PMDB: A Range-Based Key-Value Store on Hybrid NVM-Storage Systems

Baoquan Zhang, Haoyu Gong, and David H.C. Du, *Fellow, IEEE*

Abstract—Emerging Nov-Volatile Memory (NVM) may replace DRAM as main memory in future computers. However, data will likely still be stored on storage due to the enormous large size of available data. We investigate how key-value stores can be efficiently designed and implemented in a hybrid system, called NVM-Storage system, consisting of NVM as memory and traditional storage. We first discuss the performance trade-offs among Put, Get, and Range Query of the existing designs. Then, we propose PMDB, a range-based key-value store on NVM-Storage systems. PMDB achieves good performance for Put, Get and Range Query at the same time by utilizing a range-based data management and deploying a light-weight index on NVM. We compare PMDB with the state-of-the-art schemes including SLM-DB [21] and MatrixKV [40] for hybrid NVM-storage systems. Evaluation results indicate that in workloads with mixed Put, Get and Range Queries, PMDB outperforms existing key-value stores by $1.16\times - 2.49\times$.

Index Terms—Non-volatile memory, key-value store, log-structured merge tree, interval tree

1 Introduction

Byte-addressable Non-Volatile Memory (NVM) provides data persistence in memory speed [5]. Although NVM has a higher storage density than DRAM, it still may not be large enough to hold all data in this big data era [1], [41]. Therefore, in this paper we target a hybrid system, called NVM-Storage system, which includes both byte-addressable NVM as memory and block-based storage like Solid State Drive (SSD) or Hard Disk Drive (HDD) to achieve both high performance and large capacity at a reasonable cost.

In current storage infrastructure, key-value (KV) stores play an essential role in various application scenarios including data storage services [10], streaming platforms [28], machine learning pipelines [4], etc. Generally, KV stores provide three major access functions, Put (write a single KV pair), Get (read a single KV pair) and Range Query (a query to read multiple KV pairs within a specified key range). Deletion of a KV pair is implemented by a Put function with the key and a deletion mark as the value. Building KV stores on NVM-Storage systems to achieve a better performance can be very critical for many upper-layer applications. Although existing studies have explored and proposed several approaches [21], [22], [40], there are performance tradeoffs in these approaches for Put, Get and Range Queries.

Several approaches [22], [40] build KV stores on NVM-Storage systems based on Log-Structured Merge Tree (LSM-Tree) [30]. NVM is used as a write buffer to merge repetitive updates before writing KV pairs to storage. However, LSM-Tree trades performance of Get for that of Put. A Get in an LSM-Tree is inefficient since reading a KV pair will result in multiple storage accesses. [8], [9]. In addition, although an LSM-Tree provides a higher Put efficiency than other data structures, the level-based compaction of LSM-Tree introduces a relatively large number of data rewrites on storage. A KV store's Put performance can be further improved by reducing the number of data rewrites on storage [31], [39]. Since the compactions are typically triggered by inserting new KV pairs (Put functions), we consider the compaction cost as part of Put cost.

SLM-DB [21] is another approach that builds a B+ Tree index on NVM and stores KV pairs on storage in a single level. The B+ Tree identifies the location of each KV pair in storage. SLM-DB ensures a good Get performance since each Get in SLM-DB only requires one storage read after searching the B+ Tree in NVM for the location of the target KV pair. However, the global B+ Tree index and the single-level structure intensify the performance trade-off between Put and Range Query. That is, it can only ensure a good performance for either Put or Range Query, but not both at the same time.

