PAPER • OPEN ACCESS

HybridKV: An Efficient Key-Value Store with HybridTree Index Structure Based on Non-Volatile Memory

To cite this article: Chen Ding *et al* 2021 *J. Phys.: Conf. Ser.* **2025** 012093

View the article online for updates and enhancements.

HybridKV: An Efficient Key-Value Store with HybridTree Index Structure Based on Non-Volatile Memory

Chen Ding[1], Jiguang Wan[1,2*] and Rui Yan[1]

[1] WNLO, Huazhong University of Science and Technology, Wuhan, China
[2] Shenzhen Research Institute of Huazhong University of Science and Technology, Shenzhen, China
Email: jgwan@hust.edu.cn

Abstract. Non-Volatile Memory (NVM) is a new type of storage media with non-volatile data, higher storage density, better performance and concurrency. Persistent key-value stores designed for earlier storage devices, using Log-Structured Merge Tree (LSM-Tree), have serious read-write amplification problem and do not take full advantage of these new devices. Existing works on NVM index structure are mostly based on Radix-Tree or B+-Tree, index structure based on Radix-Tree has better performance but takes up more space. In this paper, we present a new index structure named HybridTree on NVM. HybridTree combines the characteristics of Radix-Tree and B+-Tree. The upper layer is composed of prefix index nodes similar to Radix-Tree, which is indexed by the key prefix speed up data locating, and providing multi-thread support. The lower layer consists of variable-length adaptive B+-Tree nodes organizing key-value data to reduce space waste caused by node sparseness. We evaluate HybridTree on a real NVM devices (Inter Optane DC Persistent Memory). Evaluation results show that HybridTree's random write performance is 1.2x to 1.62x compared to Fast & Fair and 1.11x to 1.52x compared to NV-Tree, with 54% space utilization reduced compared to WORT. We further integrate HybridTree into LevelDB to build a high performance key-value store HybridKV. By storing HybridTree directly on NVM, the problem of read and write amplification of LSM-Tree is avoided. We evaluate HybridKV on a hybrid DRAM/NVM systems, according to the results, HybridKV can improve random write performance by 7.5x compared to LevelDB and 3.23x compared to RocksDB. In addition, the random read performance of HybridKV is 7x compared to NoveLSM.
Keywords. Key-value store; non-volatile memory; log-structured merge tree; hybrid-tree index; prefix index.

1. Introduction

With its high performance, high availability, and high scalability, key-value stores (KVs) have become the main storage technology in current data centers, which support a wide range of cloud applications [1]. In the other hand, with DRAM-like performance, disk-like persistency, and higher compacity, Non-Volatile Memory (NVM) is becoming the core component for building high performance applications, bringing new opportunities and challenges to the design of KVs. In this paper, we are targeting the design of index structure for KVs on NVM.

In write-intensive scenarios, log-structured merge trees (LSM-Trees) [2] are the backbone index structure for persistent key-value stores, such as RocksDB [3], LevelDB [4], HBase [5], and Cassandra [6]. LSM-Trees defer and batch write requests in memory to exploit the high sequential write bandwidth of storage devices. Figure 1 shows a classical LSM-Tree structure used by LevelDB which is composed of a DRAM component and a disk component. To serve write requests, writes are first batched in DRAM by two skiplist (MemTable and immutable MemTable), then the MemTable is flushed to L_0 on disk generating Sorted String Tables (SSTables). To deliver a fast flush, L_0 is unsorted where key ranges overlap among different SSTables. SSTables are compacted from L_0 to deeper levels $(L_1, L_2 \dots L_n)$ through background jobs called compaction making each level sorted (except L_0) thus bounding the overhead of reads and scans. This design applies updates out-of-place to reduce random I/Os thus suitable for the storage devices where sequential access is much faster than random access. However, it needs period background jobs to maintain the tree shape which introduce large read/write amplification. When we test LSM-Tree on NVM, we find that the overhead caused by compaction is more significant, as shown in figure 2. LSM-Tree do not take full advantage of NVM, which requires a new index structure for KVs on these devices.

Current research for index structure on NVM can be classified into two categories: tree-based index structure and hash-based index structure. Hash-based index structures have excellent single-point read/write performance, but does not support complex range queries, thus not suitable as an index structure for KVs. The tree-based index structures mainly include: B/B+-Tree-based, Radix-Tree-based, LSM-Tree-based, etc. As mentioned before, LSM-Tree-based index structures have serious read/write amplification problem, while the index structures based on B/B+-tree and Radix-Tree have its own limitations. Our experimental studies show that Radix-Tree-based index structure has better performance than B/B+-Tree-based index structure but take up more space.

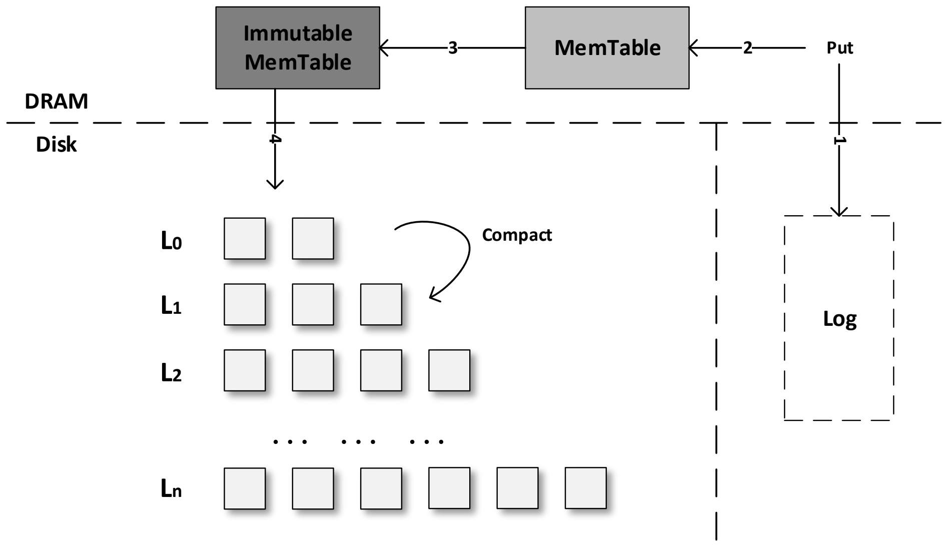


Figure 1. Structure of an LSM-Tree.

