Dotori: A Key-Value SSD Based KV Store

Carl Duffy
Seoul National University
Seoul, Korea
cduffy@snu.ac.kr

Jaehoon Shim
Seoul National University
Seoul, Korea
mattjs@snu.ac.kr

Sang-Hoon Kim
Ajou University
Suwon, Korea
sanghoonkim@ajou.ac.kr

Jin-Soo Kim
Seoul National University
Seoul, Korea
jinsoo.kim@snu.ac.kr

ABSTRACT

Key-value SSDs (KVSSDs) represent a major shift in the storage stack design, with numerous potential benefits. Despite this, their lack of native features critical to operation in real world scenarios hinders their adoption, and these benefits go unrealized. Moreover, simply adapting existing key-value stores to run on KVSSDs proves underwhelming, as KVSSDs operate at lower raw device performance when compared to modern block SSDs.

This paper introduces Dotori. Dotori is a KVSSD based key-value store that provides much needed functionality in a KVSSD through an upper layer in the host, and takes advantage of the unique KVSSD interface to enable further gains in functionality and performance. At the core of Dotori is a novel B+tree design that is only practical when the underlying storage device is a KVSSD.

We test Dotori with an enterprise grade KVSSD against state-of-the-art block SSD based key-value stores through a range of micro-benchmarks and real world workloads. Despite low KVSSD raw device performance, Dotori achieves superior performance to these block-device based key-value stores while also showing significant gains in other important metrics.

PVLDB Reference Format:
Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. Dotori: A Key-Value SSD Based KV Store. PVLDB, 16(6): 1560 - 1572, 2023.
doi:10.14778/3583140.3583167

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/snu-csl/Dotori.

1 INTRODUCTION

Key-value stores (KV stores), and by extension, the key-value abstraction, are a favored choice for storage throughout various environments and applications. Examples include stream processing systems [7], cloud storage environments[26], caching systems[33], and more [2, 12, 14]. These KV stores are typically realized as a software layer on top of traditional block-based storage devices; hard disk drives (HDDs) and solid state drives (SSDs). However, mismatch between the key-value abstraction and the block interface causes a range of problems when block device-based KV stores introduce indexing to find key-value data. For example, reads and rewrites to keep data sorted on disk and delete stale values (compaction) notably impacts performance and device lifetime, as does index persistence for crash recovery.

We perform an experiment where we record the operations per second of two 100% random insert workloads using RocksDB [15], a popular LSM-tree based KV store, and ForestDB [1], a KV store that uses a series of B+trees in a Trie to index data. For the first run for each store, we use the default settings. For the second run, we disable compaction in RocksDB and disable index persistence in ForestDB. Without compaction, insertion in RocksDB is 4.9x faster, and without index persistence, insertion in ForestDB is 3.3x faster. Moreover, despite a pure insert workload, both stores read a significant amount of data, and both suffer from notable write amplification.

In an attempt to rectify these issues through a hardware-based approach, key-value SSDs (referred to as KVSSDs) have been created. Data is written directly to the device with the key itself, and as the burden of key-value pair management is delegated to the device, KVSSDs can offer high performance at little resource cost from a host system standpoint. We carry out the same workload as before on a KVSSD, which results in at least 1.6× the operations per second compared to the other stores when compaction or index persistence is on. Moreover, the KVSSD does not read any data, and application level write amplification is 1, as only data requested to be written by the user is sent to the KVSSD (as opposed to data for compaction or index persistence). However, as KVSSDs currently only expose a simple interface, they lack features critical to building real-world applications such as batching, transactions, snapshots, recovery, and range queries. As we will demonstrate in this work, the cost of providing these features is non-negligible; an index is required on the host side to enable them, however, running indexes previously designed for block devices on a KVSSD can cause performance to plummet.

In this paper we propose a hybrid architecture wherein both the host and underlying KVSSD work together to provide necessary KV store functionality, as opposed to shifting all functionality to one layer. In the host, a software layer called Dotori provides KV store auxiliary functionalities by managing a KVSSD-tailored host-side index. Underneath, the KVSSD provides fast key-value storage and the interface necessary for the host-side layer to efficiently manage said index. Evaluations using Dotori and a real KVSSD show significant gains in terms of throughput, latency, write amplification, and space amplification when compared to state-of-the-art KV stores running on comparatively faster block devices.

The major contributions of this work are:

(1) A novel B+tree design tailored to KVSSDs, named the OAK-tree.
(2) The design of a KVSSD based KV store, Dotori, based on the OAK-tree.

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583167

(3) An in-depth comparison of Dotori versus state-of-the-art block-device based KV stores running on an identical SSD with different firmware.

2 KVSSDS AND THE KV INTERFACE

In this section we overview the KV interface and KVSSDs, and then discuss a set of issues that the block interface causes in KV stores.

2.1 The KV Interface

The SNIA KV API Standard [42] defines the *store_kvp*, *retrieve_kvp*, *delete_kvp*, and *iterate* commands, among others. *store*, *retrieve*, and *delete* are self-explanatory. *Iterate* allows a user to provide a prefix for which KV pairs on the device will be compared with, and returned if they match. These commands take a key-value pair, not a logical address, and translation from key to physical location on disk is done entirely inside the device. A KVSSD is essentially a hardware KV store, but the features exported by each KVSSD are different. Some designs export features in-device such as transactions [21][25], whereas others export a more simple interface focusing on store and retrieve performance [19, 22].

2.2 Why is the KV Interface Better for KV Stores?

Key-Value Stores KV stores aim to provide fast access to key-value pairs while providing data consistency, transactions capabilities, and more. However, modern storage devices operate on a block-based interface, which causes mismatch when key-value pairs are to be stored as offsets on a device. For example, user data belonging to a key `VLDB23` somehow needs to ultimately be written to and retrieved from an offset on a block device. This is realized through translations at the KV store itself, the file system, and on the device. KV stores try to close this semantic gap by introducing persistent indexes for efficiently storing and retrieving key-value pairs.

The mismatch between keys and the block interface ultimately leads to numerous problems, and we explore them in this section. At the crux of the KV interface lies three main benefits that the block interface cannot provide. First, processing required for managing KV data is performed closer to storage, reducing data transfer between the host system and device. Second, the burden of locating KV pairs is delegated to the device itself, freeing the host from persisting indexes of its own. Third, users can operate on KV pairs directly, instead of needing to organize data in larger logical units such as blocks.

Software Stack Overheads When a user wants to store or retrieve a KV pair in a KV store designed for block devices, the request goes through several translations; KV store to file, file system to device logical address, and device logical address to device physical address. Such overheads are now non-negligible as device speeds have increased [24, 46].

The KV interface greatly simplifies the journey a request must make; the KV pair is sent directly to the device, and the only translation performed is from key to physical address on the device. We note that this benefit of the KV interface is, currently, mostly theoretical, as extra work at the device level to maintain its own index increases latency notably (see section 2.3).

Read and Write Amplification Read and write amplification reduce performance as more time is spent transferring data to and from host to device instead of serving user queries, and read amplification reduces cache effectiveness [29]. While a large amount of work has gone into reducing these penalties in popular indexes [4, 5, 11, 29, 37, 47], it is ultimately a losing battle; the problem lies at the interface, not the index design. Whenever key-value data resides within the same logical unit (a block), data copies will always be necessary to read and rewrite valid entries during processes like compaction that free up stale data. It is worth mentioning that some forms of compaction have further utility in keeping data sorted on disk for fast scans, but as random access speeds on modern storage devices increase this trade-off between lower performance due to compaction and scan optimization becomes less worthwhile.

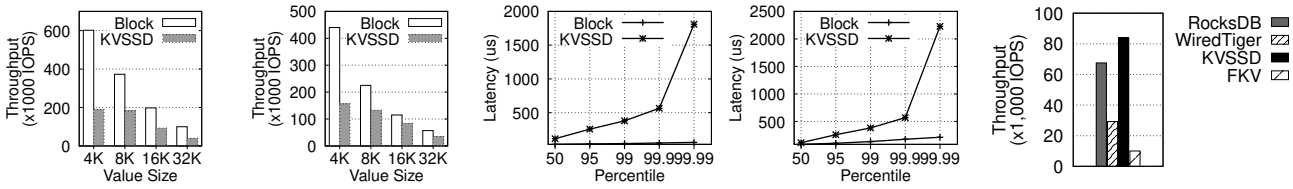
In comparison, the KV interface alleviates this problem by enabling KV data to be deleted inside the device with a delete command, completely avoiding any data transfer between host and device.

