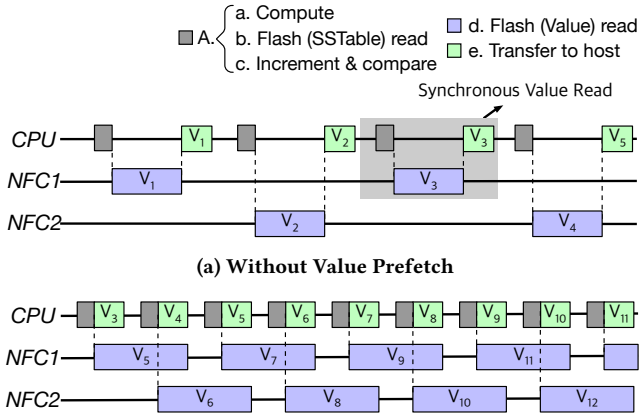
Figure 4: Description of Index Prefetch.

With Key-Value separation, the size of Key-Value address pair is 12 B. If we align the size of SSTables (48 KB in our setup) with the NAND flash page size (16 KB in our setup), 4 K Key-Value address pairs can be stored in one SSTable. That is, considering the multiple levels in the LSM-tree, f is typically more than a few thousands which means Step(b) happens once while processing thousands of Next(), which is relatively big compared to real world cases [3]. As described above, the effect of the Index Prefetch on the total time T_{RQ} is insignificant. However, the rare occurrence of Step(b) can introduce tail latency of Next() operations. Therefore, Index Prefetch is expected to be effective in alleviating the tail latency problem.

4.5 Value Prefetch

Figure 5(a) shows where the device CPU is blocked while *IterKVSSD* reads the value during Step(d), leading to decreased range query throughput. To solve this problem, we propose the idea of reading values in advance (Value Prefetch).

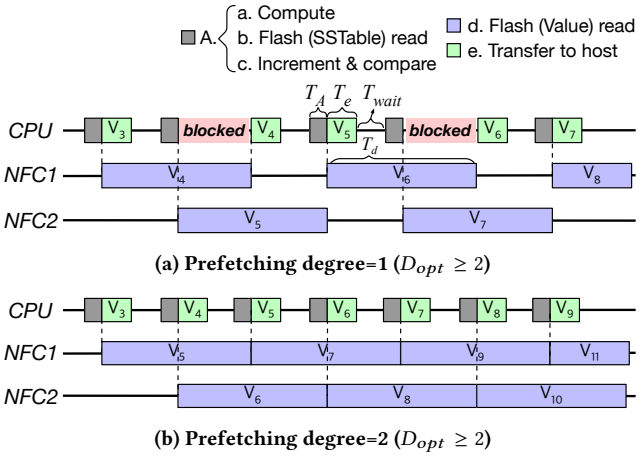
Figure 5(b) depicts the execution of Value Prefetch with a degree of 2. Whenever Step(e) is executed, the CPU requests the NFCs to prefetch the next values in advance to minimize synchronous NAND access. Next, we describe the process of Value Prefetch in detail. Assume that the prefetching degree is N and the Next() command will return (K_i, V_i). Before returning (K_i, V_i) from memory to the host in Step(e), *IterKVSSD* requests NFCs to prefetch (K_{i+N}, V_{i+N}). Since the physical location of V_{i+N} can be obtained from the SSTable buffered in memory, there is no need for additional access to NAND flash. As such, the larger the prefetching degree, the more it can be expected to hide CPU blocking time for NAND flash access. However, if the prefetching degree is too large, unnecessary



(a) Without Value Prefetch

(b) With Value Prefetch (prefetching degree=2)

Figure 5: Description of Value Prefetch.



(a) Prefetching degree=1 ($D_{opt} \geq 2$)

(b) Prefetching degree=2 ($D_{opt} \geq 2$)

Figure 6: Description of the situation where CPU blocking occurs in Step(d) according to the degree of prefetch.

read ahead causes waste of memory and loss of internal bandwidth. Therefore, finding an appropriate prefetching degree is critical in terms of memory space utilization and performance.

Finding Optimal Degree of Prefetch (D_{opt}): Figure 6 shows the effect of the prefetching degree in Value Prefetch. Figure 6(a) depicts the case where the prefetching degree is smaller than D_{opt}. In this case, the CPU blocks in Step(d) because the requested value is not present in memory. On the other hand, Figure 6(b) describes the case where the prefetching degree is larger than or equal to D_{opt}. Because sufficient values are prefetched, the CPU does not block in Step(d).