To meet the challenge of performance trade-offs among Put, Get and Range Query, we propose PMDB, a range-based KV store on NVM-Storage systems that provides efficiently Put, Get and Range Query at the same time. To ensure good performance of Put, Get and Range Query, instead of using a single-level (like SLM-DB) or multi-level (like LSM-Tree) structure in storage, PMDB manages KV pairs on storage based on disjoint key ranges. A *Partition* consists of multiple disjointed key ranges on storage and an individual write buffer (MemTable) on NVM. The number of partitions is dynamically increased based on the number of KV pairs inserted. However, it will be upper bounded due to the NVM space limit for write buffers. The KV pairs are compacted (re-organized) with a new type of two-stage compaction called Partition and Range compactions. To

• *The authors are with the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455 USA. E-mail: {zhan4281, gong0098}@umn.edu, du@cs.umn.edu.*

Manuscript received 18 January 2022; revised 17 June 2022; accepted 4 August 2022. Date of publication 29 August 2022; date of current version 7 April 2023.
(Corresponding author: Baoquan Zhang.)
Recommended for acceptance by A.R. Alameldeen.
Digital Object Identifier no. 10.1109/TC.2022.3202755

further ensure a good Get performance, PMDB also builds an index on NVM. To reduce the update and NVM space overheads, instead of using a per-key index, PMDB uses a light-weight and block-based index which combining a binary search tree with a novel Interval Filter Tree (IFTree) on NVM. That is, a node in the binary search tree has a link pointing to a data block in a Sorted String Table (SST) (SST will be defined later) instead of an individual KV pair.

PMDB fully leverages the byte-addressable and data persistent properties of NVM to build complex indexes for key-value pairs on storage. Besides, NVM has a larger capacity than DRAM which allows PMDB partitions the storage and deploys a MemTable for each partition. We compare PMDB with the state-of-the-art schemes including SLM-DB [21] and MatrixKV [40]. Evaluation results indicate that PMDB is the only KV store achieving good performance for Put, Get and Range Query simultaneously. In workloads with mixed queries, PMDB outperforms other KV stores by $1.16\times - 2.49\times$.

The rest of the paper is organized as follows: In Section 2, we discuss the challenges of designing KV stores on NVM-Storage systems. The design of PMDB is presented in Section 3. The implementation issues of PMDB are discussed in Section 4. The comparisons of PMDB performance with existing schemes including SLM-DB and Matrix-KV are provided in Section 5. The related work of PMDB is included in Section 6 and we offer some conclusion in Section 7.

2 Challenges of Building Key-Value Stores on NVM-Storage Systems

Workload characterization [4] of KV stores indicates that the performance of Put, Get and Range Query is essential to the support of upper layer applications. Therefore, we focus on building KV stores on NVM-Storage systems to achieve good performance for Put, Get and Range Query. Existing research has proposed and exploited several approaches. However, there are trade-offs among the performance of Put, Get and Range Query. It is challenging to improve the performance of all of them with better trade-offs.

LSM-Tree-Based Key-Value Stores. LSM-Tree is a write-optimized data structure widely used in KV stores [10], [11], [13]. In the implementation of LevelDB [13] (a type of LSM-Trees), it buffers new KV pairs in a MemTable in DRAM first. Once the MemTable is full, it will be flushed to Level 0 (L_0) on storage as a Sorted String Tables (SST)(i.e., all KV pairs in an SST are sorted based on their keys). That means L_0 will include SSTs with overlapping key ranges. Beyond L_0, LevelDB stores KV pairs using multiple levels with increasing level sizes based on a configurable size ratio between adjacent levels. In each level above L_0, KV pairs are stored in SSTs with disjoint key ranges and each SST consists of multiple data blocks with sorted KV pairs.

A compaction process migrates data from a lower level to the next level if the lower level's size reaches a threshold. For example, a compaction can include one SST from L_1 and multiple SSTs from L_2 whose key ranges overlap with the key range of the SST from L_1. The compaction is executed by reading all involved SSTs from storage and merge them in memory. After the merging, the newly sorted SSTs are written back to L_2 on storage. An approach based on

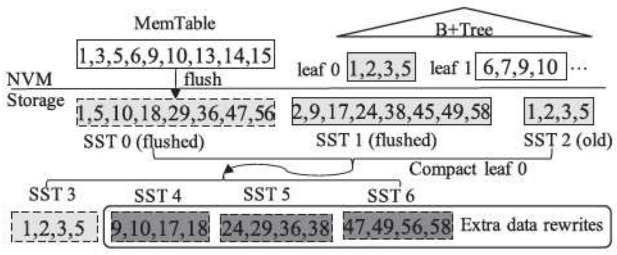


Fig. 1. SLM-DB.