Targeting these challenges, this paper proposes HybridKV, a high performance KVs for systems with DRAM-NVM storage. The design principle behind HybridKV is (1) integrate the advantages of Radix-Tree and B+-Tree to build a high performance index structure with little space occupancy named HybridTree, and (2) leverage NVM CPU's direct access, byte addressing and high concurrency characteristics to build a high performance KVs based on HybridTree.

2. Background

In this section, we present the necessary background on NVM, KVs for NVM, and the challenges and motivation in optimizing KVs for NVM.

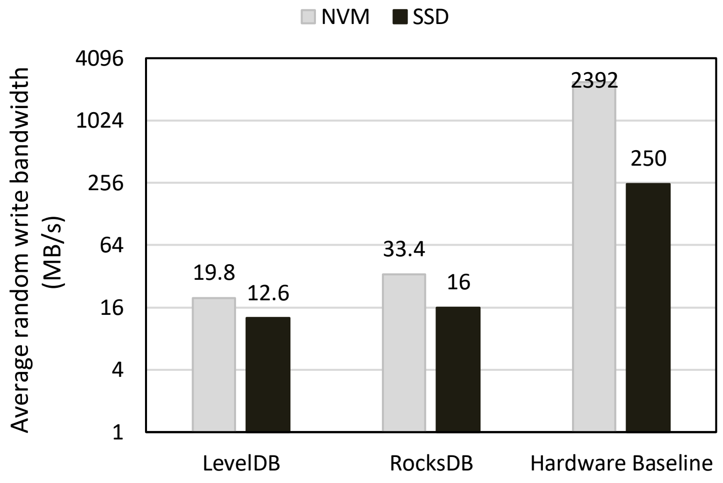


Figure 2. Random write bandwidth comparison of LevelDB, RocksDB and Hardware Baseline on NVM and SSD.

2.1. Non-Volatile Memory

Non-volatile memory (NVM) is a new type of storage device that has the characteristics of non-volatile, high storage density, low static power consumption, and bytes addressing. Nowadays, it mainly includes phase-change memory (PCM) [7], ferroelectric random access memory (FeRAM) [8], magnetic random access memory (MRAM) [9], resistive random access memory (RRAM) [10], and memristor [11]. Due to its DRAM-like performance, disk-like persistency and high compacity, it becomes a cost-efficient choice to build storage systems.

NVM works either as a persistent block device or as main memory. For the latter, although the performance of NVM is close to that of DRAM, there are problems such as asymmetric reads and writes, data security, and data consistency caused by the mismatch of the CPU cache line and NVM update granularity. A lot of modifications are required to adapt to upper-layer applications or operating systems. Therefore, using NVM directly as main memory is not compatible. Another way is to use NVM as a supplement to DRAM, which requires to find a balance between DRAM and NVM, thus has not yet been widely used. As a result, HybridKV focuses on efficiently using NVM as persistent block device in a hybrid system of DRAM and NVM.

2.2. KV Stores on NVM

With the rapid development of non-volatile memory and the requirements for high-performance massive data storage, using NVM to build KVs has become a hot research topic.

Kannan and Kaiyrakhmet proposed NovelLSM [12] and SLM-DB [13] respectively, which are KVs for systems with hybrid storage of DRAM, NVMs, and SSDs. NovelLSM reduce write stall by putting a bigger MemTable in NVM. However, when the dataset size exceeds the compacity of NVM MemTables, flush stall still happens. SLM-DB propose a single-level LSM-Tree on SSD with a B+-Tree in NVM to provide fast read, but it introduces overhead of maintaining the consistency between B+-Trees and LSM-Trees.

Xia et al. proposed HiKV [14] for the hybrid memory of DRAM and NVM, which provide excellent read/write performance. HiKV combined the advantages of B+-Tree and hash list to build a hybrid index structure, it uses hash list on NVM to provide fast single-point read/write and build a B+-Tree on DRAM to support range query. However, its fixed-size hash partitions are not flexible for data growth databases.

Yang et al. proposed NVStore [15], using NVM as persistent storage, to reduce the consistency overhead and improve the CPU cache hit rate. NVStore use a B+-Tree based index structure named

NV-Tree that decouples the intermediate node and the leaf node, only ensure the consistency of the leaf node and reconstruct the intermediate node by leaf node when the system restarts, thereby reducing the consistency overhead. NV-Tree keeps the data in leaf node unsorted to reduce data migration caused by sorting operations, thus reducing NVM writes. To improve the efficiency of CPU cache, NV-Tree organizes internal nodes in a cacheline optimized format. Hwang el al. continued to optimize the B+-Tree on NVM and proposed the Fast&Fair [16]. Keeping data sorted in nodes can bring better search performance, but ensuring that the write operations arrive in the NVM in an orderly manner during sorting will cause greater consistency overhead, such as the call of memory fence (mfence) and cache line refresh (cflush). Fast&Fair uses the feature of "there are no two duplicate pointers in the B+-Tree node" to construct an inconsistent state that can be ignored, reducing the call of mfence and clfush, thereby reducing the cost of data consistency. In addition, by modifying the operation mode of read and write in the same node, Fast&Fair's search operation does not need to be locked, which improves concurrency.

Lee et al. optimized the Radix-Tree structure on the NVM single-tier storage system and proposed WORT [17] to reduce NVM writes. The structure of the traditional Radix-Tree is determined by the prefix of the inserted key, and does not require tree rebalancing operations and node granularity updates. It has high read/write performance but low memory space utilization. WORT designed a failure atomic path compression strategy on this basis, adding 8 bytes metadata to the node header to ensure structural consistency, so that there is no need to log or copy-on-write, thus reducing NVM writes. In addition, by sorting the common prefix in the node head, and deleting some intermediate index path nodes, the NVM space utilization is improved.