Next, we describe an example of how to find the optimal degree (D_{opt}) with the following mathematical notations. On a request for the i^{th} Key-Value pair, we define $T_A(i) = T_a(i)+T_b(i)+T_c(i)$, the sum of the times Steps(a), (b), and (c). $T_{wait}(i)$ is the time interval that the device waits for the next request after the completion of the current request. $T_F(i)$ is the time from when the i^{th} Key-Value pair is requested to be prefetched to when the i^{th} Key-Value pair is actually requested. Refer to the notations in Figure 6 for a better understanding.

Assume that the i^{th} Key-Value pair has been requested with the prefetching degree of D_{opt} at time t_i. Then, the time to finish loading the V_i into DRAM can be expressed as $t_i + T_d(i + D_{opt})$. In the future, assume that the $(i + D_{opt})^{\text{th}}$ Key-Value pair will be requested. When it is requested, to prevent the CPU from blocking in Step(d), its value must have been loaded into memory before t_i+$T_F(i + D_{opt})$. Therefore, the condition for D_{opt} can be expressed as follows: t_i+$T_F(i + D_{opt}) \geq t_i + T_d(i + D_{opt})$.

Meanwhile, $T_F(i + D_{opt})$ can be expressed as follows:

$$\begin{aligned} T_F(i + D_{opt}) &= T_e(i) + T_{wait} + T_A(i + 1) \\ &\quad + T_e(i + 1) + \ldots + T_{wait} + T_A(i + D_{opt}) \\ &= \sum_{j=1}^{D_{opt}} \left[T_e(i + j - 1) + T_{wait} + T_A(i + j)\right] \\ &= D_{opt} \cdot T_{wait} + \sum_{j=1}^{D_{opt}} \left[T_e(i + j - 1) + T_A(i + j)\right] \end{aligned}$$

Assuming that T_A, T_e, and T_d are constant, $T_F(i + D_{opt}) = D_{opt} \cdot (T_A + T_e + T_{wait})$. Also, since $T_F(i + D_{opt}) \geq T_d(i + D_{opt})$, where $T_F(i + D_{opt}) = D_{opt} \cdot (T_A + T_e + T_{wait})$ and $T_d(i + D_{opt}) = T_d$, D_{opt} can be determined from the following inequality.

$$D_{opt} \geq \frac{T_d}{T_A + T_e + T_{wait}}$$

Note that the inequality above shows the lower bound of the prefetching degree that maximizes the range query performance. In a more realistic scenario, T_A, T_e, and T_d may vary by Value size, I/O traffic from the host, and so on. As a result, the optimal degree of D_{opt} could also vary, making it difficult to predict the exact optimal degree. Therefore, the device may need to prefetch more than the lower bound to maximize the range query performance safely.

However, since the memory resource in the device is limited, the range query performance will diminish over a certain prefetching degree. In other words, if the prefetching degree is too high compared to available memory in the device, prefetched entries will be evicted due to memory shortage. It suggests that the prefetching degree should be carefully set between D_{opt} and the upper bound determined by the memory constraint. To maximize the performance of range queries in a more realistic scenario, we plan to investigate the mathematical model of the upper bound and the adaptive prefetching mechanism at the host level in the future.

5 EVALUATION

5.1 Experimental Setup

We prototyped *IterKVSSD* on the OpenSSD Cosmos+ platform [17] which has Xilinx Zynq-7000 SoC, 1 TB NAND, and 1 GB DRAM. We set up OpenSSD Cosmos+ with the host machine through PCIe Gen2 8-lane with NVMe protocol. The host machine is equipped with Intel i7-8700K running at 3.7GHz and 16 GB DRAM.

