LightKV: A Cross Media Key Value Store with Persistent Memory to Cut Long Tail Latency

Shukai Han, Dejun Jiang, Jin Xiong
SKL Computer Architecture, ICT, CAS; University of Chinese Academy of Sciences
{hanshukai, jiangdejun, xiongjin}@ict.ac.cn

***Abstract*—Conventional persistent key-value stores widely adopt LSM-Tree to manage data across memory and disk. However, expensive write ahead logging, inefficient cross-media indexing and write amplification are three limitations faced by LSM-Tree based key-value store. Thanks to the development of non-volatile memory (NVM), persistent key-value stores can exploit NVM-based persistent memory (PM) to directly persist data avoiding costly logging. With this design choice, we propose LightKV a cross media key-value store with persistent memory. To support efficient cross-media indexing, we design a global index Radix-Hash Tree (RH-Tree) consisting of the upper-layer radix tree and hash table based leaf nodes. We explore the specific features of real PM product to balance the persistency and performance of RH-Tree. Meanwhile, replying on the range partition of RH-Tree, LightKV organizes key-value pairs with the same key prefix into SSTables within the same partition. LightKV then conducts partition-based data compaction with carefully-controlled compacted data volumes. By doing so, LightKV greatly reduces write amplification. We evaluate LightKV against state-of-the-art PM-based key-value stores NoveLSM and SLM-DB as well as LevelDB and RocksDB. The experiment results show that LightKV reduces write amplification by up to 8.1x and improves read performance by up to 9.2x. Due to the reduced write amplification, LightKV also reduces read tail latency by up to 18.8x under read-write mixed workload.**

***Index Terms*—key-value store, persistent memory, tail latency, log-structured merge tree**

I. Introduction

Persistent key-value (KV) stores have become an important part of storage infrastructure in data centers. They are widely used in various web services such as search engines [8], [24], social networking [14], e-commerce platform [11], and more. Moreover, some file systems [21], [35] and relational databases [12] use KV stores as either metadata store or storage engine. KV stores usually serve data with small granularity, such as from tens or hundreds of bytes to a few kilo bytes. Thus, KV stores are latency-critical applications. Nowadays, tail latencies, such as 99^{th} and 99.9^{th} percentile latencies, are especially important to guarantee quality of services [11], [37].

Log-Structured Merge Tree (LSM-Tree) [32] is a fundamental component for existing persistent KV stores, such as BigTable [8], LevelDB [17], RocksDB [14] and Cassandra [25]. LSM-Tree maintains an in-memory write buffer (namely MemTable in LevelDB and RocksDB) to receive incoming writes. Any insert/update/delete operation is written into this buffer in an append-only way without in-place updating. When the write buffer is full, LSM-Tree flushes KV pairs in the buffer into underlying disks in a batch to achieve high write performance. However, LSM-Tree requires to conduct data compaction to reclaim KV pairs.

LSM-Tree based KV stores face challenges when providing low tail latency. We observe that the 99^{th} and 99.9^{th} percentile read latencies reach up to 13x and 28x compared to average read latency for LevelDB under read-write mix workloads. The tail latency is affected by the inefficient cross-media indexing and level-based data compaction of existing LSM-Tree based KV stores. On the one hand, existing KV stores adopt skiplist and embedded index block to separately index in-memory KV pairs and on-disk KV pairs. This brings slow data read. For example, when searching a KV pair, the embedded index block is first read into memory to locate the target key. Then, the data block including the target KV pair is read into memory. On the other hand, data read latency further increases when conducting level-based data compaction. Currently, LSM-Tree logically organizes data into multiple levels. The number of SSTables at an upper level is usually 10X than that at a lower level [30]. When compacting SSTable(s) from lower level to upper level, the compaction process usually involves all SSTables in the upper level. This brings significant write amplification. The extra data writes in background further affect the front-end data reads, and results in heavy tailed read latency.

Recently, SILK [6] is proposed to adopt an I/O scheduler to carefully conduct resource allocation and preemption-based scheduling between client and internal operations (flushing and data compaction). In such doing, SILK can prevent latency spikes. Thanks to the development of byte-addressable and fast non-volatile memories (NVMs), such as PCM [39], ReRAM [5], and 3D XPoint [1], persistent memory (PM) is proposed by placing NVM in memory bus to serve persistent data directly. Thus, in this paper, we explore the tail latency issue from an different perspective by revisiting data index and data organization and meanwhile exploiting PM.

A few recent works propose to adopt PM to optimize LSM-Tree. For example, both NoveLSM [23] and SLM-DB [22] build persistent MemTable based on PM to reduce the costs of write ahead log and (de-)serialization of KV pairs. Similar to previous works [13], [22], [23], we adopt PM in KV stores to serve as persistent write buffer, which can avoid write log overhead and reduce write latency in critical paths. Moreover, we take the following two insights for building persistent KV stores. First, a global index structure is required to efficiently

locate cross-media data. By building an efficient global index that both manages KV pairs in PM and SSD at the same time, we can effectively reduce read amplification, thereby reducing device IO, improving read performance, and reducing tail read latency.

Secondly, one can reduce tail latency by carefully organizing KV items and controlling the key ranges involved in a data compaction. For example, assuming the key size is 1 byte and there exist three SSTables with key ranges of (a, g), (h, n), and (o, q) separately. Note that, since SSTable is sorted, the key range of an SSTable is represented by its starting key and ending key within a parentheses. In case of compacting an SSTable having the key range of (c, p) with these three SSTables, all the three SSTables need to be read and compacted. Their valid KV pairs are then re-written to disks again, which results in write amplification. If one can control the key range of an SSTable, the number of SSTables required by a data compaction can be reduced. For example, we partition the key range of (a, q) into three partitions that are (a, g), (h, n), and (o, q) respectively. One decides the belonging partition of a KV pair according to its key. KV pairs belonging to the same partition are first buffered together and then written into SSTables. Thus, the key range of a compacting SSTable only overlaps with the SSTables within the same partition. When conducting compaction, one can compact SSTables belonging to the same partition. This greatly reduces the number of SSTables involved in the compaction.

In this paper, we first propose Radix Hash Tree (RH-Tree), an index consisting of the upper-layer Radix tree and hash table based leaf nodes. RH-Tree acts as the global index to locate data across persistent memory and disk. Compared to separate cross-media indexes, the global RH-Tree allows to quickly locate KV pairs either in PM or disk. Moreover, the upper-layer radix tree stores keys according to their prefixes. Each branch of the upper-layer Radix tree clusters KV pairs with the same key prefix into a partition. As a result, the KV pairs indexed by a leaf node belong to the same partiton. The leaf node exploits the fast accessing of hash table to support efficient single-point operations. We then propose partition-based data compaction for reclaiming invalidate KV pairs on underlying disks. We conduct compaction in a per-partition way. This allows us to carefully control the number of SSTables involved in compaction and thus reduce write amplification.

We finally build LightKV, a cross media KV store adopting RH-Tree to index cross-media data and partition-based compaction to reduce tail latency. We conduct extensive experiments to evaluate LightKV against state-of-the-art KV stores NoveLSM and SLM-DB as well as the widely used LevelDB and RocksDB. The experiment results show that LigthKV outperforms these KV stores by 33.0x and 9.2x for random writes and random reads separately. Moreover, LightKV reduces write amplification by up to 8.1x and reduce read tail latency by up to 18.8x.

In summary, this paper makes the following contributions:

- We propose Radix Hash Tree (RH-Tree), a cross-media index based on DRAM and PM, which can effectively manage data both in PM and SSD.
- We propose the partition-based compaction, which organizes KV pairs with the same prefix in one partition and performs well-controlled compaction to reduce write amplification.
- We implement LightKV based on RH-Tree and partition-based compaction. We conduct extensive experiments to confirm the efficiency of LightKV.

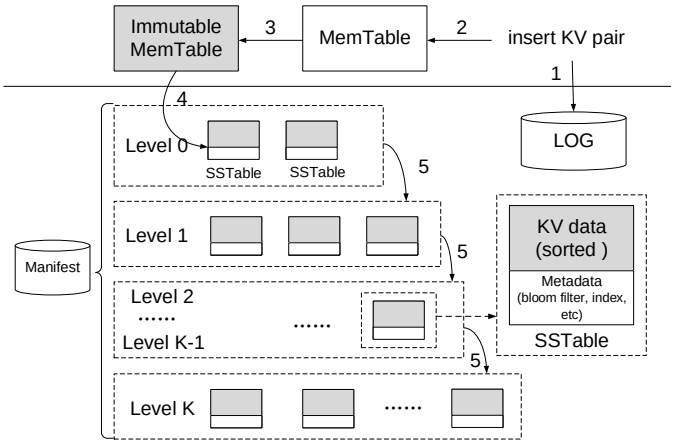


Fig. 1: **LevelDB architecture,** *(1) Writing Ahead Log; (2) Inserting key-value pair into MemTable; (3) Converting full MemTable to Immutable; (4) Immutable MemTable is flushed to disk to become a SSTable file; (5) Compaction between* $Level_i$ *to* $Level_{i+1}$.