Space Amplification High space amplification is a problem discussed in several recent works [13, 18, 28]. In KV stores, there is a trade off between space amplification and performance. For example, in RocksDB space amplification can be controlled by changing the size of levels and size multipliers between levels of the LSM-tree [16], where lower level sizes and multipliers, leading to lower space amplification, may decrease performance and increase write amplification.

In hash table based KVSSDs (such as the one we test in this work), space usage may be aggressively curbed because removing key-value data (by sending a delete command) is a lightweight operation that can be scheduled to be performed when the device sees fit.

Recovery Performance KV stores running on block devices optimize recovery by designing the store in such a way that only the key-value pairs from the latest commit or flush are required to be read for the recovery. We run a write heavy workload on ForestDB and WiredTiger [43] (a state-of-the-practice KV store) and crash the stores at a time where recovery performance will be close to its worst. During recovery, ForestDB's steady-state read performance is 45MB/s, and WiredTiger's is 35MB/s. The tested drive had a max sequential read throughput of 3100MB/s. Despite recovery being theoretically simple in that all that is usually needed is scan of a log or series of records, intricacies in the recovery process introduced by the block interface slow down the process significantly. In the ForestDB example, the crash happened during a compaction, and the compaction recovery process incurred an initial slow circular scan of the file to find the latest valid header. In WiredTiger, the log scan is slowed significantly by checks needed to verify each log record in order.

When using the KV interface in the above examples, ForestDB's header scan could have been carried out in parallel as we would already know how a header key is formed. In the WiredTiger case, every pair in the log could be directly read simultaneously and checked, because logical separation of KV pairs at the interface level means we do not need to first discover where the records begin and end in a larger block or file.



(a) Read IOPS (b) Write IOPS (c) Read Latency (d) Write Latency (e) Real Workload Performance

Figure 1: Performance of a KVSSD and block SSD running on identical hardware.

2.3 KVSSD Performance Characteristics

On paper, KVSSDs have potential, but does this translate to better real-world performance? We evaluate the performance of a KVSSD running on high-end hardware. The device is a 1 TB Samsung PM983 NVMe KVSSD. For the performance comparisons, the block SSD is the same device as the KVSSD, with the firmware re-flashed to provide a block interface. We use 20 threads performing random, synchronous I/O for each test. Our tested KVSSD is a hash-table based KVSSD as described in [22].

IOPS and Latency Figures 1a and 1b show the retrieve/read and write/store IOPS at value sizes from 4 KB to 32 KB. The KVSSD's IOPS are notably low when compared to the block SSD, around 66% less in the worst case. Considering that both evaluations ran on the same device, this evaluation demonstrates a non-negligible overhead for actual KVSSD implementation on fast, modern hardware. The possible reasons for the overhead are numerous. First, in-device mapping management in a block SSD can be less complex; data is of fixed size, and DRAM and space allocation is simpler than in a KVSSD where both key and value size are of arbitrary length [38]. Second, KVSSDs must account for not only where to store the value, but where to store the key too. Third, KVSSDs (like the one we test) may need to persist extra information to support iterators, such as buckets of key prefixes.

Figures 1c and 1d show the read and write latencies for each device at a 4K value size. The latency results display an even more stark difference than throughput. We attribute this to the extra overheads required at the device level described before. In particular, tail latencies for both reads and writes are higher in the KVSSD we test due to collisions that happen at the device's hash table. Retrieves require potentially several extra hash table reads to find a key-value pair. Similarly, collisions affect write latency too, as the KVSSD needs to read and check the relevant parts of the hash table for duplicate KV pairs before inserting a new one.

Real Workload Performance The results paint a somewhat dire picture for the KVSSD's performance. However, given that KVSSDs can free a system from expensive processes such as compaction, are such results mirrored in real workloads when compared to block KV stores? We compare the performance of the KVSSD on a 50/50 uniform read/write workload against popular block-device based KV stores RocksDB and WiredTiger. Key sizes are set to 8 bytes and value sizes range from 40 to 4000 bytes. We run the workload for 10 minutes. Figure 1e shows the results.

Despite the lower device performance, the KVSSD offers superior performance to the other stores. The reasons are those discussed previously; performance penalties for compaction (RocksDB), index persistence and lookup (WiredTiger), lower read performance due to level lookups (RocksDB), and more. However, despite these initial impressive results, we contend that the performance testing in this manner is unrepresentative of actual performance if a KVSSD was to replace feature-complete KV store in the real world. The hash-table based KVSSD we test does not support transactions and snapshots, and has limited support for range queries. To provide such features, an index needs to be present in the host system.

The rightmost bar on the graph, named 'FKV', is the performance of an implementation of ForestDB (which provides all of the features mentioned previously) where we replace the block SSD with a KVSSD and disable ForestDB compaction. KV pairs are written and read directly, instead of being placed within blocks as in the original ForestDB. This configuration provides the full feature-set of ForestDB, including its index, representing a naïve implementation of a KV store on a KVSSD where a leaner I/O stack is in use, and compaction is avoided thanks to the key-value interface (compaction is replaced by sending delete commands to the KVSSD). Performance of this prototype plummets as the KVSSD needs to execute frequent read-modify-writes to maintain ForestDB's index, exposing the KVSSDs lower device performance.

2.4 The Different Types of KVSSD

In this work we use a hash-table based KVSSD that follows the SNIA KV API standard instead of a) a KVSSD that already supports features such as transactions natively in the device [21], and b) a KVSSD that uses an LSM-tree instead of a hash-table to index KV pairs [19]. We focus on such a KVSSD for several reasons.

First, the KVSSD we test is, to the best of our knowledge, the only available commercial-grade KVSSD on a modern, fast NVMe device, representing actual current KVSSD performance at the enterprise level. Likewise, the KVSSD we test is the only one we know of that conforms to either the SNIA KV Storage API Specification [42] or NVMe KV Command Set [34]. Targeting a device that uses a standardized interface results in work that is more valuable to the community at large, as devices with features that do not conform to standardized interfaces require users to make adaptations or changes which they may not be willing to make. Moreover, such a focus on creating a KV store for a standardized KVSSD interface is even more important at this early stage of KVSSD development, where uptake of the devices is slow.

Second, hash-table based KVSSDs represent a class of KVSSD that is cheaper to make. When designing a system for KV storage,

the trade-offs include cost vs performance, not just a sole focus on performance. The hash-table based KVSSD we test runs on identical hardware to a standard block NVMe drive, with only the firmware changed. Current LSM-tree based KVSSDs require an additional accelerator in the device to facilitate fast compaction.

Third, it is advantageous to system designers and developers to have auxiliary KV store functionality such as transactions be implemented on the host system, as they can be more easily modified.

2.5 Discussion

The results from this section form our motivation for this work. KVSSDs have the potential to outperform modern block-device based KV stores, despite having significantly lower raw device performance. However, if the KVSSD itself does not provide features such as transactions, snapshots, and range queries, an index will need to be maintained on the host system. Using an index that performs well on modern, fast block devices and simply replacing the block device with a KVSSD is both problematic and wasteful. It is problematic because as current KVSSDs are much slower than modern block devices, as the KVSSD's lack of performance is amplified when such an index is ran on it. It is wasteful because such a configuration does not fully exploit the benefits of the KV interface, such as delegating data placement and retrieval to the device.

It is an attractive notion to place all of these features inside the device, but until such features are standardized (if ever), developing for and improving KVSSDs themselves is challenging as the target is an ever-changing set of commands, constraints, and requirements. Designing a KV store for a standardized interface, as we do in this work, gives developers and researchers a fixed set of constraints to work in, and device manufacturers a fixed set of features to improve. In this work we ask and demonstrate, can the KV interface be used to design a new, faster index that eventually allows the KVSSD to overtake the block SSD in performance?

3 DOTORI

3.1 Overview

Dotori is a host-side software layer that provides transactions, snapshots, and recovery functionality while at the same time achieving high performance when using a KVSSD as the underlying storage device. Dotori is based on ForestDB, but switches out the index of ForestDB with a new B+tree-like index (Section 3.2) called the OAK-tree, and makes several major design changes (Section 3.4). Data from the OAK-Tree is persisted to the KVSSD for recovery, as well as user KV data. The full read and write paths of Dotori are detailed in section 3.3. We first introduce Dotori's index.