2.3. Challenges and Motivations

To explore challenges in index structure for KVs on NVM, we conduct a preliminary study on the LSM-Tree based KVs. We use the FIO tool to test the basic random write performance of NVM and SSD devices, and then use the db_bench tool to test the random write performance of LevelDB and RocksDB on NVM and SSD respectively. During the test, a total of 80GB data was written, and each key-value pair was 8B and 1KB. The evaluation environments and other parameters are described in section 5. As shown in figure 2, NVM random write performance is 9.6x that of SSD in the hardware basic performance test, but the random write performance of LevelDB and RocksDB on NVM is only 1.6x and 2.1x higher than when they run on SSD, reflecting that the existing LSM-based KVs cannot take full advantages of NVM devices.

We further use db_bench to test the performance and space occupancy of various state-of-art index structures on NVM, including B+-Tree based Fast&Fair, Radix-Tree based WORT, and skiplist in LSM-Tree. In this experiment, an 100GB dataset of 8B-256B key-value items are loaded, and then get and delete one million of them. Figure 3 shows that the performance of WORT is the best on NVM with nearly 2x random write throughput that of Fast&Fair and about 5x that of skiplist, and the random read and delete performance also has obvious advantages over the two. During the test, we also recorded the space occupation of each index structure. WORT's space occupation is more than 2x that of Fast&Fair under random write workload, as shown in figure 4. Therefore, these index structures have their own limitations.

Motivating by these observed challenging issues, we propose HybridKV that aims at providing high performance KVs with high index space utilization, as elaborated in the next section.

3. Design

In this section, we present HybridKV, an HybridTree based key-value store for systems with two-tier DRAM-NVM storage. Figure 5 shows the overall architecture of HybridKV. From top to bottom, (1) DRAM retains the MemTable and Immutable MemTable of LevelDB, key-value pairs are first written to the write-ahead log on NVM, and then are inserted into the MemTable, when MemTable is full, it will be transformed into an Immutable MemTable, waiting to flushed to the HybridTree on NVM. HybridKV implements multi-threaded concurrent flushing from MemTable to HybridTree (§3.1). (2)

NVM contains a three-layer HybridTree structure to index and store key-value pairs. The uppermost layer of HybridTree consists of a single-layer large-capacity prefix index node, which divides the whole key space into more fine-grained key ranges to provide concurrent access. Below is a two-level dynamic combination tree called DynamicTree. The upper layer of the DynamicTree is an index layer similar to Radix-Tree to accelerate key search, while the lower layer is composed of multiple B+-Trees with variable-length adaptive nodes to improve space utilization (§3.2).

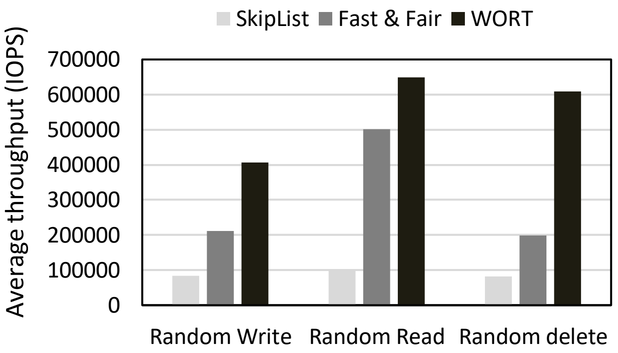


Figure 3. Random write, read and delete throughput of SkipList, Fast & Fair and WORT.

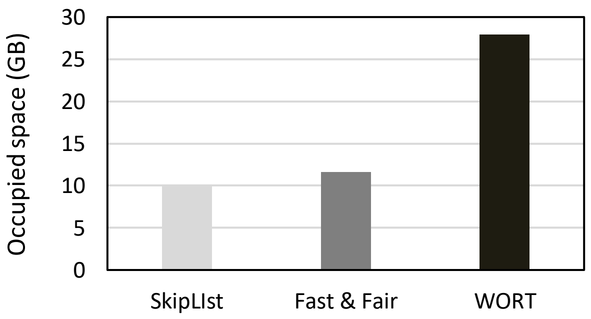


Figure 4. Comparison of space occupied by different index structures.

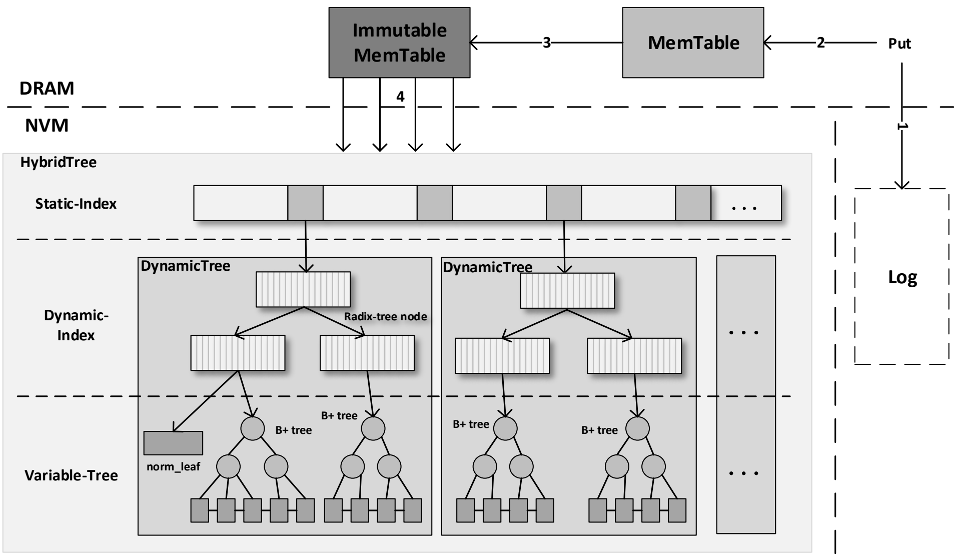


Figure 5. Overall architecture of HybridKV.