II. Background and Motivation

A. Log-Structured Merge Tree

Log-Structured Merge Tree (LSM-Tree) is the fundamental organization for persistent KV stores. As for write-intensive workloads, LSM-Tree merges random write into sequential write for disk. We use the widely used LevelDB [17] to explain the architecture of LSM-Tree. Figure 1 shows the architecture of LevelDB. It maintains a write buffer (MemTable) in DRAM. The MemTable is indexed using skiplist. To handle a write request, the KV pair is first written to the log on disk. Then, it is written to the MemTable. When the MemTable is full, a new MemTable is created. The full MemTable is modified to an Immutable MemTable and flushed to disk in the form of SSTable file by a background thread. The manifest file is used to record information of the KV store metadata.

The on-disk SSTables are organized into multi-levels. Except for Level 0, the key ranges of SSTables within the same level do not overlap. Unfortunately, the key ranges of SSTables from different levels may overlap. The maximum number of SSTable files on each level is fixed. When the number of SSTables on the low level reaches certain threshold, the background threads merge the SSTables from $level_i$ to $level_{i+1}$. This process is called **compaction**. Since LevelDB uses append write to delete or update KV pairs, compaction also recycles invalid KV pairs.

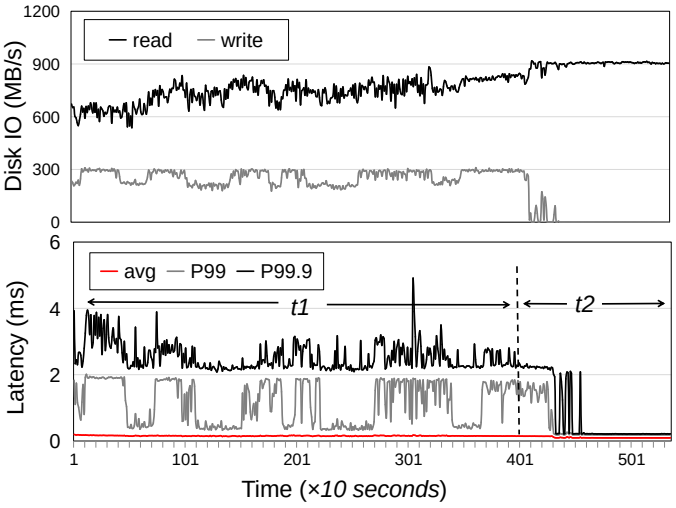


Fig. 2: **Read tail latency results and disk read/write bandwidth in real-time.** *This figure shows the average, 99^{th} and 99.9^{th} percentile read latencies and disk read/write rate for LevelDB in different workloads. We run a read-write mixed workload of random read and random write during the t1 time period, and run random read workload during the t2 time period.*

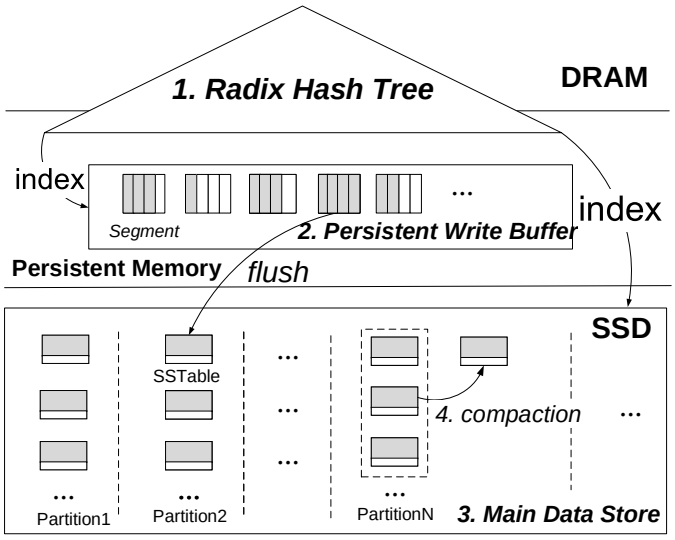


Fig. 3: **LightKV architecture.** *This figure shows the LightKV architecture. (1) Radix Hash Tree (RH-Tree) is a global index across DRAM-PM, which is used to index all KV items in PM and SSD. (2) The Persistent Write Buffer (PWB) is used to receive temporarily written data and transfer it to the SSD in segments. (3) Data in SSD is organized and partitioned in the manner of Sort String Table (SSTable). (4) These SSTables use partition-based data compaction to handle garbage collection and improve indexing efficiency*

B. Limitations of persistent KV store

Inefficient indexing for cross-media data. Persistent KV store needs to manage data cross memory and underlying disk. On one hand, LSM-Tree adopts skiplist to index in-memory data. On the other hand, LSM-Tree builds manifest files to record key range of each on-disk SSTable. Each SSTable contains an index block such that one can apply binary search to locate keys. For serving single-point query, LSM-Tree requires at least two disk IOs to fetch index block and data block separately[1].

Heavy tailed read latency under mixed workload. In order to provide high write throughput, LSM-Tree adopts append write for both new inserts, updates and deletes. This results in background garbage collection. LSM-Tree logically organizes on-disk SSTables into multiple levels. However, LSM-Tree does not control the key range of an SSTable. If the key range of a compacting SSTable at the lower level (e.g. level 0) overlaps with all SSTables at the upper level (e.g. level 1), all SSTables from the upper level are read and re-written for compaction purpose. Since the number of SSTables at an upper level (e.g. level 1) is usually 10X than that at a lower level (e.g. level 0). This results in significant write amplification. Previous studies show that the write amplification of LSM-Tree with K levels exceeds 10*K [30].

Since SSD has asymmetric read and write latency, the critical reads are affected by long writes when serving read-write mixed workloads. This is exacerbated when background data compaction is conducted. We conduct experiments to show the heavy tailed read latency. We first warm up LevelDB with 100 GB data. Then, we run a mixed workload of randomly reading 50 GB existing data and randomly inserting another 50 GB data. We measure the average latency as well as 99^{th} and 99.9^{th} percentile read latencies every 10 seconds. Figure 2 shows the read tail latency results and disk read/write bandwidth. During *t1* time period, when the disk write bandwidth increases (e.g. LevelDB executes background compaction), the 99^{th} and 99.9^{th} percentile read latencies correspondingly increase. The maximum 99^{th} and 99.9^{th} percentile read latencies can reach 13 and 28 times than the average read latency. Starting from *t2* time period, the mixed workload finishes, and we run read-only workload. However, at the beginning of *t2* period, the heavy tailed read latency is still observed as previous compactions do not end. After the compaction finishes, the read tail latency is significantly reduced. Thus, the disk write spikes caused by data compaction is a key factor increasing read tail latency. Similar to the key range overlapping issue, this also motivates us to reduce write amplification during data compaction.

[1]Note that, manifest files are usually cached in memory.

C. Non-Volatile Memory

Non-Volatile Memories (NVMs), such as 3D XPoint [1], Phase Change Memory (PCM) [39], and Resistive Memory (ReRAM) [5], provide low latency and byte addressable features. Especially, NVM can persist data after power off. Thus, NVM can reside on memory bus to serve as persistent memory (PM). Recently, a few works adopt PM to reduce the cost of write ahead log as well as data (de-)serialiation [22], [23].

The first PM product, Intel Optane DC Persistent Memory (PM), was announced [19] in April 2019. We measure the performance of Optane DC PM, and observe the following features: the write latency of Optane DC PM is close to DRAM, while its read latency is 3 to 4 times that of DRAM. The write and read bandwidths of Optane DC PM are around 2GB/s and 6.5GB/s, which is about 1/8 and 1/4 that of DRAM separately. These observed features are similar as reported in [20], [29]. The specific performance features of Optane DC PM is different from the previous assumptions about NVM performance. NVMs were expected to have read latency similar to DRAM and longer write latency than DRAM. Thus, this motivates us to explore the usage of PM when using its persistency for data and index durability and meanwhile taking its specific latencies into account. For example, when locating KV pairs stored in PM, the index is desirable to be placed in DRAM instead of PM itself to achieve fast querying.