3.2 Indexing

In KV stores designed for block devices, indexing is required first and foremost to find a key-value pair on a device. In addition, features such as snapshots necessitate that multiple versions of key-value pairs exist on the device at any time (i.e., key-value pairs are not overwritten in place), and indexing may be required to find the different versions of these pairs. If a user simply requires fast key-value storage, a KVSSD alone without an upper software layer such as Dotori is sufficient. However, when using a KVSSD that conforms to current KV interface standards, indexing is required in the host system if the user needs features such as snapshots, range queries, and transactions, as the standards themselves do not describe these features.

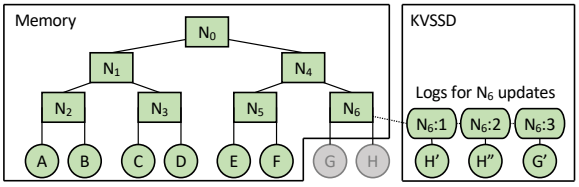


Figure 2: OAK-tree insert process.

Two prevalent B+tree designs are copy-on-write and write optimized B+trees. In a copy-on-write B+tree, a typical write process is as follows. First, the tree is read from the root to leaf node to find the location of the requested key-value pair. Each node is stored as a separate block, and usually the inner nodes are cached in the memory incurring little read overhead. Next, the leaf node is scanned and the location of the key-value pair is found. The value is updated if it previously exists, or inserted if not. This modification is applied to the node cached in memory, and then the node is persisted later to storage. The downside of this scheme is that as leaf nodes are updated per insert or update, the target leaf needs to be read and re-written in its entirety even for small updates. Such a process turns writes into a read-modify-write path, creates a large amount of write amplification, and can hamper write performance if the nodes that need to receive updates aren't in memory at the time.

Write-optimized designs differ from copy-on-write designs in that the initial insert of new data may not need a read and rewrite of the node. An example of this design is the B^{ϵ}-tree [20][36]. Index nodes contain per-node buffers where data can be inserted, and instead of new data being inserted to the leaf node immediately, data is buffered in the root. Later, data is flushed from the root down to the leaf when the buffers are full. While such a design helps write throughput, flushes later down the tree are expensive, and added complexity is introduced to the read path as write buffers and the index node buffers themselves need to be checked for the KV pair before moving down the tree.

The two designs are essentially trying to work around the same complication; we cannot blindly persist updates immediately *at the leaf* without having it in memory, because we would need to store the locations these updates somewhere, and find them after a crash. The key feature of the KV interface is that it delegates the search of KV data to the device, and as a result we are able to sparsely write index data without needing to worry about where the data lies on the disk. The problems we discuss in this section and this insight into the benefit of the KV interface motivate the design of the OAK-tree.

OAK-tree OAK-tree stands for ***O**ut-of-Order **A**ppend-Only **K**VSSD Based B+**tree***. The in-memory representation of the OAK-tree is identical to that of a typical B+tree, but the methods of index data persistence and retrieval differ significantly.

Algorithm 1 OAK-tree Insert Process

```
1:  procedure OAK-tree Insert(tree_root, K, V)
2:      leaf ← find corresponding leaf node(K, tree_root)
3:      if leaf not in memory then
4:          leaf.nr_entries++
5:          if leaf.nr_entries < T_Split then
6:              log(leaf, K, V)
7:              return
8:          /* Load the leaf node into memory.
9:          This will update leaf.nr_entries accordingly */
10:         leaf ← load node(tree_root, leaf)
11:     if K not in leaf then
12:         leaf.nr_entries++
13:     if leaf.nr_entries ≥ T_Split then
14:         split(leaf)
15:     update(leaf, K, V)
16:     log(leaf, K, V)
```

Writes The key idea behind the OAK-tree is that when using a KVSSD we do not need to write data in index nodes in order to storage, nor do we need to read the node before writing. With prior information of where index node data *could* be located (for example, special keys designated for index data), we can blindly write updates belonging to an index node and simply retrieve them later using the KV interface. Similar to write-optimized designs designed for the block interface, the OAK-tree appends index node updates to per-node logs, *without* first reading the previously existing logs for that node into the memory of the host. However, the OAK-tree appends data at the log for *that node*, and does not buffer data in another node to be flushed later. This has the advantage that expensive flushes all the way down the tree are avoided entirely, as data already exists at its final location.

Writing data sparsely to an arbitrary number of logs per index node is only practical using the KV interface. If we were to implement the same scheme using the block interface, we would need some way to locate each log (of which there are many) for each index node. The process would essentially require another index to exist just for the outstanding logs, which in turn would need to also be persisted, doubling the cost of index persistence as a whole. With the KV interface, we do not need any extra indexing to find the logs on disk; the device does it for us.

Figure 2 overviews the update process for the OAK-tree. We assume index node N_6 is not cached when updates arrive. In the OAK-tree, the modifications to N_6 will be written to a log keyed with index node ID of N_6 on storage and the ID of the next log (an in-memory counter exists per node) to be written for N_6 (e.g., $N_6 : 1$, if $N_6 : 0$ already existed on storage). No previous data relating to index node N_6 is read into host memory during the update process. As shown at the bottom right of Figure 2, updates are written to the KVSSD with keys $N_6 : 1$ to $N_6 : 3$. The figure assumes these updates were written to logs at separate times, but they could also have been written together in one log if they were present at the same time. The process is the same if index node N_6 already exists in the cache; only the updates to node N_6 are written to the log. Finally, at no point is the index node itself written to storage, only the logs are. This design avoids problems with B+trees on block storage described before; nodes are not read before an update or written in their entirety per update, and node data does not need to be flushed from upper node to its final location later, as in the B^ϵ-Tree. The OAK-tree log format is a simple series of 16B KV pairs of user key and Dotori internal key, explained in Section 3.4.

Reads A MAX_LOGS parameter determines how many logs can exist on storage at a given time for an index node. When an index node is absent from the cache and needs to be read, this MAX_LOGS parameter is used to determine the range of keys that need to be read from the KVSSD. For example, assume Dotori is configured with MAX_LOGS set to 5. If a user wants to read a key-value pair and its corresponding index node N is not in the cache, keys $N : 0$ to $N : 4$ will be read asynchronously and in parallel. Failed reads (logs that didn't exist) are simply ignored, and once all of the logs are collected they are merged and the index node is created in host memory. Logs may also be cached, and thus may not require a read to storage.

Splits The design of the OAK-tree introduces a unique problem relating to node splits. In this section we use the term update to refer to either an insert or update to an index node, and specify where appropriate. As modifications to index nodes can be written without prior knowledge of the contents of the node, we may not know how many entries an index node contains during the update process. The result is that we do not know if the current modification of the node is an insert (which could potentially cause a split), or an update (which does not increase the amount of entries in the node). To solve this problem, Dotori simply initially assumes that every modification of an index node is an insert. An in-memory counter for each index node is incremented each time the node is to be updated, and that counter is used to determine whether or not the node is a potential candidate for a split.

The process is shown in Algorithm 1. For brevity, we only consider the insert process for the leaf. T_{split} refers to the threshold at which a node is split. After finding the location of leaf node corresponding to the tree, it is possible that the leaf itself is not in memory. The search for the leaf will return without reading the node from storage if it does not exist in the cache. In this case, the in-memory counter of entries for that index node is updated to reflect the current insert (line 4). If the number of entries in the node is below the threshold for a split, the log for the leaf is updated. Otherwise, the index node is constructed in memory by reading its logs from storage. Once the node is read from storage, the number of entries in the leaf is updated with the actual (not speculated) amount of entries (line 10). Even though the node was read into memory because of a possible split, the split is avoided if the real amount of entries in the node is below the threshold (line 13). Otherwise, the split proceeds. Finally, the node is updated and the changes to the node are logged.

When a node is split in Dotori, an extra log write is incurred for the node that is the source of the split. This log contains tombstone markers for the key-value pairs that were copied to the new node. When reading an index node from storage, the tombstone markers in the logs let the merge function know that these entries are to be ignored.