Workload: We used *db_bench* [9], the representative microbenchmark tool that is used to benchmark RocksDB's performance. Before each experiment was conducted, we populated the KVSSD with key-value pairs that would be subjected to range queries. The size of the key is fixed at 4 B for all workloads, and we tested various

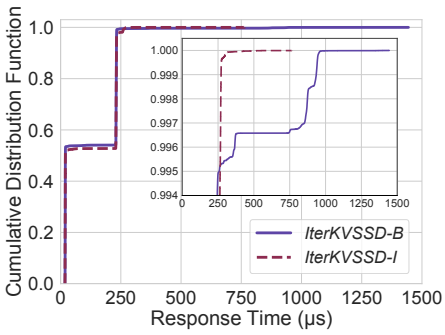


Figure 7: Analysis of the impact of Index Prefetch of *IterKVSSD*-I without Value Prefetch.

value sizes from 128 B to 128 KB. Also, the number of key-value pairs is 3 M for each workload. We used both *fillrandom* and *fillseq* workloads to verify the efficiency of our solution under the different key distributions. Afterward, we measured the throughput and latency of the range query while running the *SeekRandom* benchmark of *db_bench*. SeekRandom workload is a workload that repeatedly searches a random key range using an iterator. In this workload, a single range query consists of Create(), Seek(), multiple Next()s, and Delete() commands. Scan length determines the number of Next() commands called in a range query. To verify the performance of range queries in more practical scenarios, we determine the scan length based on previous studies on workload analysis of Key-Value store [1, 3].

5.2 Effect of Index Prefetch

In order to measure the effect of Index Prefetch precisely, we conduct our evaluation and compare our proposed design with a baseline design as follows:

- ***IterKVSSD*-B:** A baseline design of KVSSD with an iterator interface without Index Prefetch.
- ***IterKVSSD*-I:** KVSSD with an iterator interface that implements the Index Prefetch technique on *IterKVSSD*-B.

Figure 7 shows the results of the cumulative distribution function (CDF) of the response times to compare *IterKVSSD*-B and *IterKVSSD*-I. As mentioned in Section 4.4 Index Prefetch is barely triggered during the execution of Next() commands. In order to prove the effect of Index Prefetch, we set the scan length of *SeekRandom* to 200 k, which is enough to trigger an index read in the middle of a range query. Then, we measured the latency of Next() to observe the tail latency. It must be noted that Value Prefetch is disabled to only evaluate the effect of Index Prefetch for *IterKVSSD*-I.

We noticed that *IterKVSSD*-B and *IterKVSSD*-I show almost the same latency for Next(). However, we observed that *IterKVSSD*-I significantly reduces the tail latency of *IterKVSSD*-B (the enlarged plot). The P99.9 tail latency of *IterKVSSD*-I is about 3.6 times lower than *IterKVSSD*-B. Nevertheless, it is presumed that the reason *IterKVSSD*-I cannot completely eliminate the tail latency of *IterKVSSD*-B is due to the unavoidable channel conflict between NAND requests for the background Index Prefetch and foreground NAND requests.

One thing to note is that the proposed Index Prefetch technique is an orthogonal approach with the index pinning technique adopted by PinK. The index pinning technique still requires synchronous index reads for the deeper-level SSTables, which cannot be pinned in memory due to limited memory. Consequently, when accesses to deeper-level SSTables occur, the pinning index may still incur tail latency. In this regard, the advantage of Index Prefetch is that it can be applied simultaneously to PinK since it does not require much memory to operate.

5.3 Effect of Value Prefetch

In order to measure the effect of Value Prefetch precisely, we conduct our evaluation and compare our proposed design with a baseline design and PinK as follows:

- ***IterKVSSD*-B:** A baseline design of KVSSD with an iterator interface without Prefetch.
- ***IterKVSSD*-P:** KVSSD with an iterator interface that implements the Prefetch techniques (Index and Value).

Figure 8, 10 shows the efficiency of Value Prefetch on *IterKVSSD*. Here, we varied the scan length from 128 to 2048. As stated in Section 4.4, a scan length of 2048 is not long enough to trigger NAND accesses for reading consecutive SSTables. Since Index Prefetch rarely happens, Index Prefetch had little effect on the subsequent experimental results.

Comparison with PinK: We also implemented and evaluated PinK, which is a state-of-the-art solution. Our evaluation revealed that the overall performance difference between PinK and *IterKVSSD*-B is negligible under our setup. Range queries via an iterator interface first read the index from NAND flash on Seek() and then continue to traverse based on the index in memory on multiple Next() commands. In other words, once KVSSD reads the index from NAND flash, there is little chance to read the index in the middle of the range query. It means that the effect of the pinning index in memory has much little impact on the overall performance of range queries. In the following, we refer to PinK as *IterKVSSD*-B and continue the analysis.