III. LightKV Design

A. System overview

In this section, we present the design of LightKV. Figure 3 shows the system architecture of LightKV. It mainly consists of three parts: a global index Radix Hash Tree (RH-Tree) indexing cross-media data, NVM-based persistent write buffer (PWB), and SSD-based main data store.

RH-Tree is a tree-like structure consisting of Radix tree with hash table based leaf nodes. The PWB consists of multiple segments, each of which is indexed by a leaf node of RH-Tree. KV pairs with different prefix paths are written into different segments in an append-only way. According to PM specific performance feature, RH-Tree is placed across DRAM and PM, with parts of leaf nodes placed in PM. When a leaf node is fulfilled, the KV pairs in its indexing segment are flushed into SSD in the form of String Sorted Table (SSTable) [17]. After that, a new hash leaf node is generated to hold further incoming index entries. The hash leaf nodes with the same prefix paths are linked as a list.

Since KV pairs are written into SSD in an append-only way, LightKV also conducts compaction to collect invalid KV pairs. As shown in Figure 3, SSTables are logically organized into partitions, and KV pairs in the same partition have the same key prefix. LightKV conducts partition based data compaction, allowing data volume involved in each round of compaction to be carefully controlled. In such doing, LightKV greatly reduces write amplification caused by data compaction.

B. Radix Hash Tree Structure

Radix Hash Tree is the key component of LightKV to index both data and metadata. In conventional LSM-Tree based KV stores [14], [17], the MemTable is indexed using skiplist, and meanwhile the SSTables are indexed using index blocks and bloom filters. In such doing, when serving read requests, one needs to first look up the MemTable, and then search SSTables. This results in slow read performance. Previous works propose to use large persistent write buffer to reduce the costs of write-ahead log and (de-)serialization but still using skiplist to index

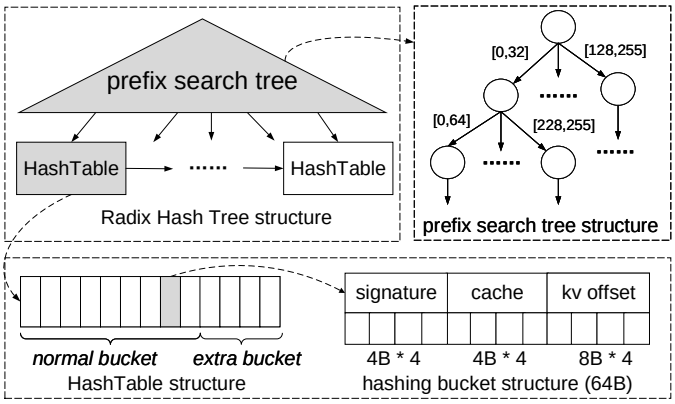


Fig. 4: **Radix Hash Tree architecture.** *Radix Hash Tree (RH-Tree) consists of Radix tree with hash table based leaf nodes. The index entries of the same prefix KV pairs are divided into the same leaf node. Due to the Radix tree structure, the leaf nodes are in order.*

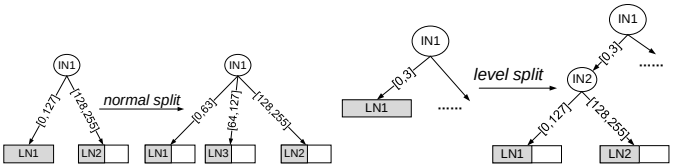
buffered data [23]. This suffers from degraded performance when the buffered data increase. This motivates us to design a global index to mange KV pairs in both PWB and SSD store.

In this paper, we propose Radix Hash Tree (RH-Tree), a tree-like structure combining Radix tree and hash table. The upper-layer of RH-Tree is Radix tree to execute prefix search for keys. Compared to B^+-tree, all operations on Radix tree have the complexity related to the key length but independent of the key numbers in a dataset. This feature is attractive when holding large volume of KV pairs. However, Radix tree with a large fanout suffers from large space consumption. To balance the search complexity and space utilization, we do not build a full Radix tree to index all KV pairs. Instead, we let each leaf node of RH-Tree use a hash table to store the keys with the same prefix. We name the leaf node in RH-Tree as *hash leaf node*.

As shown in Figure 4, each hash leaf node is composed of multiple buckets. A bucket has three types of slots, including 4 signature slots, 4 cache slots, and 4 offset slots. A signature slot contains a 32-bit (4B) signature referring to the hashed key of a KV pair[2]. A cache slot is used to cache 32-bit (4B) characters of the key. It is used for RH-Tree splitting without directly accessing the actual key, thereby avoiding additional IO overhead. An offset slot contains a 64-bit (8B) value indicating the offset of a KV pair within either the PWB or an SSTable file. Note that, RH-Tree only stores the addresses of KV pairs in offset slots. The KV items are actually stored in PWB in PM or main data store in SSD. Since the size of a bucket is 64B (one cache line size), RH-Tree can get a bucket with one memory access.

To address a KV pair in RH-Tree, one first finds the hash leaf node in RH-Tree using the key prefix. Then, the hash value of the target key is used to locate the bucket. Once the bucket is located, one needs to compare all signature slots in this

[2]Note that, since the KV pairs indexed by a hash leaf node have the same prefix, few hash collision occurs when using 32-bit signature.



(a) normal split operation (b) level split operation

Fig. 5: **Split operation in RH-Tree.** *In this figure, LN 1, LN 2 and LN 3 refers to leaf nodes 1, 2, and 3 respectively. IN 1, IN 2 refers to inner node 1, 2 respectively.*

bucket with the hash value. In case of finding a matching slot, one uses the address in the corresponding offset slot to obtain the KV pair. Since the signature slot only stores hash values of keys, a full-key comparison is required after obtaining the KV pair. RH-Tree returns the KV pair only after a successful full-key comparison.

The leaf nodes in RH-Tree handle hash collisions to increases the load factor in the two ways. First, RH-Tree adopts two hash functions to select the bucket. Secondly, RH-Tree adds an extra bucket within a hash leaf node as shown in Figure 4. When there are no empty slots in the bucket selected by the two hash functions, the key is sequentially stored in the extra bucket. RH-Tree applies linear search to locate keys stored in the extra bucket. Note that, the extra bucket only adds 5% extra space to a hash table. This allows RH-Tree to achieve high load factor but without losing too much performance. We observe the average load factor of leaf nodes in RH-Tree is 0.9 when the leaf node is unable to handle a conflict.

A leaf node becomes full when its normal bucket and extra is fulfilled. At this time, RH-Tree starts to grow through splitting (Section III-C) or generating linked leaf nodes (Section III-D).

C. RH-Tree split

Conventional Radix tree consists of two types of nodes: inner nodes and leaf nodes. An inner node is usually an array of 2^s pointers, which maps the *s* bits of the key. The 2^s pointers referring to 2^s child nodes. Given a key length *k*, Radix tree has $\lceil k/s \rceil$ levels. However, RH-Tree does not build full Radix tree. Thus, it does not have pre-defined levels. Initially, RH-Tree has only one inner node (root node)[3] and *N* leaf nodes. The inner node also maps the key prefix of *s* bits. We further divides the key prefix into *N* parts. We assign each part to a hash leaf node. For example, assuming the parameter *s* is 8 and *N* is 2. We use the digits 0 to 255 to represent the values of 8 bits key prefix. The two hash leaf nodes map key prefixes 0 to 127 and 128 to 255 separately as shown in Figure 5a. When the hash table of any leaf node (including both normal buckets and extra bucket) is full, RH-Tree split occurs. RH-Tree adopts two splitting strategies as follows: normal split and level split.

Normal split. When split occurs, RH-Tree prefers to first split the value range mapped by current leaf nodes. RH-Tree adds a new hash leaf node, and assigns a half of the values mapped by a current leaf node to the new one. This is called normal split in RH-Tree. For example, as shown in Figure 5a, hash leaf node 1 (LN1) initially maps values 0 to 127. When a normal split is executed, a new hash leaf node 3 (LN3) is added. This leaf node 3 is responsible for mapping values 64 to 127. Instead leaf node 1 only maps values 0 to 63. The index entries corresponding to values 64 to 127 stored in leaf node 1 are then moved to leaf node 3.