Scans The OAK-tree serves range queries by opening a snapshot (see section 3.4) and iterating the tree within said snapshot. This

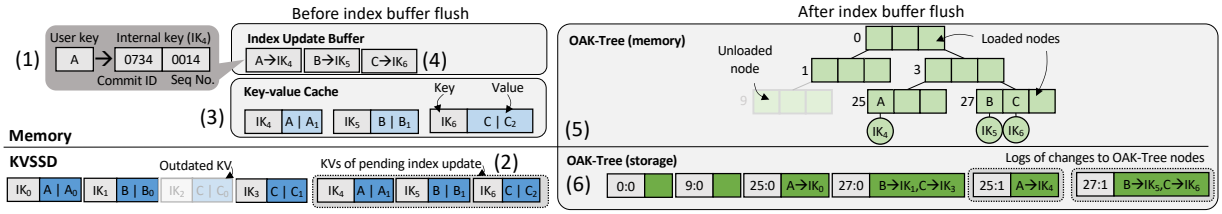


Figure 3: Detailed internals of Dotori.

snapshot is a logical abstraction in Dotori, not a snapshot function on the KVSSD. We choose to implement scans this way in Dotori as the KVSSD which it is currently designed for supports a slower, low priority scan function which isn't fast enough for scan-dependent workloads.

Log Repacking The MAX_LOGS parameter is used not only to determine the logs to read during index node retrieval, but also to place a cap on both the amount of space that can be taken for the logs of index nodes, and the time it takes to read the logs. If a log corresponding to an index node is to be written, but already has MAX_LOGS outstanding logs on storage, the outstanding logs for that node are read, stale entries are removed, and the resulting log is written back to log 0 of the index node, overwriting the previous value.

3.3 Read and Write Paths

In Figure 3 we display a detailed version of Dotori's internals, and step through the read and write paths in this section.

Writes (1) User keys *A*, *B*, and *C* are converted to internal keys *IK4*, *IK5*, and *IK6*. For example, *A's* internal key is *0734 0014*, consisting of the current commit ID and a sequence number within that commit (Section 3.4). **(2)** The values are written immediately to the KVSSD, with the Dotori internal keys as the KVSSD keys. *IK0*, *IK1*, *IK2*, and *IK3* in the KVSSD refer to earlier versions of *A*, *B*, and *C*, which were written previously. *IK0*, *IK1*, and *IK3*, still exist on the KVSSD until a deletion routine deletes them. *IK2* has already been deleted. **(3)** After the writes complete, the values is stored in a variable length KV pair cache, indexed by the internal key. **(4)** The translation of user keys to internal keys (e.g. *A* to *IK4*) are then stored in memory in the index update buffer.

Once the index update buffer reaches a threshold of unique entries, it is flushed by applying the updated mappings from user key to internal key to the OAK-tree. **(5)** Updates to keys *A*, *B*, and *C* are applied in-memory to the OAK-tree. The mapping of user key *A* to *IK4* is written to node 25, whereas the mappings for *B* and *C* are written to node 27. **(6)** While being applied to the OAK-tree in memory, KV pairs of user key to internal key for *A*, *B*, and *C* will be written to in-memory logs for their respective index node, which are written to the KVSSD once all entries from the index update buffer are applied to the OAK-tree. One outstanding log for both nodes 25 and 27 already existed on the KVSSD, so the log writes are composed of the node ID and "1", representing the second log write for each node. The index update buffer is emptied and the next writes to Dotori repeat the process.

Reads The provided user key for a read is first searched inside the index update buffer. If it is found, the internal key is used to first check for the key-value pair in the KV cache. If it is found in the cache, the value is returned. If not, the internal key is used directly to retrieve the key-value pair from the KVSSD. If the translation from user key to internal key is not found in the index update buffer, the OAK-tree is checked in memory. If the index node belonging to the key-value pair exists in memory, the internal key is read from the node, and the key-value cache check is repeated as above. If the index node does not exist in memory, Dotori collects the logs for the node and builds the index node in-memory. The node is then searched for the internal key and the key-value cache check proceeds once again.

Concurrency The index buffer is sharded according to a user specified amount of shards, and reads or writes of KV pairs will be directed to a shard of the index buffer based on a modulo of a hash of the key. A read or write to an index buffer shard will lock the shard, meaning reads can block writes. However, as neither update nor search of the index buffer require access to storage , both operations are fast, and we find that synchronization on the index update buffer is of little concern with a sufficient prime number of shards.

The OAK-Tree cache in memory is structured in a similar way. Requests to read a node of the OAK-Tree will be be again directed to a shard, and thus reads can briefly block reads. However, reads do not hold a lock on an OAK-Tree cache shard for the entire duration of the user's request; a private copy of each node is stored in the Dotori handle the user initiates the reads from, of which there can be many opened concurrently. The shard lock is held for the duration of time required to copy the node to this private buffer. When the index update buffer is flushing updates to the OAK-Tree, similar private copies of index nodes are taken before writeback at the end of the flush. Thus index buffer flush writes can briefly block OAK-Tree reads.

In the current version of Dotori, index buffer flushes are single-threaded and block incoming writes, however reads can still be serviced at all times.

3.4 Dotori Features

No Compaction for User Data When a user overwrites a KV pair in Dotori, the old version of that KV pair is placed on a stale list and soon deleted by a background deletion thread. Using the KV interface, we can simply tell the KVSSD to drop the pair from its mappings with no data copies and rewrites from host to device required, sidestepping a large cause of performance issues in block-device KV stores. This does not mean that deletes are free, as inside the KVSSD copies and rewrites may happen as a result of deletes, depending on the implementation of KVSSD. Nevertheless, as data

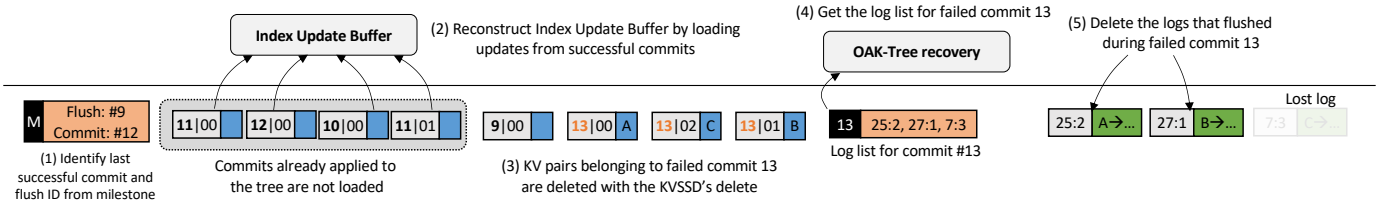


Figure 4: Recovery in Dotori.

transfer between host and storage device is avoided, this enables aggressive space reclamation.

Versioning In Dotori, key-value pairs are not written to the device with the original user key. Instead, they are written to the device with an 8 byte key that is a combination of the current commit ID (4 bytes), and the current sequence number within the commit (4 byte). This is done to support multiple versions of data, enabling snapshots and crash recovery.

Commits and Index Buffer Flushing Commits in Dotori may or may not trigger a flush of the index buffer. Typically, the size at which the index buffer is flushed will be a multiple of the commit interval, for example a 64K commit window and 128K index buffer flush window. Commits that do not flush the index buffer in Dotori persist a header and these headers are used to decide how many writes per commit are to be read for the index buffer rebuild on recovery. Commits which do trigger a flush of the index buffer also trigger a background write of the index node logs, and the index buffer is not declared completely flushed until after all of the logs are successfully written.

Crash Consistency and Recovery Writes in Dotori are considered persistent only after commit is called. Once a commit returns, the user is guaranteed that the KV pairs they wrote in the commit are on the disk, and Dotori will be able to recover to that exact point. If the store crashes between commits, the KV pairs that reached the device are considered invalid, and will be deleted during recovery. Dotori achieves this crash consistency by persisting 3 special key-value pairs each commit. A milestone KV pair, which denotes the current commit ID and commit ID of the last index buffer flush, a commit header KV pair, which contains the amount of KV pairs written in this commit (among other store metadata), and a KV pair that contains a list of the logs to be flushed in this commit (if the index update buffer is flushed this commit). How these pairs are used for recovery is described below.