3.1. Multi-thread Flush for MemTable

In LevelDB, the data in MemTable and Immutable MemTable is organized by a skiplist, which presents a multi-level linked list structure as shown in figure 6. The bottom L_0 level of the skiplist is a singly linked list. As the height increases, the number of index nodes in each level gradually decreases. This multi-level index structure can approximate the binary search of the linked list with $O(logn)$ time for insert, delete, update, and get operations. In MemTable, insert and delete operations are to add a key-value pair, the actual delete operations will be postponed to the completion of the background flush process.

HybridKV inherits the structure with adding support for multi-thread flushing. Suppose the number of index nodes in level i of the skiplist is N_i $(0 \leq i < maxHeight)$ and the number of flush threads given is M $(0 \leq \mathrm{M} < \mathrm{maxThreadsNum})$. We scan from the top to the bottom to find the first level i that meets the conditions:

$$N_i - 1 \leq M < N_i \; (1 \leq i < maxHeight)$$

And then, the index nodes of level i are selected to divide the MemTable into $N_i + 1$key ranges, with each key range corresponding to a flush thread. An example is shown in figure 6. Assuming 4 threads are given, scan from top to bottom, we find L_1level has 3 index nodes, dividing the MemTable into 4 segments, which is exactly the number of flush threads, so we choose L_1 as the target level. Multi-thread flushing can make full use of the high bandwidth of NVM devices, thereby reducing write stalls caused by MemTable full.

3.2. Structure of HybridTree

When the MemTable in DRAM is full, the key-value pairs are persistently stored in NVM through the HybridTree. HybridTree has excellent read/write performance with low space occupation and read/write amplification. This section introduces the design of HybridTree from tree structure, consistency, and concurrency.

Tree structure. The HybridTree is divided into three layers in vertical direction: static index layer called Static-Index, dynamic index layer called Dynamic-Index, and Variable-Tree with variable nodes, as shown in figure 5. The Static-Index is a single-layer statically allocated array. By combining the nodes of the first n layers of the original Radix-Tree, the number of layers is effectively reduced, improving the indexing efficiency. The Dynamic-Index is composed of prefix index nodes similar to the Radix-Tree, providing fast prefix-based indexing. Variable-Tree is used when the depth of the node in Dynamic-Index exceeds the setting threshold. The Variable-Tree is composed of multiple B+-Trees with variable length nodes, which can reduce space waste by flexibly adjusting the size of the nodes. Dynamic-Index and Variable-Tree form a dynamic combined tree called DynamicTree.

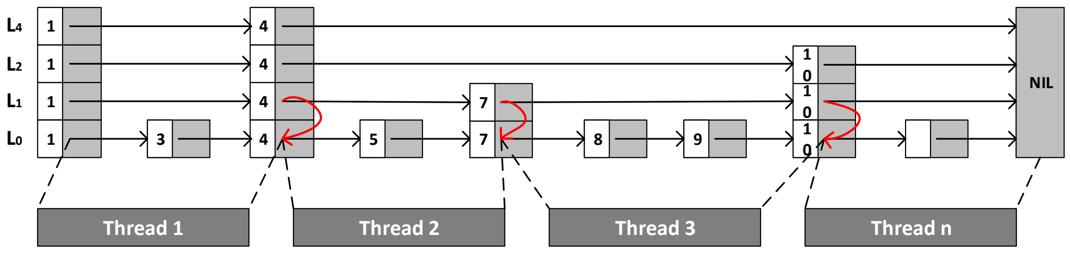


Figure 6. Multi-thread flush for MemTable.

Static-Index uses single-layer large-capacity nodes to replace the first n-layer (default $n = 3$) nodes of the original Radix-Tree. Suppose that the original Radix-Tree uses m bits (default $m = 4$) prefix as an index, then the current single-layer node uses p bits ($p = m * n$) prefix as the index, that is, the first p bits of the key is used to locate the lower-layer node during key search, and this layer contains a total of 2^p indexes. In addition to the indexes, the Static-Index also contains metadata information such as capacity and locks. The Static-Index can effectively reduce the number of layers of the original Radix-Tree, thus improving the performance of indexing.

Dynamic-Index is composed of path-compressible Radix-Tree nodes called crt_node16 proposed by WORT, which can reduce the space waste of the index structure. However, this structure still wastes a lot of space as analyzed in §2.3. Therefore, HybridKV uses Variable-Tree based on B+-Tree to organize the sparsely distributed indexes to further reduce space waste. The nodes exceeding the setting threshold will be converted to B+-Tree nodes as shown in figure 7.

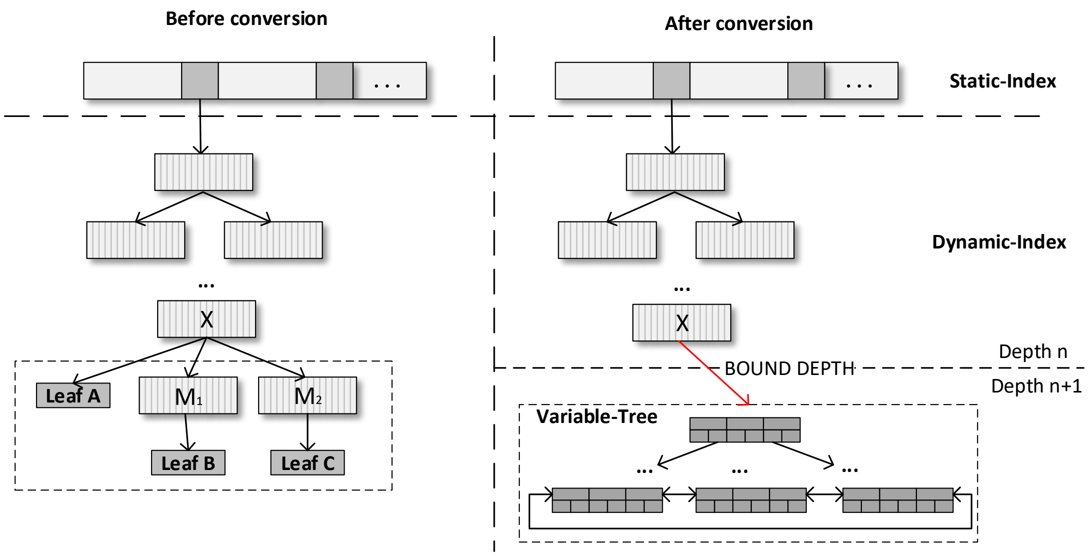


Figure 7. Node conversion of HybridTree.