After a normal split, the index entries in a leaf node (e.g. LN1 in Figure 5a) are moved to another (e.g. LN3 in Figure 5a). Assuming the depth of the current leaf node is *D*, RH-Tree only needs to use the D^{th} character of a given key to decide whether the related entry needs to be mored or not. RH-Tree designs a 4 B cache slot to store 4 consecutive characters of a key. In such doing, the entry movement can be decided by comparing cache slot in CPU cache without accessing full key in PM. This helps to reduce splitting overhead.

Level split. We set a threshold *T_normal* for normal split to be the least number of values that are mapped by a hash leaf node. When executing split, RH-Tree checks what the number of values mapped by the new leaf node will be if normal split occurs. In case of being less than *T_normal*, RH-Tree turns to execute level split. Instead of adding new hash leaf node having the same prefix path with current ones, level split adds a new inner node to map next *s* bits of the key. For example, assuming *T_normal* is 4, inner node 1 (IN1) maps the first *s* bits (e.g. 8 bits) of the key, and leaf node 1 already maps values 0 to 3 as shown in Figure 5b. In such case, when leaf node 1 needs to split, RH-Tree executes level split by adding new inner node 2 (IN2). The inner node 2 maps next 8 bits of the key, and initially has 2 hash leaf nodes. The index entries stored in the leaf nodes with the first 8 bits key prefix are moved to the two new leaf nodes according to the values of next 8 bits of the key.

Similar to normal split, level split also exploits cache slot to avoid in-PM full key access when deciding index entry movement. However, since level split involves the growth of key prefix, RH-Tree renews the 4 B cache slot with another 4 consecutive characters after performing four level splits.

Slot reuse. When deleting KV pairs, RH-Tree simply marks the corresponding signature and offset slots as invalid. Instead of reclaiming the invalid slots in leaf nodes, these slots are reused to serve newly coming KV pairs.

D. RH-Tree leaf nodes

RH-Tree grows with increasing KV pairs accompanied with either normal split or level split. Both normal split and level split generates new leaf nodes. In order to balance operation performance and space consumption due to the increased leaf nodes, we define a threshold *MAX_LEAF_NODES* as the maximum number of leaf nodes in RH-Tree. Once the number of leaf nodes exceeds *MAX_LEAF_NODES*, both normal split and level split are stopped. Instead, we handle data growth by adding linked leaf nodes.

[3]Note that, the root node is considered as the first inner node.

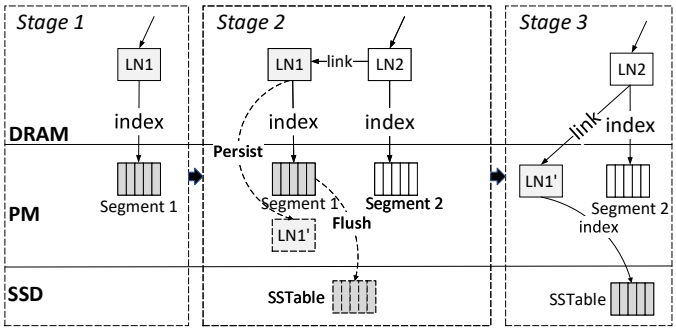


Fig. 6: **Linked hash leaf node.** *In this figure, LN 1, LN 2 and LN 1' refers to leaf nodes 1, 2, and 1' respectively.* When a leaf node is fulfilled, the data stored in a segment is converted to an SSTable which is flushed to SSD. At the same time, the corresponding leaf node is persisted to PWB.

Figure 6 shows the generation process of linked hash leaf nodes. In stage 1, leaf node 1 (LN1) indexes KV pairs stored in the PWB segment 1. When leaf node 1 becomes full and meanwhile the total number of leaf nodes exceeds *MAX_LEAF_NODES*, we generate a new hash leaf node 2 (LN2) to serve new indexes as shown in stage 2. Meanwhile, a new empty segment 2 is used to store new KV pairs indexed by leaf node 2. Leaf node 1 is linked behind leaf node 2 for serving KV pairs in segment 1. Then, we apply copy-on-write to leaf node 1 to generate a persistent copy of leaf node 1 in PWB (leaf node 1'). The KV pairs in segment 1 are sorted to generate an Sort String Table (SSTable), and the SSTable is asynchronously flushed into underlying SSD. During the flushing process, the indexes in leaf node 1 are correspondingly modified to refer to the KV location in SSD. Note that, leaf node 1 still indexes KV pairs in segment 1 for serving incoming requests. Finally, after the SSTable is flushed, leaf node 2 links to leaf node 1' instead of leaf node 1. The original leaf node 1 is deleted, and leaf node 1' acts as a new persistent leaf node referring to KV pairs in SSTable file.

The above process is repeated again when leaf node 2 is fulfilled. The number of linked leaf nodes under each prefix path is controlled to avoid unnecessary data compaction, which is illustrate in Section III-F

E. RH-Tree placement

So far, RH-Tree is able to index data in both PWB and SSD. Previous works usually place indexes in PM [22] to achieve persistency. However, according to the performance features of PM product in latest evaluation [20], [29], the read latency of Optane PM is 3 to 4 times that of DRAM. Thus, placing the whole RH-Tree index in PM affects the query performance, especially for locating KV pairs in PWB. Moreover, placing whole index in PM requires costly persistency operations (e.g. guaranteeing crash consistency for RH-Tree splits). On the contrary, placing the whole RH-Tree in DRAM is able to

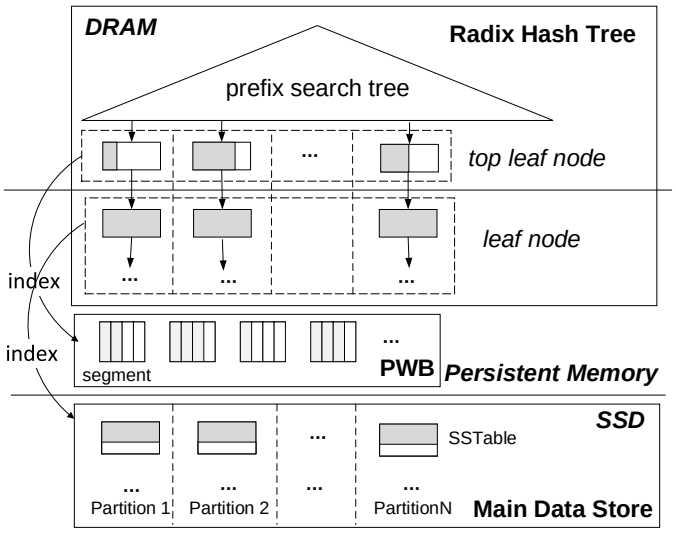


Fig. 7: **The index method of LightKV.** *This figure shows the index placement of LightKV. The top leaf node is the topmost index node located in the RH-Tree leaf node which is used to index the data in PWB. The remaining leaf nodes are the data of the SSTable in SSD. Leaf nodes except top leaf node are persisted in NVM.*

achieve high performance but losing the persistency. Once KV store is restarted, the whole index requires costly re-building.

To balance the query performance and index persistency, we design a cross-media layout for RH-Tree as shown in Figure 7. We place the prefix search tree as well as the top leaf node (the latest one) within each linked leaf node list in DRAM. As illustrated in Section III-D, a top leaf node always indexes KV pairs in PWB. In such doing, RH-Tree provides fast prefix search due to the lower DRAM latency and quickly locates data in PWB. The rest leaf nodes within a linked leaf node are placed in PM. These leaf nodes index KV pairs in SSD. Since the read latency of SSD is at least two orders of magnitude slower than that of PM, accessing KV pairs in SSD by using indexes in PM is acceptable.

Note that, the prefix tree and top leaf nodes are not persisted. One needs to scan all the segments in PWB to rebuild this part of RH-Tree. We illustrate the recovery issue in Section V.