First, the index update buffer must be rebuilt. Recall that the index update buffer contains mappings of user keys to internal keys on the KVSSD. Upon a crash, the index update buffer must be re-built as it contains entries that will be flushed to the main index in a future commit. Therefore, the index update buffer recovery process is from the last index update buffer flush to the last successful commit. When the store crashes, the milestone key-value pair, which contains the commit ID of the last commit, is read upon reopening. Dotori reads the commit headers to discover how many keys were written in each commit. Commit headers are found by reading special key-value pairs corresponding to them, for example COMMIT_10. Next, each key-value pair in each commit is read asynchronously and in parallel. For example, if COMMIT 1's header recorded that there were 64K writes in that commit, Dotori will read KV pairs "1:0" to "1:65535". Although these reads are carried out using internal keys, values in Dotori contain the original user key, which is then used to fill the index buffer with user key to internal key mappings. In contrast to block-device KV stores where blocks or logs are read in sequence to figure out which values are inside each block and what size they are before continuing, Dotori does not need to wait before reading any KV pairs during the index buffer rebuild. This process of index buffer recovery is only possible because of the KV interface provided by the KVSSD; since we already know how keys in the commit are formed, we already know what range of key-value pairs we need to ask the KVSSD to retrieve for us. No ordered checking within larger logical units i.e. blocks is needed to discover what pairs are inside.

Next, KV pairs that belong to the failed commit must be erased. Removing data that belongs to a failed commit is potentially non-trivial when using a KVSSD. Because there is no concept of a file abstraction, Dotori cannot scan backwards to the last successful commit, then discard the unwanted data. In Dotori, an internal auto-flush makes sure that all KV pair writes before the auto-flush is called have reached the disk, and this auto-flush range is used to delete KV pairs that reached the disk when the store crashed, but before the user's commit completed. With an auto-flush set to 320K (the current value), Dotori uses the KVSSD's delete function to delete internal KV pairs with version numbers 0 to 320K from the last successful commit ID + 1.

Finally, logs that are part of an incomplete flush must also be discarded. If Dotori crashes during the log flush process, using the logs from said flush after restarting can be erroneous. Consider a situation where the logs for a node split (child, new sibling, and parent) are to be flushed, but a crash happens and only one, or some combination of two of the logs exists on storage. Reading the logs will create an incorrect tree configuration, as either the parent, child, or new sibling will lack knowledge of the split. If Dotori crashes before the log flush completes, the list of logs is read and the logs are simply deleted using the KVSSD's delete function.

Figure 4 illustrates the recovery process in its whole. A special milestone key-value pair is read and the latest index buffer flush commit ID and latest successful commit ID are loaded. In the diagram, the latest successful commit was 12, meaning any data from commit 13 is to be discarded. The index buffer is reconstructed asynchronously in parallel by reading the data from commits 10 to 12, and then an iterator is opened on commit 13 to delete 13:00 to 13:02. A list of logs that were to be flushed in commit 13 exists on storage, which is then loaded and the logs inside the list are deleted from the KVSSD.

Snapshots When a snapshot is opened in Dotori, the commit at which the snapshot began is recorded and future updates to the OAK-tree first check if a to-be-updated index node belongs to an active snapshot. If the index node belongs to a snapshot, this version of the node is made immutable. Outside of the snapshot, a live copy of the index node is assigned a new version number so it can receive future updates, and the tree outside of the snapshot is updated to reflect the change. Thus, the OAK-tree is copy-on-write for the *first* update to each index node contained in a snapshot. After the first update, updates are written to the new version of the node via logging as usual.

Transactions Dotori's transaction implementation is identical to that of ForestDB's (the store Dotori is derived from), and thus we only briefly overview it. A transaction is opened by the user and writes inside the transaction are written to storage immediately. The writes inside the transaction are placed into the index update buffer as with normal writes, but are tagged as part of said transaction. If the user commits the transaction, the updates will be flushed to the index buffer during the next flush. If the transaction is canceled, the entries are removed from the buffer and the entries written to storage are added to the stale deletion list.

Atomic Read-Modify-Write Dotori does not currently posess an explicit atomic read-modify-write (RMW) command, but we sketch one possible implementation here. A user calls RMW on a KV pair and the update is blindly written to the OAK-tree without first reading the KV pair. The Dotori internal key of said update is added to the OAK-Tree node containing the original key. In this scheme, each value of the user-key to Dotori internal key mapping in the OAK-tree nodes is a list. When the value is read, the list of updates are read in parallel and the value is merged based on user-specified logic. To control memory, the full KV pair can be written back after a certain amount of updates are persisted, and the lists compacted to contain the internal key of the full KV pair only.

KVSSD I/O Dotori uses the kernel driver from the manufacturer of the KVSSD [39], and use the respective KVSSD IOCTLs for I/O. The KVSSD is the same we test in Section 2.3. A basic KVSSD batch command is used for write I/O. This batch command is not a part of the standard KVSSD API, and we were provided with a special firmware that supports it. The batch command supports up to 8 stores under 4K each in one command, and does not support retrieves. When a user writes a key-value pair, the key and value are added to a batch in user-space and the write returns immediately. When a batch is full a payload is constructed and the batch is sent as a single command to the KVSSD driver. For reads, the batch is checked first and if the key-value pair is found, the data is copied back to the user. Otherwise, data is retrieved from the KVSSD. This means that data inside a batch (up to 8 writes) will be lost upon a crash, despite the write returning as successful to the user. In Dotori, writes are only guaranteed persistent after a commit finishes. A typical process is a user writes a large number of KV pairs, then calls commit. A batch will flush upon a commit call even if it is not full.

Table 1: YCSB Workloads.

Workload	Composition
A	50% read 50% update
B	95% read 5% update
C	100% read
D	95% read 5% insert (latest distribution)
E	95% scan, 5% update
F	50% read-modify-write 50% read

4 EVALUATION

We first evaluate the performance of the OAK-tree using a series of micro-benchmarks to gain an understanding of its performance characteristics. We then compare Dotori to state-of-the-art and state-of-the-practice block-device KV stores across a number of real world workloads.

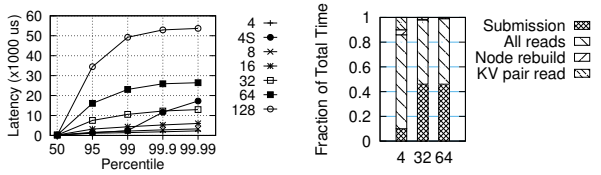
4.1 Experiment Setup

Hardware and Software Throughout these experiments, we use a machine with a 40-core Intel Xeon Gold 5218R CPU and 256 GB of RAM. The operating system is Ubuntu 16.04 with Linux kernel version 4.9.5. The KVSSD and block SSD refer to an identical 1TB NVMe SSD flashed with different firmware (as in 2.3). To ensure tests represent steady-state KV store and device performance, the SSD is pre-conditioned before the YCSB tests, except in the small value test, explained in the next section. The capacity of the block SSD is written twice, once sequential and once random. Then, an ext4 file system is created, and a file is created to fill enough space so that roughly 220GB is left on the device. To precondition the KVSSD, we adapt the block SSD's pre-conditioning method. Two times the total device capacity is written to KV pairs that fill 85% of the device.

Workloads For benchmarking Dotori and the other KV stores, we first test using the YCSB benchmark suite [10]. The YCSB workloads are summarized in Table 1. In the YCSB workloads, we test against three different value sizes; small (100B), medium (4K), and large (16K). The drives are not preconditioned and filled with data in the small value test, as the KVSSD we test in this work pads all values under 1K to 1K to keep the hash table of a manageable size, resulting in a notably skewed amount of data on disk for the same amount of KV pairs between the block-device KV stores and Dotori. This is a current limitation of hash table based KVSSDs that we discuss further in Section 4.4. We test using uniform and zipfian distributions. We limit system memory to 32GB using cgroups, and for each workload we preload enough key-value pairs to reach a roughly 100G store size (7M, 25M, and 750M pairs). We then run the workloads for 30 minutes. A 5 minute warm-up period is carried out before statistics are recorded for each test. 36 total threads are used to perform reads and writes, except in ScyllaDB, as explained in the next section.