LSM-Tree can achieve a higher write efficiency than other data structures since it only performs sequential writes and a compaction is triggered infrequently based on the size ratio between two adjacent levels. The level structure limits the total amount of data to be included in a compaction.

However, an LSM-Tree trades the performance of Get for that of Put. A Get in LSM-Tree has to check the MemTable, every SSTs in L_0, and one SST from each other levels sequentially until the KV pair is found or the highest and largest level is searched. Moreover, finding a key in an SST may also need multiple I/O accesses since an SST consists of multiple data blocks[14] (i.e., a data block is the basic unit of I/O accesses). As a result, a Get operation may lead to multiple random storage I/O accesses, thus degrading the Get performance significantly [8], [9]. Besides, although the Put efficiency of LSM-Tree can be higher than other data structures like B+ Trees, its Put efficiency can be further improved by reducing the number of data rewrites introduced by the leveled compaction[31], [39]. Several studies have built KV stores on NVM-Storage systems based on LSM-Tree [22], [24], [40]. Typically, a MemTable is used in NVM to hold and sort newly inserted KV pairs and a leveled structure like LSM-Tree is used in storage. Therefore, they suffer from the performance trade-offs of LSM-Trees and fail to provide good performance for Put, Get and Range Query at the same time.

Indexing KV Pairs on NVM. SLM-DB (as shown in Fig. 1) deploys a MemTable in NVM and store all SSTs on storage in a single level. Besides, SLM-DB builds a global B+ Tree in NVM recording the location of every KV pair in storage. SLM-DB ensures a high Get efficiency since a KV pair can be accessed from storage with only one storage read after searching the B+ Tree to identify its location. However, without compactions, the response to a Range Query can be inefficient since it may access an excessive number of overlapping SSTs. Besides, multiple versions of keys may exist in different SSTs which increasing the storage space overhead. Therefore, SLM-DB also deploys a selective compaction as described below to merge and sort overlapping SSTs in storage from time to time.

SSTs will be selected as compaction candidates under two conditions: 1) If the number of distinct SSTs referenced by keys in a leaf node of B+ Tree reaches to a given maximal number, all SSTs containing KV pairs referenced by the leaf node will be selected as compaction candidates; and 2) If the ratio of invalid keys in an SST is larger than a maximal ratio, the SST will also be selected as a compaction candidate. The selective compaction will be triggered if the total number of compaction candidates reaches a threshold.

The global B+ Tree and the single-level structure with selective compaction intensifies the performance trade-offs between Put and Range Query. That is, to achieve a good range query performance, the selective compaction will have to be performed/triggered more frequently impacting the put performance.

Considering the selective compaction, we plan to discuss its access properties based on the example shown in Fig. 1. KV pairs in a newly-flushed SST can be distributed over a wide key range and scattered over a large number of leaf nodes in B+ Tree. SST 0 is a newly flushed SST from NVM and contains KV pairs referenced by multiple leaf nodes of B+ Tree, e.g., leaf 0, leaf 1, etc. One newly flushed SST can cause an increase in the SST counts of several leaf nodes simultaneously. As a result, a good number of SSTs will be selected as compaction candidates that may increase the frequency of selective compactions.

Besides, the selective compaction may introduce some extra data rewrites by including SSTs whose key ranges are significantly different from each other, but overlapped in a small range. For instance, in Fig. 1, SST 0 and SST 1 are newly flushed from NVM with a larger size and wider key ranges than the existing SST 2. All these three SSTs will be selected as compaction candidates if leaf 0 reaches its maximal count of SSTs. Then, they will be merged and rewritten during the compaction. However, the newly flushed SST 0 and SST 1 also include many KV pairs with their keys out of the range of leaf 0.