5.3.1 Random Key Distributions. Figure 8 shows the range query throughput comparison of *IterKVSSD*-B and *IterKVSSD*-P with the random key distribution. We measured throughput (MB/s) of range queries while varying the prefetching degree, value size, and scan length.

Effect of the prefetching degree and scan length: Figure 8(a)-(d) show the change in throughput as the prefetching degree and scan length increase. The throughput of the range query increases rapidly at first as the prefetching degree increases and then increases gradually, as shown in each figure. This increase is mainly due to the effect of hiding NAND flash access time with Value Prefetch, which is processed in parallel with foreground requests. The gradual increase is due to internal parallelism no longer being exploited after a certain prefetching degree.

In Figure 8(a), we also observe that the throughput of the range query increases as scan length increases, regardless of the degree of prefetching. The reason is explained in detail in Figure 9. Figure 9 shows the difference in latency between Seek() and Next() during the range query. The result shows that the average latency of Seek()

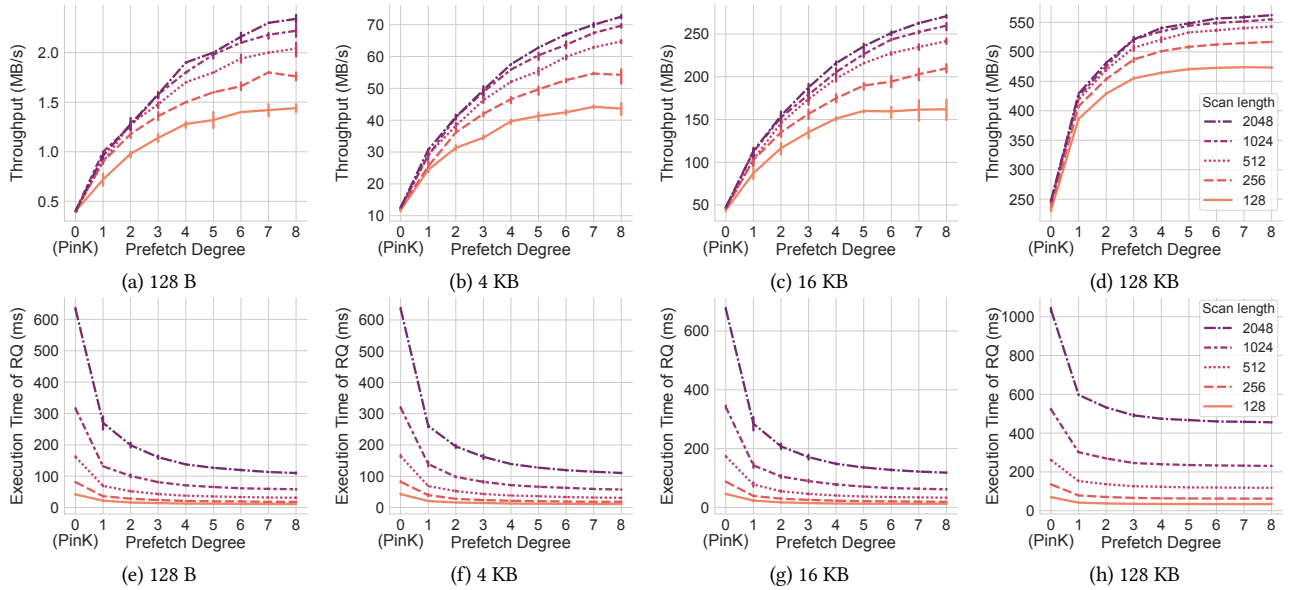


Figure 8: Evaluation of the effect of value prefetch on throughput (a)-(d) and execution time (e)-(h) for *SeekRandom* with random key distributions for different value sizes. Standard deviations are shown in error bars. RQ on the y-axis denotes Range Query.

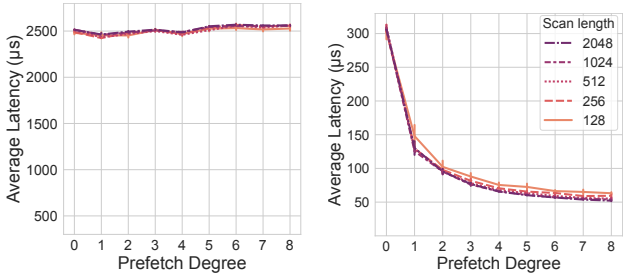


Figure 9: Performance comparison of the Seek() and Next() commands for a value size of 128 B with varying prefetch degrees.

is higher than that of Next(). That is, as scan length increases, the overall range query performance is dominated by Next(), which is faster than Seek(). Similar tendencies with Figure 8(a) can also be observed in Figure 8(b)-(d).