Key-Value Stores We compare Dotori to RocksDB, an LSM tree based KV store used widely throughout industry [17], WiredTiger [43], a popular B-tree based KV store which is the default storage engine in MongoDB, ScyllaDB, an LSM-tree based NoSQL store that boasts higher performance due to a sharded design and efficient I/O path [40], and the KVSSD itself. We give each store a 28GB cache (including a simple KV pair cache in front of the KVSSD)

in the YCSB tests, leaving some for the OS and other in-memory housekeeping structures per-store. In RocksDB, we increase the number of background compactions to 8. In WiredTiger, we set the checkpoint period to 30 seconds. For ScyllaDB, we set the number of background threads performing I/O to 28, as the minimum recommended allocated memory per shard is 1GB [41]. ScyllaDB works differently from the other stores in that clients place requests onto background threads that perform asynchronous I/O, not synchronous I/O. For this reason it is advised to have many more client threads than cores. We apply a formula for deciding on how many client threads to create from ScyllaDB from [6] by using Dotori's YCSB B medium value performance as a middle-ground baseline and arrive at 644 workers total. For Dotori, the index buffer flush limit is set to 320K entries, and MAX_LOGS to 4 for a focus on lower latency (Section 4.2). The index node cache in Dotori is set to enough to hold the OAK-Tree in memory in the medium and large values tests, requiring 1GB and 256MB, respectively. In the small value test, we give Dotori enough memory to hold the OAK-Tree's inner nodes, which is 12GB. The KV cache uses the remaining space in each configuration (roughly 29GB for medium and large, and 18GB for small). In configurations where the OAK-tree can reside in memory, KV pair reads still require a read to disk if they are not cached. We leave a detailed analysis of this sizing tradeoff for future work. Dotori is also functional if the index does not fit in memory. Nodes in the OAK-tree are rebuilt via parallel log reading, as described in Section 3.2 and demonstrated next.



(a) Read Latency vs MAX_LOGS **(b) MAX_LOGS latency breakdown**

Figure 5: MAX_LOGS read latency and request breakdown.

4.2 OAK-Tree Performance

In this section, we perform a series of random insert (results not shown), then random read-only workloads while varying Dotori's MAX_LOGS parameter to gain an understanding of the effect of MAX_LOGS on read latency and throughput. Each insert phase randomly inserts 50 million 8B/1024B KV pairs, and each read phase performs 10 million random read operations. For these tests, we set Dotori's index cache size to 820MB, which is enough space to cache all of the OAK-tree's inner nodes, but few of the leaves. We set the KV cache size to 180MB, for a total cache size of 1GB. In such a configuration with a random access workload, most reads will incur a leaf index node rebuild from storage, and a KV pair read from storage. For a clearer picture of Dotori's read latency when storage access for reading index node logs is the dominant factor, we modify Dotori so that logs are *not* cached in memory, and thus each index node rebuild requires a read to storage for every existing log for said node. The OAK-tree's claim is that its KV-interface specific design allows logs for a node to be retrieved in parallel, significantly decreasing the time taken to build an index node from the logs. To verify this claim, we also include the latency of a synchronous implementation of the OAK-tree's log collection for a MAX_LOGS value of 4 where logs for an index node are collected one-by-one, in order.

The results are shown in Figure 5. From a MAX_LOGS value of 4 to a MAX_LOGS value of 16, read latency increases steadily. At a MAX_LOGS value of 4, the P99.99 read latency is low at 2282us. However, once the value of MAX_LOGS increases to 64 and beyond, latency increases sharply. The reason for the large jump in latency is that MAX_LOGS starts to match or exceed the asynchronous I/O queue depth, and collecting a node takes an entire queue (or more) of requests. To verify this assumption, a breakdown of an average of 100 slow requests (requests that fall into the P99 or above latency) is provided for the cases when MAX_LOGS is set to 4, 32, and 64. Each breakdown measures the time to submit all the log read requests, the time spent waiting for all reads to return, the time taken to rebuild the node from the logs, and finally the time to read the actual KV pair the user requested. At a MAX_LOGS value of 4, the time spend reading the logs dominates the overall latency, while submission of all of the log reads is fast. However, at MAX_LOGS values of 32 and 64, log read command submissions only take up to 53% of the total time. The results suggest that the time it takes to submit retrieve commands for each log becomes prohibitively expensive at large values of MAX_LOGS, but introduces the question of whether a batched read command could help with this problem.

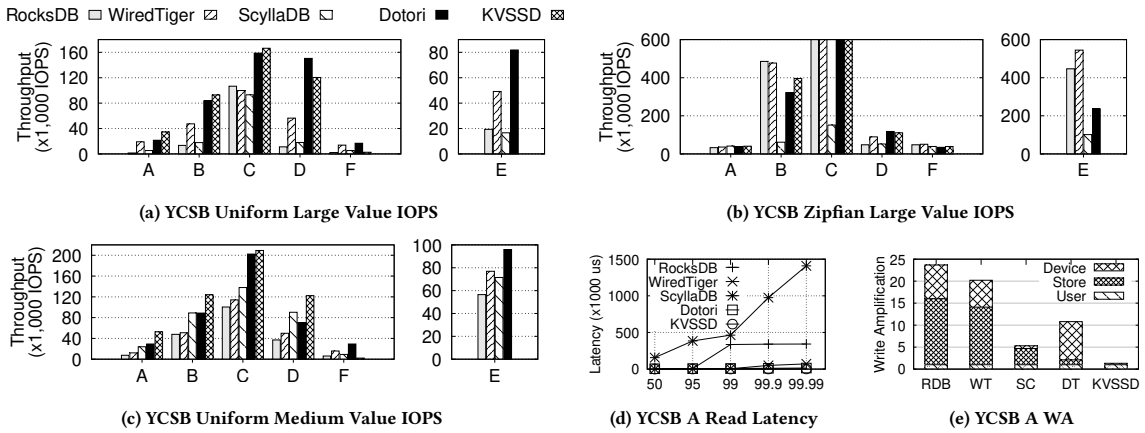
The synchronous implementation achieves similar P50 to P95 latencies to the asynchronous implementation, but the P99 to P99.99 latencies are 1.6×, 6.2×, and 7.5× higher.

4.3 Dotori Performance

Figure 6 presents the results for the YCSB workloads. We discuss the small value results in Section 4.4.

Large Values Dotori outperforms the other stores from 1.3× to 14.1× during the uniform write-heavy workloads, and significantly outperforms the other stores during the read-heavy uniform workloads, outperforming the next best store by 1.95× in YCSB B, and 1.98× in YCSB C. When storage is the bottleneck, as in the uniform distribution tests, a workload with large values favors Dotori for two reasons. First, larger values can imply less KV pairs overall, resulting in a smaller KVSSD hash table. Second, larger values result in more frequent index maintenance operations in other stores, for example, compactions due to full memtables. Despite low write interference in read-heavy YCSB B and D, Dotori still outperforms the other stores. We test the read amplification during YCSB B for each store and see that the next best performing store (WiredTiger) incurs 20× read amplification vs Dotori's 1 (only the value itself is retrieved from the KVSSD). Additionally, Dotori almost matches the KVSSD's raw performance in all large value tests, despite providing several features that the KVSSD does not.

The zipfian test results for Dotori hold across all value sizes, and thus for space we include and discuss the large value test only. Dotori is able to exceed the other stores in performance when storage is the bottleneck, as in the uniform workloads, because these workloads increase the burden of index maintenance for the other stores. However, when locality is high, the index maintenance



(a) YCSB Uniform Large Value IOPS

(b) YCSB Zipfian Large Value IOPS

(c) YCSB Uniform Medium Value IOPS

(d) YCSB A Read Latency

(e) YCSB A WA

Figure 6: Evaluation results from the YCSB benchmarks.

penalties for the block-device KV stores reduces significantly and Dotori can only match the other stores in performance at best due to the nature of the slower KVSSD underneath.

Medium Values As before, Dotori outperforms the other stores from 1.2× to 4.95× during the uniform write-heavy workloads. However, this time ScyllaDB starts to match or exceed Dotori's performance in several of the tests. This gain in throughput comes at a cost; ScyllaDB's CPU usage is consistently high, around 90%, and it suffers from high read latency (shown in the next section).

Dotori underperforms the KVSSD by 2× on YCSB A. The reason for this is the overhead needed to flush logs and delete stale KV pairs during the run. The KVSSD does not need to perform either of these tasks, because it is not providing versioning features and overwrites KV pairs in-place. Both optimizations to Dotori's log flushing scheme (resulting in less log flushes), and improvements to the KVSSD itself (a faster KVSSD means log flushes take less of the overall IOPS) would help with this issue. The KVSSD consistently performs poorly in YCSB F (read-modify-write), suggesting that contention on its inner data structures limits the performance of such a workload.