Variable-Tree contains four types of nodes with increasing sizes: Node2, Node4, Node8 and Node14. Node14 occupies 256B to ensure the size of all nodes does not exceed the minimum effective access granularity of NVM. Each type of node contains the same header field with the difference lies in the size of the key and sub-pointer arrays. Node2 means that the number of keys and sub-pointer arrays in the node is 2. A root node of type Node2 is created during initialization. With the insertion of data, the node will gradually expand into Node4, Node8, and Node14, a split operation is triggered when the node capacity exceeds the size of Node14. The shrinking process is just the opposite.

Consistency. The write-ahead log ensures that the data written to the MemTable is not lost. As for the data written to the HybridTree, we only need to consider the collapse consistency of the DynamicTree since the Static-Index is statically allocated and will not be modified after initialization. For the Dynamic-Index, the 8-byte header can guarantee the updates for the pointer is atomic. For the Variable-Tree, 8-byte atomic writing is used to ensure its failure atomicity, when insert data, first move the pointer atomically, and then move the data.

Concurrency. The consistency of concurrent access is guaranteed by read-write lock. The Static-Index can be accessed in parallel with no need to be locked, and the DynamicTree pointed by the Static-Index is used as the lock granularity to achieve high parallelism of the system. The DynamicTree bound to the write lock no longer accepts other write requests, but different DynamicTree can serve concurrent write requests. The same DynamicTree bound to the read lock can accept other read requests, as shown in figure 8.

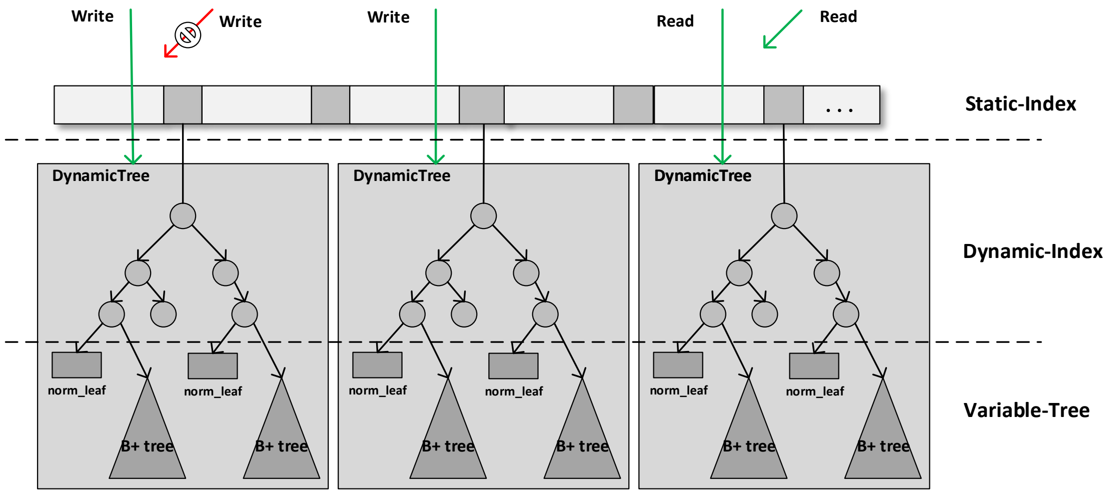


Figure 8. Current access of HybridTree.

4. Implementation

We implement HybridKV based on the popular KV engine LevelDB. HybridKV accesses NVMs via the persistent memory development kit (PMDK) [18], with which the systems can use the DAX function to allow application to access persistent memory as a memory-mapped file. Next, we briefly introduce the space management scheme of NVM and the read/write processes.

Space management of NVM. HybridKV divides the NVM space into 5 areas: metadata area, static index area, log area, reserved area, and data area. The metadata area corresponds to a fixed-size reserved space at the start position of the NVM, using to store metadata information such as space utilization and management structure. The static index area corresponds to a small piece of continuous static space after the metadata area, using to store Static-Index. It is initialized when the system starts, and will not be changed afterwards. The log area is used to store the write-ahead log and can be released after the log is deleted. The data area stores other index nodes and key-value pairs in the HybridTree. Index nodes include three types: Radix-Tree node (crt_node16) with 16 pointers, normal leaf node (norm_leaf) and Variable-Tree node (vbpt_node). The existing persistent memory allocator performs poorly when allocating a lot of small objects. In order to make better use of the NVM space, we implement a space allocation strategy suitable for a variety of different node sizes, as shown in figure 9.

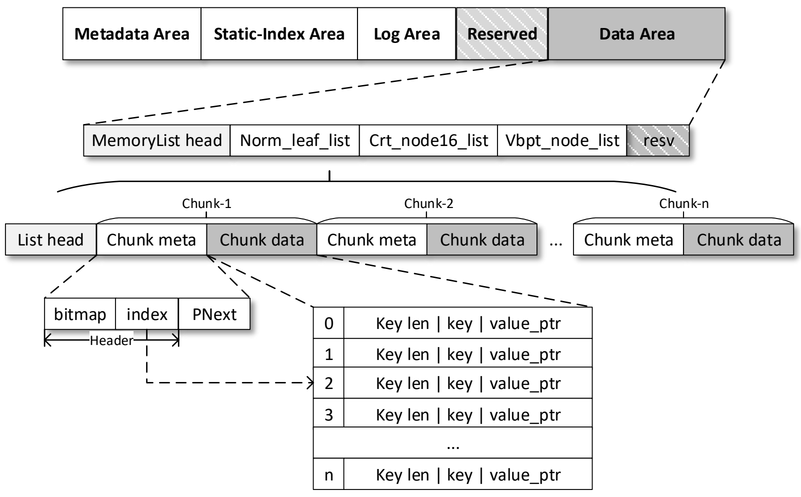


Figure 9. Space management of NVM.