Effect of value size: Figure 8(a)-(d) show the increase in throughput as value size increases. Although the number of Key-Value pairs to retrieve through a range query remains the same, the overall data retrieved increases in size, which improves throughput. Moreover, when the value size is 128 KB or higher, the increase rate of the throughput by the prefetching degree rapidly becomes slower (refer to Figure 8(d)). This is because the internal bandwidth of the SSD is quickly saturated when value sizes are sufficiently large.

Figure 8(e)-(h) shows the execution time of the same experiments. We observe that the execution time decreases as the prefetching degree increases. This result also explains the reason for the increase in throughput as the prefetching degree increases. We also find that execution time increases as scan length increases. This is because the larger the scan length, the larger the number of key-value pairs retrieved through range queries, and thus a larger amount of data is requested. Moreover, we observe that the execution time increases as value size increases as well. This is because the amount of data to be read through range query increases as the size of the value to be read increases as well. Interestingly, we observe that execution times are the same in Figure 8(e) and Figure 8(f). This is due to the 4 KB minimum transfer size of the NVMe protocol. Thus, even when the device sends 128 B data to the host, it actually transfers 4 KB with the data to be sent.

5.3.2 Sequential Key Distributions. Figure 10 shows the results with the sequential key distribution. Except for the workload characteristics, the experimental methodology is the same as the experiment for Figure 8. Figure 10 shows that the overall throughput of range queries in sequential key distribution is higher than that with random key distribution as shown in Figure 8. In sequential key distribution, Key-Value pairs with adjacent keys are also stored contiguously in vLog as opposed to random key distribution. Because of this high spatial locality, *IterKVSSD*-P can prefetch more values with fewer NAND flash reads and can utilize more internal bandwidth. Except for these observations, the overall trends in throughput of range queries are similar to those in Figure 8. Also, Figure 10(e)-(h) shows similar tendencies for the execution time of range queries with the results in Figure 8(e)-(h).

5.3.3 Latency comparison for each command of range query. A range query is composed of one Seek() and repeated Next() commands. To compare the performance difference between these two commands, we measured the average latency of each, with a random key distribution, while increasing the prefetching degree and