F. Partition-based data compaction

Conventional LSM-Tree conducts data compaction when an lower level is fulfilled with SSTables. Moreover, since the key range involved in a compaction may spread over all SSTables within the upper level, this usually results in large number of data read and re-write. This further results in significant write amplification. Thanks to the prefix radix tree of RH-Tree, it helps to guarantee KV pairs indexed under a hash leaf node (namely partition) have same key prefix. This allows LightKV to conduct data compaction in a per-partition way. Thus, we propose partition-based data compaction to well control the involved data volume. We define two parameters *compaction size (CS)* and *compaction weight (CW)* to control

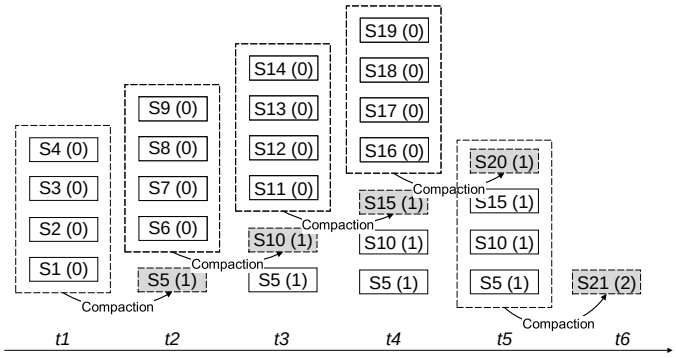


Fig. 8: **A example of compaction in one partition.** *This figure shows the compaction process of flushing 16 SSTables within a partition. At time t1, t2, t3, and t4, four SSTables are flushed into SSD respectively.*

partition-based data compaction. Compaction size refers to the number of SSTables within a partition that execute compaction each time. Compaction weight instead refers to the number of compactions incurred by KV pairs in an SSTable. For each compaction, LightKV only compacts *CS* SSTables with the same compaction weight.

We take an example shown in Figure 8 to illustrate the process of partition-based compaction. We assume the compaction size is 4. Initially, the compaction weight of a new SSTable is 0, indicating it has not been compacted. The SSTables S1 to S4 belong to the same partition in RH-Tree. Once they are all fulfilled at time *t1*, we compact them together by recycling invalid KV pairs and sort them to generate a larger SSTable S5. The KV pairs in S5 incur once compaction, and thus the compaction weight of S5 is 1. At time *t2*, another four new SSTables S6 to S9 are fulfilled. They all have a compaction weight of 0. Thus, at time *t2* another compaction occurs by merging them to the larger SSTable S10. Note that, when SSTables S6 to S8 are fulfilled, they are not compacted with S5 as their compaction weights (*CW* equals to 0) from S5 (*CW* equals to 1). Only when another new SSTable S9 with the same compaction weight is fulfilled, these four SSTables begin to compact. This process repeats to recycle invalid KV items and sort remaining ones. At time *t5*, there exist four SSTables S5, S10, S15 and S20 with the same compaction weight of 1. Then, they are compacted together to generate S21 with the increased compaction weight of 2.

The partition-based data compaction greatly reduces the write amplification by controlling the compaction size. Assuming the compaction size is S and the total number of newly flushed SSTables (excluding SSTables generated by compaction) is N, the write amplification of LightKV is $\log_S N + 1$. For example, at time *t6* in Figure 8, the newly flushed SSTables are 16, the KV pairs in these SSTables are compacted twice. One is for generating SSTable with compaction weight 1 and the other is for generating SSTable with compaction weight 2. Taking the initial SSTable flushing into account, the write amplification is 3. Note that, increasing compaction size can reduce write amplification but at the cost of decreasing searching efficiency.

G. Discussion

Although RH-Tree is able to cluster KV pairs into partitions according to their key prefixes, it faces the data skew issue. Unbalanced key distributions result in a few overloaded partitions with a large number of SSTables. This degrades query performance. Several optimizations can be used to reduce the impact of data skew.

For example, one can add random hash values at the beginning of keys to scatter key distributions but at the cost of reduced scan performance. Alternatively, one can set a load factor to indicate the maximum keys stored in one partition. Once the number of keys of a partition exceeds this load factor, we split the partition. In this paper, LigthKV performs well under the default skewed workloads of YCSB. as shown in Section VI-E. We leave the optimizations for very skewed data workloads for the future work.

IV. LightKV operations

Insert: When serving an insert request, LightKV first searches for an empty entry in leaf node of RH-Tree. If no empty entry is found, LightKV conducts RH-Tree split according to current number of leaf nodes. If the number of leaf nodes is greater than *MAX_LEAF_NODES*, LightKV adds linked hash leaf node. After that, LightKV appends KV pair to the segment indexed by either the newly split leaf node or linked leaf node. Finally, LightKV updates RH-Tree index.

Update/Delete: Updates and deletes in LightKV are similar to insertion as LightKV adopts append write. The difference is that update operation only updates the offset slot in index entry, while delete operation only updates the signature slot in index entry by marking it to be invalid.

Get: When searching for a key, LightKV searches the prefix tree of RH-Tree through the prefix of the key to locate a certain partition. Then, it searches all hash leaf nodes in the partition from front to back until it finds the corresponding slot (the signature value of the slot is equal to the 32-bit hash value of the searched key). Through the address slot, it finds the KV item from the PWB or SSTable. Finally, it performs a full key comparison, and returns the result if they are equal.

Range query: To serve range query, LightKV first uses the upper-layer radix tree to locate the prefixes of starting and ending keys. Correspondingly, LightKV locates the starting and ending partitions. As for KV pairs stored in PWB, LightKV needs to fully scan the hash leaf nodes. As for KV pairs stored in SSD, each SSTable is sorted and contains index block. After locating the starting and ending partitions, LightKV reads index blocks of the SSTables within the partition. In such doing, LightKV is able to quickly find corresponding KV pairs. Moreover, LightKV adopts multiple threads to scan different SSTables in parallel. The scanned results are then merged and returned. This helps to improve range query performance.

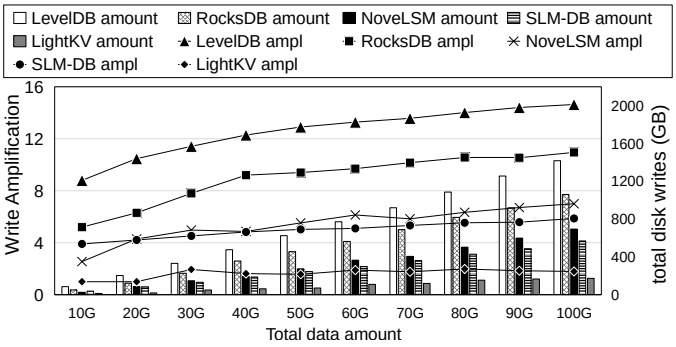


Fig. 9: **Write Amplification.** *This figure shows the write amplifications and write amounts for different KV stores with different data amounts.*

V. Recovery

At runtime, LightKV persists a leaf node to PM when its corresponding segment is flushed into SSD. Similar to the persistent memory management in PMDK [4], LightKV itself maintains a persistent root structure referring to the physical offset addresses of leaf nodes (except top leaf nodes) and PWB in PM. The root data structure is stored in a fixed location in PM. In case of system recovery after crash, these leaf nodes can be used directly. On the contrary, the prefix search tree and the top leaf nodes are maintained in DRAM, which are lost after system crash. As for system recovery, LightKV scans all key-value pairs in PWB to rebuild them. According to our observation, when the PWB capacity is 8 GB, the recovery time takes 2.4 seconds using a single thread. As for larger PWB, the recovery can be further accelerated by using multiple threads.

Note that, in case of normal shutdown, LightKV persists the prefix tree and top leaf nodes of RH-Tree in PM. This helps LightKV to serve incoming requests directly after normal reboot.

VI. Evaluation

A. Experiment Setup

System and hardware configuration. We conduct all experiments on a server equipped with two Intel Xeon Gold 5215 CPU (2.5GHZ), 64GB memory and one Intel DC P3700 SSD of 400GB. We run CentOS Linux release 7.6.1810 with 4.18.8 kernel and use ext4 file system.

Persistent Memory. We use Intel Optane DC Persistent Memory in our evaluation. The firmware version of Optane DC PM is 01.00.00.3279. We configure Optane DC PM using App Direct Mode as follows. For the configuration of PM, we first create an ext4 file system on PM and mount it using DAX mode [2]. Then, we create a file of 64 GB on PM as a PM pool, and use mmap [3] to create page table for data accessing. As for writing data to PM, we use both clflush/clwb/clflushopt and ntstore/mfence to ensure persistence [18], [41]–[44].

Compared systems. We compare LightKV against LevelDB [17] and RocksDB [14], which are widely used persistent LSM-Tree based KV stores. We set MemTable size to 64MB and configure a Bloom filter using 10 bits per key to optimize lookup. Note that, LevelDB and RocksDB do not use PM and adopt asynchronous WAL.

NoveLSM [23] is an optimized KV store using PM to reduce (de-)serialization cost. We set up 8 GB persistent MemTable for NoveLSM, which is the same as the size of PWB in LightKV. Moreover, it still has in-DRAM MemTable, which is set to 64MB as in paper. The Bloom filter is configured to be same as LevelDB.