Read. HybridKV processes read requests in the same way as LevelDB. The read thread searches with the priority of MemTable > Immutable MemTable > HybridTree. It uses pre-order traversal to search the tree, and uses binary search inside the tree node.

Write. (1) write requests are inserted into a write-ahead log on NVM to prevent data loss from system failures. (2) Data batched in DRAM, forming MemTable and immutable MemTable. (3) The immutable MemTable is flushed to NVM and the data is stored in HybridTree.

5. Evaluation

5.1. Evaluation Setup

All experiments are run a test machine with an Intel(R) 2.30GHz 18-core processors and 32GB of memory. The kernel version is 64-bit Linux 4.13.9 and the operating systems in use is Fedora 27. The experiments use two storage devices, an 800GB Intel SSDSC2BB800G7 SSD and 512GB NVMs of four 128GB Intel Optane DC PMM [19]. Table 1 lists their maximum single-thread bandwidth, evaluated with the versatile storage benchmark tool FIO.

Table 1. FIO 4MB read and write bandwidth.

	Optane DC PMM	SSDSC2BB800G7
Random Write	2392 MB/s	250 MB/s
Sequential Write	2417 MB/s	283MB/s
Random Read	3506 MB/s	291MB/s
Sequential Read	3509 MB/s	356 MB/s

We mainly compare HybridKV with LevelDB, RocksDB and NoveLSM. Considering that NoveLSM and LevelDB do not support multi-threaded optimization technology, the four systems all used single-threaded merge operation and single-threaded flush operation to ensure the fairness of the test. To verify the high performance of HybridTree on NVM, we used the same dataset to compare it with NV-Tree, Fast&Fair and WORT.

5.2. Performance of HybridTree

In this section, we first evaluate the read/write performance and space utilization of the four index structures using db_bench, the micro-benchmark released with LevelDB. Then, we evaluate the multi-thread scalability of HybridTree. For the first test, we set the key size of each key-value pair to 8B, and the value size ranges from 256B, 512B, 1KB to 4KB, and perform a single-threaded test under the limit of 200GB value space. We first random write 200M key-value pairs, and then randomly read and delete one million of them. For scan operations, the range size of 100, 1000, 10000, 100,000 is used for testing.

Write performance. Figure 10a shows the random write throughput of the four index structures as a function of value size. The performance of WORT based on Radix-Tree is the best, and the random write throughput is 1.7x~2.0x that of Fast&Fair and NV-Tree which are based on B+-Tree. This is mainly because the Radix-Tree uses the key prefix as an index to speeding up key search and avoids the balance overhead of the B+-Tree. NV-Tree's performance is better than Fast&Fair for its leaf nodes are not sorted, reducing the additional writes. HybridTree combines the characteristics of Radix-Tree and B+-Tree. The Radix-Tree-liked Static-Index and Dynamic-Index speed up key search, so the performance is better than Fast&Fair and NV-Tree, with 1.52x~1.62x throughput improvement when the value size is 1KB. While the B+-Tree-liked Variable-Tree introduces additional balance overhead, thereby the random write throughput of HybridTree is 0.7x~0.9x of WORT. The delete operations show similar results for the same reason, as shown in figure 10b.

Read performance. The random read performances of the four index structures are shown in figure 10c. Within the value size of 256B~4KB, WORT shows the best random read throughput, which is 1.3x~1.6x that of Fast&Fair and 1.6x~1.9x that of NV-Tree, since WORT uses key prefix to index, reducing the number of levels to search. Fast&Fair guarantees that the data in internal nodes are in order, so its throughput is better than NV-Tree that uses non-sorted nodes. Due to the full use of the advantages of prefix index, the random read throughput of HybridTree is close to WORT, which is 0.95x~0.97x that of it. Specifically, it reduces the height of the tree through the Static-index and retains the Radix-Tree-liked Dynamic-Index to speed up key search.

Figure 10d shows the scan performance of the four indexes as the function of range size when the value size is 256B. We use the results of NV-Tree as the baseline, and the ordinate represents the ratio of scan performance of other index structures to it, and the data label on the bar represents the actual IOPS. Under different range size, Fast&Fair shows the best performance while the WORT and NV-Tree have the worst performance. This is mainly because the internal data of Fast&Fair is ordered and the leaf nodes are connected by pointers, so the subsequent keys can be quickly read after the initial key is located, but the data in the nodes of NV-Tree and WORT are not sorted. Since the Variable-Tree of HybridTree is scattered and the leaf nodes are not completely linked together, the scan performance is weaker than Fast&Fair.

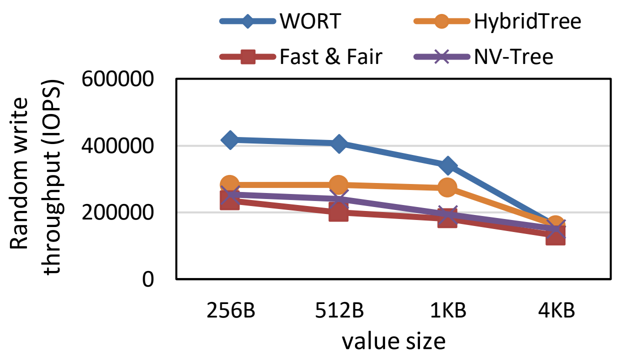
Space utilization. To compare the space utilization of the four index structures, we randomly write about 200GB of data with 8B-256B key-value pairs and count the space occupied by each index structure after all the data are written. As shown in figure 11, NV-Tree and WORT occupy much more space than the other index structures. NV-Tree occupies the most space because it pre-allocates contiguous memory to store the parent nodes of intermediate nodes and leaf nodes. The space occupied by WORT is close to 2.19x that of HybridTree and 2.33x that of Fast&Fair, since the WORT waste a lot of space when the keys are out of order.