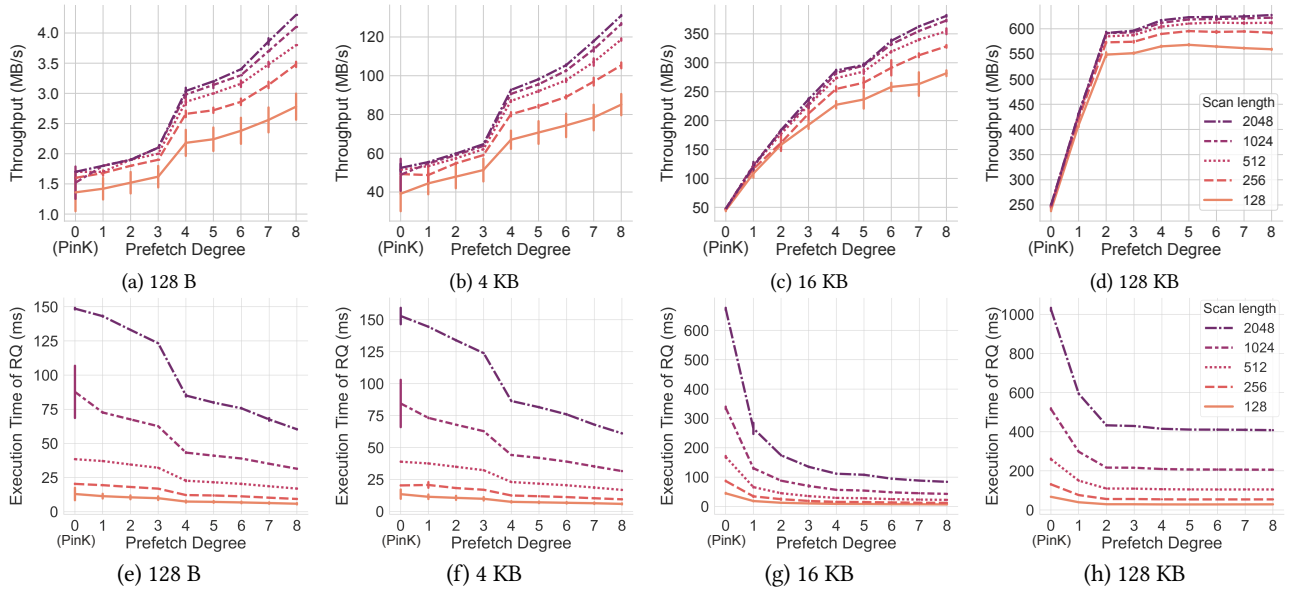


Figure 10: Evaluation of the effect of Value Prefetch on throughput (a)-(d) and execution time (e)-(h) for *SeekRandom* with sequential key distributions for different value sizes.

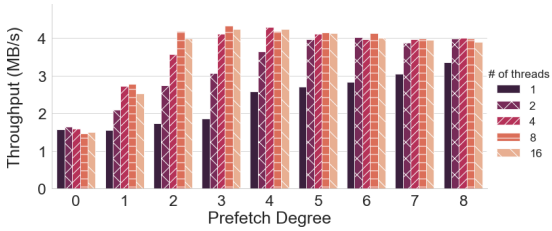


Figure 11: Evaluation of concurrent range queries.

scan length. Figure 9 shows the results when the value size is 128 B. In Figure 9, we observe that the average latency of Seek() is higher than that of Next(). This is because Seek() must synchronously read SSTables covering the search range from NAND flash. On the other hand, synchronous SSTable reads from NAND flash are not always necessary for Next(), as Next() can retrieve the keys using the SSTable already loaded into DRAM.

Moreover, we observe that the average latency of Seek() does not improve at all as the prefetching degree increases while the average latency of Next() greatly decreases. Since Seek() initiates a range query, it is hard to exploit the advantage of Value Prefetch. However, in the case of Next(), while it is repeatedly called, successive values can be prefetched in the background, which greatly improves the average latency. For this reason, we notice that the latency of Next() decreases as the prefetching degree increases.

5.3.4 Evaluation of concurrent range queries. We evaluate the performance when multiple iterators perform concurrent range queries for *IterKVSSD*-P. For the evaluation, we performed the following experiments. We used the *fillseq* workload of *db_bench* the same way as for the previous experiments. The value size and scan length were fixed at 128 B and 128, respectively. After that, we created multiple threads, each thread performing a range query with the workload *SeekRandom*, and measured the aggregate throughput of threads by increasing the prefetching degree. Figure 11 shows that the aggregate throughput increases as the prefetching degree and the number of threads both increase.

We observe that the rate of increase in throughput slows down when the scan length is large, and the degree of prefetching is high. This is because some iterators already saturate the SSD's internal bandwidth completely.

In the current implementation, we allocate device memory statically for versioning data structures, so the versioning feature itself has little impact on the performance of range queries, even if memory is allocated for multiple iterator instances. However, if multiple iterator instances concurrently conduct range queries, it may affect the performance of each iterator instance because the iterators share some resources, such as prefetching buffer and internal bandwidth for NAND flash controllers. This interference between multiple iterators' performance should also be handled from the host side by adaptively setting the prefetching degree and range query traffic control.

6 RELATED WORK

Support for Range Query in KVSSDs: There have been few previous studies on KVSSDs that support range queries even though it is an essential feature of KVSSDs. Since LSM-tree is an ordered data structure, it has been actively discussed as a suitable candidate for range queries in KVSSDs. For instance, KevinSSD [16] also presents an iterator interface to support range queries, along with PinK. In KevinSSD, a new type of file system named KevinFS that supports the Key-Value interface was proposed as a replacement for a traditional file system. However, proposed techniques adopted by KevinSSD, such as index compression, are compatible and feasible only with KevinFS.