SLM-DB [22] is another recently proposed KV store using PM. It adopts B^+-Tree to index KV pairs in SSTables and conducts well-tuned garbage collection. In the original paper, the persistent MemTable of SLM-DB is set to 64MB. However, using a larger PM is more effective for SLM-DB to sort more data in memory, thereby reducing the frequency of compaction of data in the SSD. Therefore, we also set a Persistent MemTable of 8 GB for SLM-DB which in similar to LightKV and NoveLSM in our experiment.

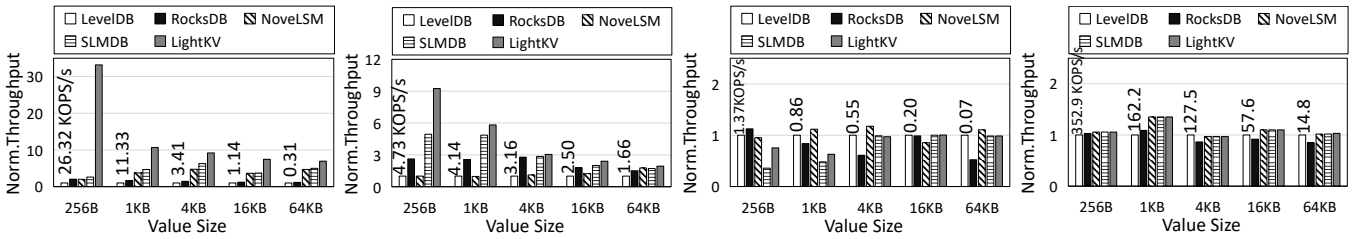
As for LightKV, we set *MAX_LEAF_NODES* to 2048, the segment size in PWB to 4 MB, and the compaction size to 4. Since LevelDB and NoveLSM only support 1 background thread to execute SSTable flush and compaction, we keep this configuration in our evaluation. As for RocksDB, SLM-DB, and LightKV, we use 2 background threads. For simplicity, compression is turned off for all key-value stores.

Workloads. We use db_bench [14], [17] as the micro-benchmark and YCSB [10] as the actual workload for evaluation. We execute 5 runs for each experiment and use average results.

B. Reducing write amplification

We first use db_bench to evaluate write amplification. Figure 9 shows the write amplifications of different KV stores with different data amounts while value size is 1KB. When writing totally 100 GB data, the write amplifications of LightKV are reduced by 7.1x, 5.1x, 2.9x and 2.3x compared to that of LevelDB, RocksDB, NoveLSM, and SLM-DB respectively. LightKV relies on RH-Tree to cluster keys with same key prefixes into the same partition. When executing compaction, LightKV only involves a few SSTables. On the contrary, the key range of a compacting SSTable usually overlaps with all SSTables in an upper level in LevelDB and RocksDB. This results in large write amplification. NoveLSM instead adopts a large persistent write buffer to reduce the compaction frequency. This partially reduces write amplification. However, when compacting on-disk SSTables, NoveLSM still faces the key range overlapping issue which results in large write amplification. SLM-DB organizes SSTables in a single level and performs restricted compaction. However, its compaction still needs to be conducted frequently in case of serving a large amount of data writes.

When the total amount of written data increases, the write amplification of LightKV remains stable (e.g. from 1.6 to 1.8 when the data amount increases from 50 GB to 100 GB). On the contrary, the write amplifications of other KV stores



(a) Random Write (b) Random Read (c) Range Query (d) Sequential Read

Fig. 10: **Normalized throughputs of basic operations for db_bench.** *These figures show the throughputs of different KV stores with varied value sizes, which are normalized to LevelDB.*

increase a lot. For example, the write amplification of SLM-DB increases from 3.9 to 5.9 when data amount increases from 50 GB to 100 GB.

C. Basic Operations

We then evaluate the throughputs of basic operations with varied value sizes, including random write, random read, range query, and sequential read. Figure 10 shows the results normalized to LevelDB.

Random write. Figure 10a shows the random write results with 100 GB KV pairs. LightKV outperforms LevelDB, RocksDB, NoveLSM, and SLM-DB by up to 33.4x, 16.3x, 16.0x, 12.7x and 13.5x, 8.3x, 5.0x, 4.0x on average. NoveLSM persists MemTable directly in PM to reduce (de-)serialization costs, and thus it achieves higher write throughput than LevelDB and RocksDB. SLM-DB adopts single-level data organization and carefully conducts compaction, which further reduces write amplification and in turn achieves higher throughput than NoveLSM. LightKV instead applies partition-based data compaction by only involving well-controlled number of SSTables in each compaction. This allows LightKV to greatly reduce write amplification and achieve the highest throughput. Moreover, instead of flushing the PWB data as a whole into SSD, LightKV flushes KV pairs in a per segment way. This avoids frond-end write blocking due to the background flush. When the value size increases, the OPS throughput decreases for all KV stores. However, LightKV still achieves highest write throughput.

Random read. We randomly read 20 GB data in a warmed up KV store with 100 GB KV pairs. Figure 10a shows the results. LightKV outperforms LevelDB, RocksDB, NoveLSM, and SLM-DB by up to 9.2x, 3.5x, 9.2x, 1.9x and 4.5x, 1.9x, 4.2x, 1.3x on average. LightKV uses RH-Tree to globally locate target KV pair in either PM or SSD, which provides fast querying performance. However, the other 4 KV stores need to use separate indexing to locate KV pairs. Note that, the B^+-Tree in SLM-DB only globally indexes data in SSD. SLM-DB still needs to search skiplist-based persistent MemTable. When the value size increases, the value reading time gradually dominates the whole execution time. Thus, all KV stores achieve similar throughputs.

Range query. We execute short range queries to fetch totally 20 GB data in a warmed up KV store with 100 GB KV pairs. Each range query fetches 100 KV pairs. As for LevelDB, RocksDB, and NoveLSM, different SSTables (except the ones in level 0) do not overlap with each other. On the contrary, both SLM-DB and LightKV controls the data volume involved in each compaction. This helps to reduce write amplification but results in more key overlappings among different SSTables. As a result, when serving range query, both SLM-DB and LightKV perform worse than the other three KV stores. For example, compared to LevelDB, the throughputs of SLM-DB and LightKV are reduced by 24.3% and 13.2% on average respectively as shown in Figure 10c.

Sequential read. We first warm up 100 GB KV pairs, and then sequentially read all of them. Since the data amount is large, all KV stores need to read SSTables from underlying SSD. Thus, they perform similarly.

Sequential load. LightKV performs poor when serving sequential writes. The sequential write throughput of LightKV is reduced by 35% on average compared to other KV stores. This is because LightKV needs to perform compaction within partitions even during sequential writes. However, the sequential write workloads usually occur in data loading stage. Thus, we can use multiple threads to accelerate sequential data loading. For example, when LightKV uses two threads to load data in parallel, it achieves similar throughput as other KV stores.

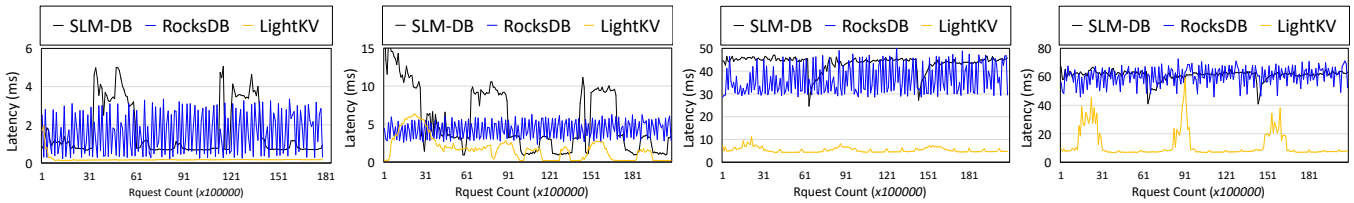
D. Tail latency under read-write workload

The read-write mix workload becomes common in recent years [36], which usually causes heavy tailed latency. Thus, we evaluate both read and write tail latency in this section. We modify db_bench to obtain read and write latencies of each request. We calculate the 99^{th} and 99.9^{th} percentile read and write latencies every 100,000 requests. We first warm up the KV store using 100 GB KV pairs. Then, we randomly write 100 GB data meanwhile randomly read 100 GB data. The key size is 16 B and the value size to 1 KB.