Multi-thread scalability. To verify the scalability of HybridTree under concurrent access, 200 million 8B-1KB key-value pairs were written, and then 50 million of them were read and deleted. figure 12 shows the random read, random write and random delete performance of HybridTree as the function of thread counts. As the number of concurrent threads increases from 2 to 8, the performance of HybridTree increases proportionally. In addition, the performance of read operations improves more than the other operations since the concurrent threads of read operations do not block each other. In summary, HybridTree has good scalability under concurrent access.

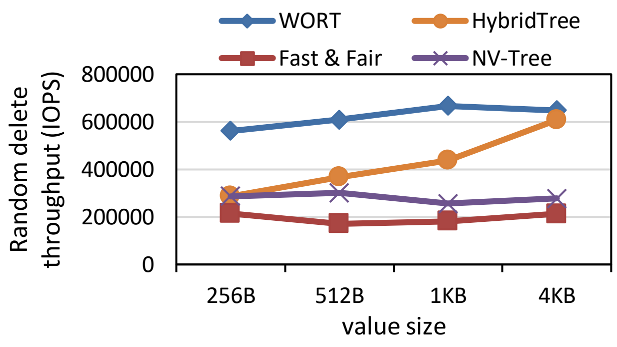
5.3. Performance of HybridKV

In this section, we use YCSB macro-benchmark [20] to test the performance of HybridKV on NVM, and compare it with LevelDB-NVM, RocksDB-NVM and NoveLSM. Table 2 shows the YCSB workload used for testing. During the test, 80GB of data with the value size of 4KB is written to the system, and the total number of operations is one million. Due to the large gap in the throughput of each system under different workloads, to better display the performance difference, RocksDB-NVM is used as a baseline, and the ordinate represents the ratio of the performance of other systems to it, as shown in figure 13. The data label on the bar represents the actual IOPS.

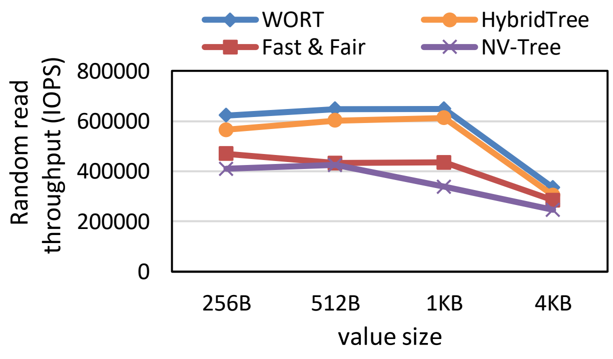
From the test results, we can see that the performance of HybridKV is generally better than the other three systems in various test workloads. In the process of data loading, HybridKV's performance is 3.22x that of RocksDB-NVM, 6.1x that of NovelLSM and 24x that of LevelDB-NVM, which shows that HybridKV has excellent writing performance on skew distributed data sets. For the write-intensive workloads (A, F), HybridKV makes full use of the in-place update feature of NVM, thus has great write performance, which is around 2x~3x that of RocksDB-NVM, 3x~6x that of NoveLSM, and 7x~9x that of LevelDB-NVM. Due to eliminate the read amplification of KVs based on LSM-Tree and use prefix index to accelerate search, the advantages of HybridKV are also obvious under read-intensive workloads. For the workloads (B, C, D) with high proportion of single-point read operations, it outperforms RocksDB-NVM, NoveLSM, and LevelDB-nvm by 4x~5x, 5x~7x and 7x~11x respectively. For the range query workload (E), the performance of HybridKV exceeds other systems by more than 10x.



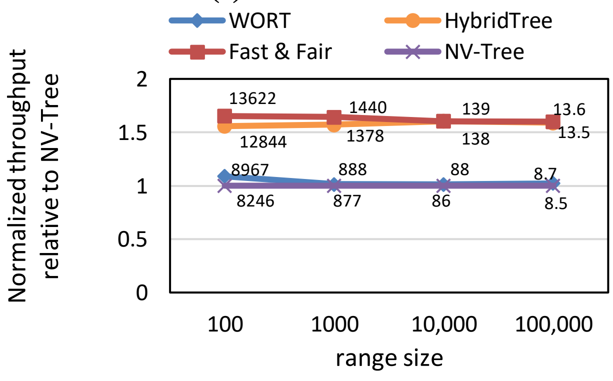
(a) Random Write



(b) Random delete



(c) Random read



(d) Scan

Figure 10. Random write, delete, read and scan performance of the four index structures.

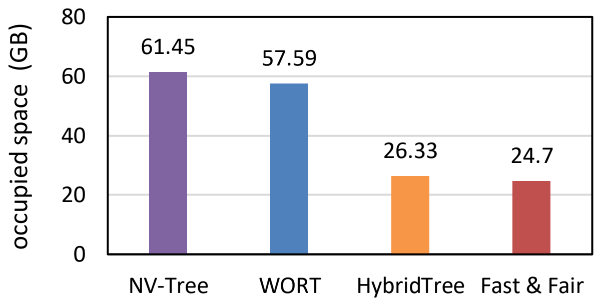


Figure 11. Space occupied by the four index structures. The space on each bar shows the amount of space occupied by the index structures.

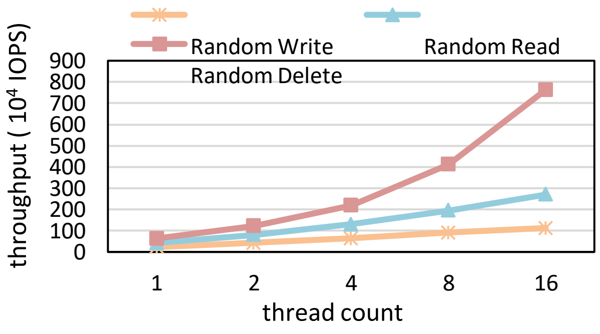


Figure 12. Performance of HybridTree under different thread counts.