Host-side Key-Value Store: There are many key-value solutions that try to follow the aforementioned principles (**P1**, **P2**, **P3**) from the host side. For the past years, there have been various key-value stores at the host side adopting the LSM-tree such as BigTable, LevelDB, Cassandra, RocksDB [5, 8, 11, 18]. In particular, RocksDB [8], developed by Facebook, has been widely deployed by a number of companies due to its high performance and adaptability. To resolve the three challenges described in Section 3, RocksDB has adopted the following solutions [8, 10]. First of all, for the **P1**, RocksDB introduces three concepts named Version, VersionEdit, and transactional log for VersionEdit (*MANIFEST*). By enabling the version control mechanism, iterator objects in RocksDB can keep track of the states of the whole database at their creation time. RocksDB meets **P2** by a huge amount of indexes pinned in memory. As a result, iterators in RocksDB can usually find the location of a value without accessing storage. For the last **P3**, RocksDB also has a large block cache layer and performs internal auto-prefetching when it notices sequential reads. With auto prefetching in the background, RocksDB can hide the latency for accessing storage devices. The above solutions adopted by RocksDB mostly leverage host-side abundant DRAM to minimize access to storage.

Along with RocksDB, multiple host-level Key-Value stores such as WiscKey [20] and HashKV [4] introduced Value Prefetching to boost the performance of range queries. However, since these are also host-level solutions, their approaches are not feasible for an SSD device. Furthermore, they entail significant I/O amplification and require large I/O bandwidth. On the other hand, KVSSDs offer full control over data placement at the hardware level, including NAND flash channels, blocks, and pages. This enables efficient management of data placement at the hardware level and the exploitation of internal parallelism of SSDs. In other words, *IterKVSSD* can directly access physical addresses and execute prefetching without any overhead from other layers, such as file systems.

7 CONCLUSION

In this paper, we present *IterKVSSD* strictly following the design principles with three specific enhancements: (i) A versioning mechanism using the *Summary* of the LSM-tree with a memory-efficient pooling method; (ii) Index Prefetch to mitigate the NAND access cost for index reads; (iii) Value Prefetch to mitigate the NAND access cost for value reads. Extensive evaluations using a real implementation on the OpenSSD Cosmos+ platform show that our proposed ideas exhibit high memory space efficiency and high range query performance.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Patrick P. C. Lee, for his feedback. This work was funded in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386) and in part by SK hynix research grant.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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*. ACM, 53–64.

[2] Janki Bhimani, Jingpei Yang, Ningfang Mi, Changho Choi, and Manoj Saha. 2021. Fine-grained Control of Concurrency within KV-SSDs. In *Proceeding of the 14th ACM International System and Storage Conference (SYSTOR '21)*. ACM, Association for Computing Machinery, 1–12.

[3] 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)*. 209–223.

[4] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[5] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. 1–15.

[6] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. 2019. LightStore: Software-Defined Network-Attached Key-Value Drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 939–953.

[7] 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)*. 143–154.

[8] Facebook. 2021. *RocksDB*. http://rocksdb.org

[9] Facebook. 2021. *RocksDB Database Benchmark Tool*. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

[10] Meta. Facebook. 2022. RocksDB v7.2.2 Release. https://github.com/facebook/rocksdb/releases/tag/v7.2.2.

[11] Google. 2017. LevelDB. https://github.com/google/leveldb.

[12] 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 (ATC '20)*. 173–187.

[13] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven 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.

[14] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG 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)*. 144–154.

[15] Yang Seok Ki. 2017. Key Value SSD Explained–Concept, Device, System, and Standard. In *Storage Developer Conference*.

[16] 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)*. 75–92.

[17] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage*, Article 15 (July 2020), 35 pages.

[18] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.

[19] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '19)*. 384–395.

[20] 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 USENIX Conference on File and Storage Technologies (FAST '16)*. 133–148.

[21] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[22] Donghyun Min and Youngjae Kim. 2021. Isolating namespace and performance in key-value SSDs for multi-tenant environments. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)*. 8–13.

[23] Inc NVM Express. 2021. NVM Express Key Value Command Set Specification. https://nvmexpress.org/developers/nvme-specification/

[24] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[25] Manoj P. Saha, Adnan Maruf, Bryan S. Kim, and Janki Bhimani. 2021. KV-SSD: What Is It Good For?. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*. 1105–1110.

[26] SK hynix and Los Alamos National Laboratory. 2022. Los Alamos National Laboratory and SK hynix to demonstrate first-of-a-kind ordered Key-value Store Computational Storage Device. https://discover.lanl.gov/news/0728-storage-device.

[27] SNIA. 2020. Key Value Storage API Specification. https://www.snia.org/keyvalue

[28] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2018. KVSSD: Close Integration of LSM Trees and Flash Translation Layer for Write-efficient KV Store. In *Proceeding of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE '18)*. IEEE, 563–568.

[29] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*. 51–64.