Figures 11a and 11c show the variations of 99^{th} percentile read and write latencies. Due to space limitation, we only show the results for LightKV, RocksDB, and SLM-DB. Overall, LightKV achieves the lowest and stable 99^{th} read and write latencies. This is because LightKV uses global indexing and well-controlled partition-based compaction to reduce both read and write amplification. Moreover, LightKV flushes PWB data into SSDs in a per segment way (e.g. 4 MB per segment)

KV Store	256B			1KB			4KB			16KB			64KB		
%	avg	99	99.9	avg	99	99.9	avg	99	99.9	avg	99	99.9	avg	99	99.9
LevelDB	294	2.7	5.8	370	3.6	6.5	429	4.3	7.0	653	5.7	8.7	829	6.1	10.2
RocksDB	106	0.3	3.6	152	2.2	5.1	274	4.1	6.9	671	5.9	9.1	1383	7.9	14.9
NoveLSM	243	0.7	3.5	321	1.4	8.0	399	4.1	9.8	524	6.2	11.1	891	8.7	18.6
SLM-DB	81	0.3	1.5	101	0.8	4.0	201	2.5	4.7	414	7.5	13.8	641	6.5	8.1
LightKV	58	0.2	0.4	74	0.2	2.4	131	0.8	3.6	258	3.0	5.5	409	4.2	6.5

TABLE I: **Read tail latency for db_bench.** *This table shows the average (ns), 99^{th} and 99.9^{th} percentile read latencies (ms) in read-write mixed workload for varied value length with db_bench.*



(a) 99^{th} read latency (b) 99.9^{th} read latency (c) 99^{th} write latency (d) 99.9^{th} write latency

Fig. 11: **Read and write tail latency in real-time.** *This figure shows the 99^{th} and 99.9^{th} percentile read and write latencies in real-time during read-write workload. We calculate the tail latency every 100,000 requests.*

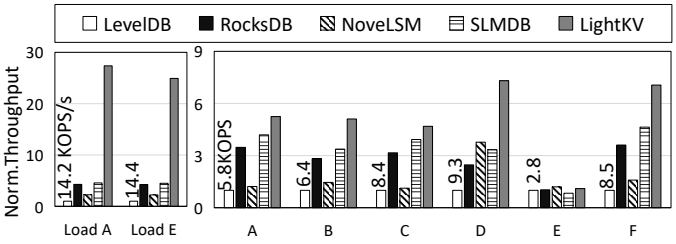


Fig. 12: **The throughput of different workloads for YCSB.** *This figure shows the throughput of different mixed workloads for YCSB which normalized to LevelDB. Workload A performs 50% reads and 50% updates; Workload B performs 95% reads and 5% updates; Workload C performs 100% reads; Workload D performs 95%reads for latest keys and 5% inserts; Workload E performs 95% range queries and 5% inserts; Workload F performs 50% reads and 50% read-modify-writes. Zipfian distribution is used for workload A, B, C, D and F and uniform distribution is used for workload E.*

to avoid heavy write tail latency. The 99^{th} read and write latencies of RocksDB reach up to 50 ms and 3.5 ms. These spikes are caused by the uncontrolled background compaction accompanied with large number of disk reads and writes. Unlike RocksDB, SLM-DB organizes SSTables into single level and performs restricted compaction to reduce write amplification. Moreover, it uses B^+ tree to index data in SSD to reduce read amplification. Thus, it achieves lower 99^{th} percentile read latency except a few latency spikes. SLM-DB will decide whether to perform compaction based on the coverage of the key value, which also cause the read latency spikes (e.g. up to 5.0 ms) for SLM-DB as shown in Figures 11a.

As for 99.9^{th} read and write latencies in Figure 11b and 11d, the results are similar. Note that, around serving 3 M[4], 9 M and 15 M requests, large latency spikes are observed in LightKV as shown in Figure 11b. This is mainly due to the compaction of large SSTables within a partition. However, LigthKV still achieves lower tail latency compared to other KV stores.

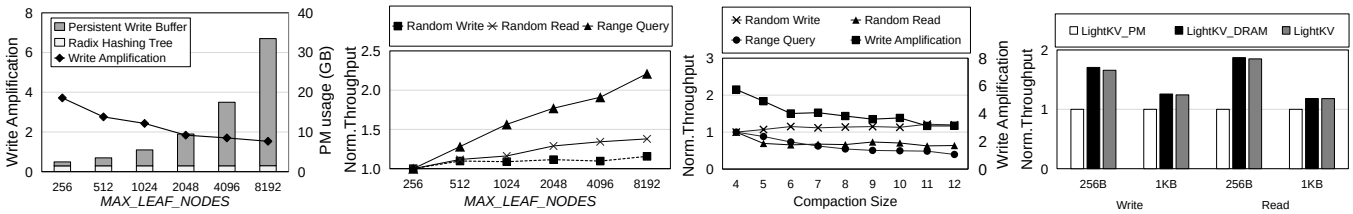
Table I shows the statistical latency results (including average, 99^{th} and 99.9^{th} percentile read latencies) for the whole testing period under read-write workloads. These results are tested with different value sizes. Similar to the real-time results, LightKV reduces the 99^{th} percentile read latencies by up to 17.9x, 10.5x, 6.4x, and 3.5x compared to LevelDB, RocksDB, NoveLSM, and SLM-DB. For 99.9^{th} percentile read latency, LightKV outperforms LevelDB, RocksDB, NoveLSM, and SLM-DB by up to 15.7x, 9.2x, 8.8x, and 3.4x. When the value size increases, the tail latencies of all KV stores increase correspondingly.

E. Results with YCSB

We use the six workload patterns in YCSB as real applications to evaluate LightKV. We first load 100 GB KV pairs into the KV store for Workload A (namely LoadA by randomly inserting KV pairs). Then we run Workloads A, B, C, F, and D in order. After that, we delete the database and reload 100 GB KV pairs for workload E (namely LoadE by randomly inserting KV pairs). Then we run Workload E. We use the default KV size in which key is around 20B and value is around 1KB. As for workloads A, B, C, D and F, the key distributions follow the default Zipfian distribution in YCSB.

Figure 12 shows the throughput results normalized to LevelDB. For all workloads (including LoadA and LoadE but except Workload E), LightKV achieves the highest throughput. For read-write mixed workloads A, B, F, due to the efficient search and write amplification reduction, LightKV

[4]M indicates million



(a) *MAX_LEAF_NODES* sensitivity (b) *MAX_LEAF_NODES* sensitivity (c) Compaction size sensitivity (d) Index placement sensitivity

Fig. 13: **Sensitivity analysis of LightKV.** *(a) shows the PM usage and write amplification, (b) shows the operation throughputs with different MAX_LEAF_NODES, which are normalized to MAX_LEAF_NODE of 256. (c) shows the operation throughputs with different compaction sizes, which are normalized to compaction size of 4. (d) shows the operation throughputs with different index placements, which are normalized to LightKV_PM.*

outperforms LevelDB, RocksDB, NoveLSM, and SLM-DB by up to 7.0x, 2.0x, 4.4x and 1.5x respectively. Workload C performs 100% reads, and the highly efficient RH-Tree indexing helps LightKV to achieve 4.7x, 1.5x, 4.2x and 1.2x higher throughput than LevelDB, RocksDB, NoveLSM, and SLM-DB. For workload D, it reads latest keys. Since NoevLSM, SLM-DB and LightKV maintain a large write buffer in NVM, the frequently accessed KV pairs are cached in the write buffer. Thus, their throughputs are higher than LevelDB and RocksDB. However, NoveLSM and SLM-DB adopts skiplist to index data in PM, whose performance decreases when the number of KV pairs increases. On the contrary, RH-Tree used in LightKV is able to serve large number of KV pairs. Workload E is dominated by range query. Thus, both LightKV and SLM-DB are affected by accessing multiple SSTables. Therefore, their performance is 10%-20% lower than other KV Stores. Load A and Load E in Figure 12 represent the performance of random write. Similar to the micro-benchmark results, LightKV outperforms LevelDB, RocksDB, NoveLSM, and SLM-DB by 27.4x, 6.4x, 12.2x, and 6.0x respectively.

Through experimental results, we can see that LightKV still achieves the highest throughput under different workloads.

F. Sensitivity analysis

1) MAX_LEAF_NODES: Figure 13a shows the write amplifications and the PM usages when configuring different *MAX_LEAF_NODES*. The MAX_LEAF_NODE is equal to the number of partitions. The increased MAX_LEAF_NODE results in more partitions. Thus, the PM usage increases to hold more partitions. Meanwhile, SSTables are distributed among more partitions and each partition is filled with less SSTables. This in turn reduces the compaction frequency, and thus brings less write amplification. As shown in Figure 13a, when *MAX_LEAF_NODES* reaches 8192, the write amplification can be reduced by 58% compared to using 256 partitions.