Read Latency Figures 6d displays the P50 to P99.99 percentile read latencies for the YCSB A run. Dotori achieves 93× lower P99.99 latency than ScyllaDB, 31× and 22× lower P99.9 and P99.99 latency than RocksDB, and 4.64× and 4.72× lower P99.9 and P99.99 latency than WiredTiger. We attribute such low latency to the fact that the OAK-tree's low maintenance overheads reduce contention at the device level, allowing more resources to be prioritized for user KV pair reads, writes, and garbage collection. The KVSSD itself has significantly lower latency than all of the other stores at every percentile, but is not providing the same feature set.

Write Amplification We test write amplification for the YCSB A medium run at the application level and at the device level. For the block KV stores, application level write amplification is recorded using the total amount of data written by a benchmark as seen in /proc/io divided by the actual data written by the user. For Dotori, application level write amplification is calculated by recording the amount of data sent to the KVSSD by Dotori, divided by the data written by the user. For both block KV stores and Dotori, device level write amplification is recorded by dividing total bytes actually written in the SSD (taken from SMART data) by the amount of bytes sent to the device by the benchmark.

Figure 6e shows the write amplification at the application and device level, and the total write amplification (application multiplied by device). Dotori's total write amplification is 2.2× less than RocksDB , 2× less than WiredTiger, and 2.1× higher than ScyllaDB's. Dotori's application level WA is the lowest out of all the stores; updating nodes in the OAK-tree requires only a single write I/O with the log. However, Dotori experiences high device level WA, which ultimately causes it to have a higher total write amplification than ScyllaDB. The reason is due to internal operations needed to manage key-value data in the KVSSD. The KVSSD sees lower internal WA than Dotori as it has less data overall due to the absence of small log writing (giving it more chances to find empty victim blocks for GC), and does not perform stale KV pair deletion (deletions may trigger hash table updates, increasing device-level WA).

Space Amplification To measure space amplification for the block KV stores, we take the size of the directory divided by the total amount of user data. For Dotori, we take the total amount of data in the KVSSD (recorded inside Dotori) divided by the total amount of user data. In the YCSB A medium values run, Dotori's space amplification is the lowest at 1.12×. The next best store, WiredTiger, had a space amplification of 1.23×. Dotori's space amplification is lower than the other stores because the KV interface enables comparatively lightweight stale data removal with the delete command.

Recovery Speed To test recovery speed, each KV store is preloaded with 10 million key-value pairs. Then, we run a 50/50 read/write workload and crash each store in the middle of the workload. The store is recovered and the process is repeated 4 more times for a total of 5 recoveries. RocksDB and ScyllaDB have average recovery times of under 1 second. WiredTiger's recovery time averages to 31 seconds, depending on how close to a checkpoint the crash happened. Dotori's recovery time averages 3.8 seconds, and during recovery Dotori is able to rebuild the index buffer at the maximum IOPS of

the underlying KVSSD. Dotori's average recovery time improves significantly on a naive design where we adapt ForestDB's recovery process to a KVSSD. The naive design takes over 80 seconds to recover a crashed Dotori instance.

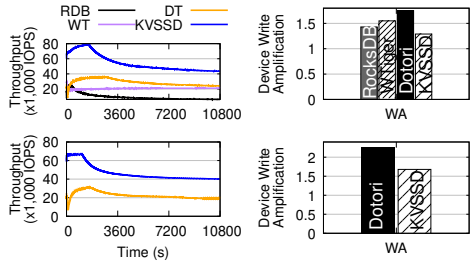


Figure 7: Large Dataset Test Results

Large Dataset Performance We preload each KV store with 65M KV pairs varying from 100B to 16K in size, totaling roughly 530GB of space (55% of the device space exported to the user) and run a 3 hour 50/50 read/write workload with uniform key distribution. ScyllaDB is left out of the results as it ran out of space during the run. Next, we preload Dotori and the KVSSD with 90M KV pairs, occupying roughly 80% of the device. The results are shown in Figure 7. Dotori outperforms WiredTiger by 1.14× and RocksDB by 3.89×. Dotori maintains a throughput penalty compared to the KVSSD due to log flushing and stale KV pair deletes, discussed previously. However, Dotori and the KVSSD maintain an almost identical read latency (not shown).

Between the two tests, Dotori and the KVSSD show similar performance. In the second test, Dotori's WA increases by 1.3×. Despite this, Dotori manages to hold similar, stable performance at both 50% and 80% device occupancy.

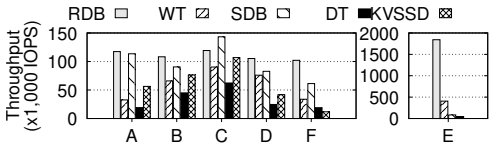


Figure 8: YCSB Small Value Results

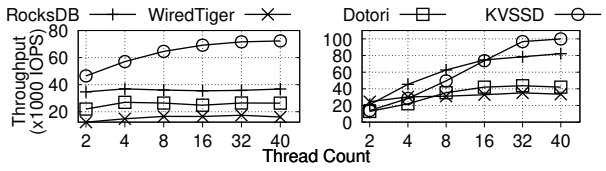


Figure 9: Concurrency Test Results. 100% Update (left) and 50% Update 50% Read (right)

4.4 Tradeoffs and Limitations

Small Value Performance Small value sizes represent a challenge to Dotori, and KVSSDs in general. Given databases of similar size, smaller values results in more KV pairs overall, increasing the size of the hash table inside the KVSSD. A larger hash table means less overall can be stored in memory, resulting in more reads to flash to service KV pair read and writes. Moreover, contemporary block-device based KV stores group writes from different threads in the same commit, allowing multiple KV pair writes to fit inside the same write unit to disk (a block) [44][9]. In comparison, KVSSDs offer no such abstraction, meaning that each KV pair write results in a system call and command sent to disk.

The small value YCSB results are shown in Figure 8. As the KVSSD's performance drops in this configuration, so does Dotori's. Although the KVSSD manages to compete with the other stores, Dotori's OAK-Tree maintenance overheads, while low, are enough of a penalty in this setting that Dotori underperforms the other stores. We discuss a possible solution to this in Section 5.

Small KV Pairs and Device Endurance As shown in the YCSB medium value tests (and, though not shown, the large values tests too), Dotori shows lower *total* write amplification than the other KV stores. However, its device level write amplification is notably higher due to internal KV pair management inside the KVSSD. Despite this, Dotori is able to achieve lower overall write amplification when value sizes are above 1K. However, when values are under 1K in size Dotori will experience high levels of device level write amplification due to how the underlying KVSSD pads all values under 1K to control the size of the hash table. In the YCSB A small value Dotori's device level WA is 8× that of the block KV stores. This limitation is not introduced by or inherent to Dotori, and Dotori would be free of this limitation given a KVSSD that does not have such restrictions.

Sequential Accesses Block SSDs can benefit from logical sequential write and read access for numerous reasons, including reducing mapping costs, reading many KV pairs in the same block, device internal parallelism, and more. However, such a concept is not as straightforward in a KVSSD as the hash table may break ordering of KV pairs. We populate Dotori and the KVSSD by sequentially inserting 25M 4K KV pairs and then perform a sequential read test, then a sequential update test for 25M operations each. We then repeat the test but populate randomly and perform random reads and updates. Performance is identical between sequential and random accesses. Neither Dotori or the KVSSD benefit from sequential accesses as what are sequentially written KV pairs in the mind of the user aren't seen as sequential at the disk level. This has negative performance implications for workloads that benefit from sequential accesses to large amounts of data, such as in replicated environments where fast reads of the entire disk are used to copy data to new nodes.

Concurrency We perform a 10-minute update-only and 10-minute 50/50 read/write workload while increasing thread counts to test concurrency. The results are shown in Figure 9. Index buffer flushes impede scalability in Dotori in the update-only test, whereas the KVSSD scales up to 32 threads. The block-device KV stores likewise suffer scalability issues due to synchronization around logs [9] (or at the filesystem [31]). Dotori scales better in the read/write test, as writes do not block reads and index buffer flushes are less frequent, but ultimately does not scale as well as the KVSSD again due to index buffer flushes. Allowing writes to proceed while a previous index buffer flushes is marked as future work for Dotori, and the OAK-Tree is compatible with either design.