Table 2. YCSB workload.

Workload	Description
A	50% read 50% update
B	95% read 5% update
C	100% read
D	95% read 5% insert
E	95% scan 5% insert
F	50% read 50% read-modify-write

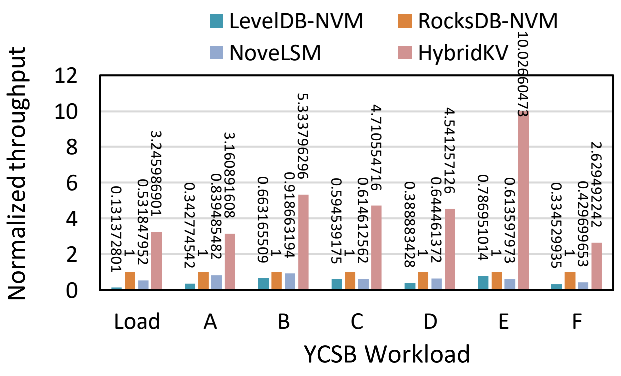


Figure 13. Micro-benchmarks of HybridKV. The y-axis shows the throughtput of each KV store normalized to RocksDB-NVM. The number of each bar indicates the throughtput in ops/s.

6. Conclusion

In this paper, we present HybridKV, a high performance KVs on NVM. HybridKV is based on a novel index structure HybridTree. By combining the advantages of Radix-Tree and B+-Tree, HybridTree can effectively save index space while providing excellent read/write performance. Through multi-threaded flushing for MemTable and separation of read and write locks, the advantages of high concurrency of NVM devices can be fully utilized. HybridKV is implemented on a real system based on LevelDB. Evaluation results demonstrate that HybridKV can achieve much better performance than LevelDB, RocksDB and NoveLSM on NVM.

Acknowledgments
This work was sponsored in part by the National Key Research and Development Program of China No.2018YFB1003305, the Shenzhen basic research project No. JCYJ20190809095001781, and the National Natural Science Foundation of China under Grant No.62072196.

References

[1] Lepers B, Balmau O, Gupta K and Zwaenepoel W 2019 Kvell: The design and implementation of a fast persistent key-value store *Proceedings of the 27th ACM Symposium on Operating Systems Principles* pp 447-461.
[2] O'Neil P, Cheng E, Gawlick D, et al. 1996 The log-structured merge-tree (LSM-Tree) *Acta Informatica* **33** (4) 351-385.
[3] Rocksdb: A persistent key-value store for fast storage environments 2019 http://rocksdb.org/.
[4] Ghemawat S and Dean J Leveldb 2016 https: //github.com/google/leveldb.
[5] Harter T, Borthakur D, Dong S, Aiyer A, Tang L, Arpaci-Dusseau A C and Arpaci-Dusseau R H 2014 Analysis of under HBase: a Facebook messages case study *12th Conference on File and Storage Technologies* pp 199-212.
[6] Lakshman A and Malik P 2010 Cassandra: a decentralized structured storage system *ACM SIGOPS Operating Systems Review* **44** (2) 35-40.
[7] Raoux S, Burr G W, Breitwisch M J, Rettner C T, Chen Y C, Shelby R M, et al. 2008 Phase-change random access memory: A scalable technology *IBM Journal of Research and Development* **52** (4.5) 465-479.
[8] Mikolajick T, Dehm C, Hartner W, Kasko I, Kastner M J, Nagel N, et al. 2001 FeRAM technology for high density applications *Microelectronics Reliability* **41** (7) 947-950.
[9] Lam C H 2010 Storage class memory *2010 10th IEEE International Conference on Solid-State and Integrated Circuit Technology* pp 1080-1083.
[10] Zhuang W W, Pan W, Ulrich B D, Lee J J, Stecker L, Burmaster A, et al. 2002 Novel colossal magnetoresistive thin film nonvolatile resistance random access memory (RRAM) *Digest. International Electron Devices Meeting* pp 193-196.
[11] Strukov D B, Snider G S, Stewart D R and Williams R S 2008 The missing memristor found *Nature* **453** (7191) 80-83.
[12] Kannan S, Bhat N, Gavrilovska A, Arpaci-Dusseau A and Arpaci-Dusseau R 2018 Redesigning LSMs for nonvolatile memory with NoveLSM *2018 Annual Technical Conference* pp 993-1005.
[13] Kaiyrakhmet O, Lee S, Nam B, Noh S H and Choi Y R 2019 SLM-DB: single-level key-value store with persistent memory *17th Conference on File and Storage Technologies* pp 191-205.
[14] Xia F, Jiang D, Xiong J and Sun N 2017 HiKV: A hybrid index key-value store for DRAM-NVM memory systems *2017 Annual Technical Conference* pp 349-362.
[15] Yang J, Wei Q, Chen C, Wang C, Yong K L and He B 2015 NV-Tree: Reducing consistency cost for NVM-based single level systems *13th Conference on File and Storage Technologies* pp 167-181.

[16] Hwang D, Kim W H, Won Y and Nam B 2018 Endurable transient inconsistency in byte-addressable persistent b+-tree *16th Conference on File and Storage Technologies* pp 187-200.
[17] Lee S K, Lim K H, Song H, Nam B and Noh S H 2017 WORT: Write optimal radix tree for persistent memory storage systems *15th Conference on File and Storage Technologies* pp 257-270.
[18] 2019 Persistent memory development kit https:// github.com/pmem/pmdk.
[19] Izraelevitz J, Yang J, Zhang L, Kim J, Liu X, Memaripour A, et al. 2019 Basic performance measurements of the intel optane DC persistent memory module *arXiv preprint arXiv:1903.05714.*
[20] Cooper B F, Silberstein A, Tam E, Ramakrishnan R and Sears R 2010 Benchmarking cloud serving systems with YCSB *Proceedings of the 1st ACM symposium on Cloud Computing* pp 143-154.