Figure 13b shows the throughputs of basic operations with different *MAX_LEAF_NODES*. When *MAX_LEAF_NODES* is 8192, the throughput of random write is 30% higher than that using 256 partitions. This is mainly due to the decrease of write amplification. The increased *MAX_LEAF_NODES* results in more partitions and less SSTables within each partition. range query benefits from this by incurring less disk IOs. Thus, using 8192 *MAX_LEAF_NODES* achieves the highest range query performance. As for random reads, more data are buffered in PWB with the increased partitions. Thus, the throughputs of random read increase by fetching data quickly from PWB.

2) Compaction size: Figure 13c shows the throughputs of basic operations with different compaction sizes. Since a small partition number is more likely to trigger compaction, we set *MAX_LEAF_NODES* to 64 here for clear observation on the sensitivity of compaction size. The increased compaction size results in less frequent compaction. This helps to decrease the write amplification and improve the write throughput. However, the reduced compaction results in more SSTables per partition, which lowers the search efficiency. As shown in Figure 13c, the throughputs of random read and range query decrease with increased compaction sizes.

3) Index placement: RH-Tree is placed across DRAM and PM. Here we implement different index placements to evaluate their efficiencies. Figure 13d shows the throughputs of random read and write for different index placements. LightKV refers to the proposed index placement in Section VI-F3. LightKV_PM maintains the whole RH-Tree in PM, and uses synchronous update to ensure strong consistency of RH-Tree. LightKV_DRAM instead maintains the whole RH-Tree in DRAM without persisting it.

Compared to LightKV_PM, LightKV incurs less expensive prefix tree operations in PM, and thus achieves higher read and write throughputs by 70%, 1.0x with the value size of 256 B. LightKV benefits from locating in-PM KV items using prefix tree and top leaf nodes that reside in DRAM. Thus, although LightKV_DRAM places the whole RH-Tree in DRAM, LightKV performs close to LightKV_DRAM. As the value size increases, the difference between LightKV and LightKV_PM becomes less obvious. This is because the KV item read and write dominate the performance. In summary, LightKV_PM can do instant recovery but at the cost of lower performance. LightKV_DRAM instead provides highest performance but suffers from costly system recovery for reconstructing the whole RH-Tree. On the contrary, LightKV balances the system performance and recovery cost.

VII. Related works

LSM-Tree redesign for NVM. Recently, a few research efforts have been made to optimize LSM-Tree using NVM. NoveLSM [23] uses NVM as large persistent MemTable to reduce the cost of data (de-)serialization. However, when using large persistent MemTable, NoveLSM still adopts skiplist to index data in persistent MemTable. This affects the system query efficiency. NVMRocks [12] is an optimization of RocksDB for NVM. It stores persistent MemTable in NVM. SSTables on NVM are stored on a file system optimized for NVM and use a special table format (PlainTable). SLM-DB [22] is a persistent memory based key-value store using a B^+-Tree to index on-disk data. SLM-DB organizes on-disk SSTables into a single level to reduce write amplification but at the cost of degraded range query performance. LightKV takes the similar approach with these works by using NVM as a persistent write buffer. However, LightKV especially focuses on efficient global indexing for locating cross-media data and well-controlled data compaction for reducing write amplification.

Optimizing LSM-Tree performance. PebblesDB [34] presents Fragmented Log-Structured Merge Trees (FLSM) to avoid rewriting data in the same level. WiscKey [30] separates key and value by storing keys in LSM-Trees and values in log. However, the values stored in log are usually unsorted. This results in degraded range query as a number of random reads are required to fetch data. Similarly, HashKV [7] also separates keys and values. Moreover, HashKV uses a hash function to partition the values to optimize garbage collection. LSM-Trie [40] uses tries to organize data which reduces write amplification. However, it does not support range queries. bLSM [36] proposes a new compaction scheduler to reduce write amplification. LOCS [38] improves the performance of LSM-Trees based on open-channel SSDs. cLSM [16] is designed to increase concurrency. SILK [6] proposes well-designed compaction mechanisms in RocksDB to reduce heavy tailed latency. On one hand, LightKV has similar optimization targets with these works. On the other hand, unlike these works, LightKV explore the usage of persistent memory in building persistent KV stores and does not restrict design choices based on LSM-Tree itself.

Optimizations for NVM-based index. There are a number of research works focusing on optimizing indexing structures when built in NVM. CDDS [15], NV-Tree [42], FP-Tree [33], wB+-Tree [9] and FAST-FAIR B+-Tree [18] are optimized for NVM-based B^+-Tree from the perspective of read/write performance and crash consistency. Instead, Path Hashing [43], Level Hasing [44] and CCEH [31] target to provide optimized hashing index scheme for NVM. WORT [26] is optimized based on the radix tree [28]. HiKV [41] proposes a hybrid index consisting of hash table and B^+-Tree based on hybrid DRAM-NVM memory. The hybrid index is used to support both single-point operations and range query efficiently. Recipe [27] provides a principled approach to convert DRAM indexes into persistent indexes. Unlike these works, RH-Tree in this paper focuses on improving inefficient indexing for cross-media data management. More importantly, the design of RH-Tree targets to help control data volume involved in data compaction.

VIII. Conclusion

In this paper, we propose LightKV a cross media key-value store with persistent memory. Similar to previous works, LightKV use PM to avoid costly write ahead logging. Moreover, on one hand, LightKV adopts a global index RH-Tree to efficiently manage data across memory and disk. By exploring the specific features of PM product, LightKV achieves balance between persisting index and providing high indexing performance. On the other hand, LightKV exploit the range partition of RH-Tree to conduct partition-based data compaction to greatly reduce write amplification. We evaluate LightKV against both PM-based KV stores and existing LSM-Tree based KV stores. The experiment results show LightKV greatly reduce write amplification and meanwhile improve PUT and GET performance by up to 33.0x and 9.2x.

IX. Acknowledgments

We thank the anonymous reviewers for their helpful comments. We thank Ying Wang and Huan Zhang for useful discussions. This work is supported by National Key Research and Development Program of China under grant No.2018YFB1003303, Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDB44030200, Beijing Natural Science Foundation under grant No. L192038, and Youth Innovation Promotion Association CAS.

References

[1] Intel and micron produce breakthrough memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[2] Linux dax file system. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[3] Linux programmer's manual mmap(2). http://man7.org/linux/man-pages/man2/mmap.2.html.

[4] Pmdk: Persistent memory development kit. https://github.com/pmem/pmdk.

[5] I. G. Baek, M. S. Lee, S. Seo, M. J. Lee, D. H. Seo, D. . Suh, J. C. Park, S. O. Park, H. S. Kim, I. K. Yoo, U. . Chung, and J. T. Moon. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004.*, pages 587–590, Dec 2004.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: preventing latency spikes in log-structured merge key-value stores. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 753–766. USENIX Association, 2019.

[7] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1007–1019. USENIX Association, 2018.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

[9] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, 2015.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154. ACM, 2010.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[12] Facebook. Myrocks. http://myrocks.io/.

[13] Facebook. Nvmrocks. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/.

[14] Facebook. Rocksdb. http://rocksdb.org/.

[15] Gregory R. Ganger and John Wilkes, editors. *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*. USENIX, 2011.

[16] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 32:1–32:14, 2015.

[17] Google. Leveldb. https://github.com/google/leveldb.

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200. USENIX Association, 2018.

[19] Intel. Intel optane dc persistent memory. https://newsroom.intel.com/news-releases/intel-data-centric-launch/#gs.7kv3ru.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[21] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315. USENIX Association, 2015.

[22] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205. USENIX Association, 2019.

[23] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005. USENIX Association, 2018.

[24] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, May 2015.

[25] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[26] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270. USENIX Association, 2017.

[27] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 462–477. ACM, 2019.

[28] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, April 2013.

[29] Jihang Liu and Shimin Chen. Initial experience with 3d xpoint main memory. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*, pages 300–305, 2019.

[30] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1):5:1–5:28, March 2017.

[31] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44. USENIX Association, 2019.

[32] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[33] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386. ACM, 2016.

[34] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 497–514. ACM, 2017.

[35] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156. USENIX, 2013.

[36] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228. ACM, 2012.

[37] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 163–176. USENIX, 2011.

[38] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel SSD. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 16:1–16:14, 2014.

[39] H. . P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[40] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82. USENIX Association, 2015.

[41] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362. USENIX Association, 2017.

[42] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181. USENIX Association, 2015.

[43] P. Zuo and Y. Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, May 2018.

[44] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476. USENIX Association, 2018.