Key and Value Sizes Dotori currently supports 8 byte or smaller keys only. This is a limitation of the in-memory B+Tree Dotori inherits from ForestDB, and not a limitation introduced by the OAK-Tree design. Dotori could accept keys longer than 8 bytes if modified to work with an in-memory B+Tree that supports them.

5 DISCUSSION

Importance of a Batch Command Throughout this work we identified several areas where a batch command could be critical in helping KVSSDs and Dotori, and believe such a command should be present in KVSSD API specifications moving forward. For example, log flushing in Dotori reduces performance due to writing large numbers of small logs, but a batch command supporting many stores would likely help performance notably. Additionally, Dotori's write performance suffers as value sizes get smaller due to having no mechanism to group writes together into a single NVMe command.

Make scans accurate or performant, or do not support them at all The KVSSD we tested in this work supported a limited iterator command in which a 4 byte prefix could be matched with key-value pairs on the device. A previous iteration of Dotori used this function to find KV pairs from a failed commit to delete, but it was ultimately too slow and replaced with the internal auto-flush. The iterator function was not fast or responsive enough to be used elsewhere, and was too fuzzy to be used as the main method for range queries. Given the choice of whether to have this limited iterator function or more IOPS and lower latency, we would have chosen the latter. We suspect maintaining key prefix buckets for iteration was a large contributor to the lack of performance vs the block SSD.

Dotori on other types of KVSSD We leave the testing of Dotori on other types of KVSSD (e.g. LSM-tree based [19]) for future work. To the best of our knowledge, the KVSSD we test in this work is the only available production-ready KVSSD on enterprise hardware, which makes comparison with other types of KVSSD infeasible as they are prototypes that are generally unavailable, and the performance of the hardware varies significantly [21][19]. Dotori is still compatible with other types of KVSSD, assuming they have a store, retrieve, delete, and list or iterator function. However, as LSM-trees have first-class scan and potentially versioning support, it is unclear whether the full feature set of Dotori would be necessary on such a device. The stance we take in this work is that hash table based KVSSDs with fast KV access and less auxiliary features have advantages over LSM-tree based KVSSDs (Section 2.4).

6 RELATED WORK

Key-Value Stores Recent KV store designs frequently center around improving the LSM-tree. SILK [5] reduces the interference between client and background operations for compaction. PebblesDB [37] introduces an LSM-tree modification called the fragmented LSM-tree to avoid data rewrites within LSM-tree levels. WiscKey [29] introduces a technique wherein keys and values are separated in the LSM-tree, reducing compaction overhead. FASTERKV [8] is a KV store that enables a mix of in-place and out-of-place updates while achieving high concurrency. The design of ScyllaDB [40] departs from synchronous I/O in the era of very fast disks to achieve high throughput, yet keeps the LSM-tree design. Dotori is different than these stores because it uses a KVSSD as the underlying storage device. Instead of developing new techniques to reduce inefficiencies whose root cause is the block interface, Dotori does not use the block interface at all, and instead uses a KVSSD and the key-value interface to implement a novel index storage design.

Key-Value SSDs Several KVSSD designs [19, 21, 27, 30, 45] and works that utilizes a KVSSD [25] have appeared in recent times. KAML [21] is a KVSSD that exposes a transaction interface to the user, but lacks iterator functionality that would be critical to Dotori. PiNK [19] is a KVSSD based on an LSM-tree design, as opposed to a hash table design like in our tested KVSSD. While these KVSSD designs make efforts to enable further native functionality on the device itself, this work takes a different direction. We do not focus on providing features in KVSSD hardware, and rather use an underlying KVSSD with basic KVSSD functionality as a fast key-value storage device instead of a fully-featured KV store.

B+tree Design B+trees power many databases used throughout industry and academia [3, 32, 35, 43]. All of these B+trees are designed for block devices and to one extent or another suffer from the problems we discuss in this work. In comparison, our work introduces a novel B+tree-like design in the OAK-tree that is designed for a KVSSD, not a block device. Dotori's OAK-tree shares similarities with the log-structured B-tree (LSB-tree) [23] and B^{ϵ}-tree[20]. Like the OAK-tree, the LSB-tree aims to append updates to index nodes in a log. However, unlike the OAK-tree, the LSB-tree needs to read in logs to memory before updating them. Additionally, mapping information for finding logs on disk must be maintained and persisted. The B^{ϵ}-tree appends data without first reading a node, similar to the OAK-tree, but must append said data to the root and flush data towards the leaf later, unlike the OAK-tree which appends data directly to the leaf. The OAK-tree's design is enabled by the KV interface, which makes it practical to persist and later find data sparsely across the disk.

7 CONCLUSION

The adoption of KVSSDs suffers due to the lack of features they provide that are critical to operation in real world scenarios. However, key value SSDs operate at lower raw device performance, and cannot compete with mature block device based key-value stores without new techniques and designs. In this work we introduced Dotori, a KVSSD based key-value store that uses the key-value interface to facilitate higher performance than state of the art block SSD based key-value stores through a novel host-side index design.

ACKNOWLEDGMENTS

We thank our shepherd and the reviewers for their invaluable help with the work. This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773), an Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korean government (MSIT), and an Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (23ZS1310). This work was also supported in part by a research grant from Samsung Electronics.

REFERENCES

[1] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2015. Forestdb: A fast key-value storage system for variable-length string keys. *IEEE Trans. Comput.* 65, 3 (2015), 902–915.

[2] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. 2012. Storage infrastructure behind Facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.* 35, 2 (2012), 4–13.

[3] Apple 2022. FoundationDB. https://www.foundationdb.org.

[4] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC)*. 363–375.

[5] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC)*. 753–766.

[6] Benchmarking ScyllaDB 2022. Best Practices for Benchmarking ScyllaDB. https://www.scylladb.com/2021/03/04/best-practices-for-benchmarking-scylla/.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

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

[10] 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*. 143–154.

[11] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[12] 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. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[13] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[14] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 25–36.

[15] Facebook 2022. RocksDB. https://www.rocksdb.org.

[16] Facebook 2022. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[17] Facebook 2022. RocksDB Users. https://github.com/facebook/rocksdb/blob/master/USERS.md.

[18] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*. 651–665.

[19] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC)*. 173–187.

[20] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST)*. 301–315.

[21] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–384.

[22] 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*. 144–154.

[23] Bo-kyeong Kim and Dong-Ho Lee. 2015. LSB-Tree: a log-structured B-Tree index structure for NAND flash SSDs. *Design Automation for Embedded Systems* 19, 1 (2015), 77–100.

[24] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[25] 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 *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 75–92.

[26] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[27] 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 *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 384–395.

[28] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. Kvell+: Snapshot Isolation without Snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 425–441.

[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[30] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *2015 USENIX Annual Technical Conference (USENIX ATC)*. 207–219.

[31] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 71–85.

[32] MongoDB 2022. MongoDB. https://www.mongodb.org.

[33] Netflix 2016. Application data caching using SSDs. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef.

[34] NVMe 2022. NVMe Command Set Specifications. https://nvmexpress.org/developers/nvme-command-set-specifications/.

[35] Michael A Olson, Keith Bostic, and Margo I Seltzer. 1999. Berkeley DB.. In *USENIX Annual Technical Conference, FREENIX Track*. 183–191.

[36] Percona 2022. Percona Fractal Tree. https://github.com/percona/PerconaFT.

[37] 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 (OSDI)*. 497–514.

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

[39] Samsung 2022. Samsung OpenMPDK KVSSD Software and Drivers. https://github.com/OpenMPDK/KVSSD.

[40] ScyllaDB 2022. ScyllaDB. https://www.scylladb.com.

[41] ScyllaDB System Requirements 2022. ScyllaDB System Requirements. https://docs.scylladb.com/stable/getting-started/system-requirements.html.

[42] SNIA 2022. SNIA Key Value Storage API Specification. https://www.snia.org/keyvalue.

[43] WiredTiger 2022. WiredTiger. https://docs.mongodb.com/manual/core/wiredtiger/.

[44] WiredTiger Group Commit 2022. WiredTiger Group Commit. https://source.wiredtiger.com/develop/tune_durability.html.

[45] Shuotao Xu et al. 2016. *Bluecache: A scalable distributed flash-based key-value store.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[46] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[47] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC)*. 17–31.