Finally, the write performance can be further impacted by the required relatively large NVM space and update overhead of the global B+ Tree. Since the B+ Tree stores every KV pair's location, its size becomes larger with an increasing number of KV pairs. With a fixed NVM size, the available NVM space for write buffer becomes less. A smaller write buffer may increase the total number of data writes to storage since it can absorb fewer updates before flushing out to storage. Besides, the B+ Tree must be searched and updated for each KV pair when creating new SSTs during compactions. The performance of searching and updating the B+ Tree can be significant if we assume that the read/write performance NVM is multiple times slower than that of DRAM.

Summary. LSM-Tree based KV stores suffer from the performance trade-off between Put and Get. The Put performance can also be further improved by reducing the number of data rewrites introduced by the leveled compaction. SLM-DB ensures a good Get performance by building B+ Tree in NVM for every KV pair in storage. However, the performance trade-off between Put with compactions and Range Query is intensified by the global B+ Tree and its single-level structure. Therefore, our goal is to design a KV store on NVM-Storage systems which can better deal with the performance tradeoffs and achieve high performance for Put, Get and Range Query at the same time.

3 PMDB Overview

3.1 PMDB Architecture

Fig. 2 shows the overall architecture of PMDB. PMDB includes two major mechanisms: a range-based data management and a light-weight NVM index to achieve a better performance trade-off among Put, Get and Range Query.

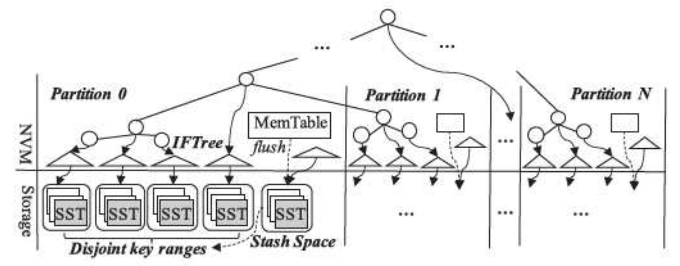


Fig. 2. Overall architecture of PMDB.

Range-Based Data Management. Instead of a single-level or multi-level structure, PMDB stores KV pairs on storage in small disjoint key ranges. NVM has a larger capacity than DRAM. Existing studies assumes that the NVM space can be 10% of the total data set [22], [40]. Therefore, PMDB deploys multiple write buffers (MemTables) on NVM: one for each partition. Each partition represents a disjoint key range that consists of a number of SSTs. If we use a single large MemTable, a newly-flushed SST will overlap with a large number of key ranges on storage which may increase the data rewrites on storage in later compaction process. Besides, flushing a large MemTable may also impact the Put performance [3], [40]. On the other hand, a small MemTable may not be very efficient for flushing data to storage. Therefore, a reasonable size of MemTable has to be chosen. That is, in PMDB, a fixed size MemTable is used. With a fixed size NVM, this will limit the number of partitions.

As shown in Fig. 2, PMDB partitions the possible key range into a number of disjoint key ranges with each as a partition, and maintains a MemTable for each partition. Each flush operation will only flush one MemTable in a partition to reduce the number of overlapped key ranges for the newly-flushed SST and to avoid impacting foreground write operations significantly. At the beginning of a KV store, there is only a single partition with one very wide key range. The number of partitions will increase with more KV pairs are inserted until a limit is reached. Since MemTable needs to maintain a certain size for better I/O performance, when the number of KV pairs is continuously inserted, the number of disjoint key ranges (represented by the nodes in the binary search tree) will increase, but not the number of partitions. Thus, a partition may consist of several consecutive smaller disjoint key ranges. That is, a partition is represented by an intermediate node in the binary search tree and all nodes under this node represents disjoint key ranges of this partition.

As we discussed in Section 2, compacting overlapping SSTs whose key ranges are widely different may result in an extra number of data rewrites. Therefore, PMDB utilizes a two-stage, partition and range, compaction to reduce the frequency of compactions. It also ensures that any compaction will only include SSTs with largely overlapped key ranges.

As shown in Fig. 2, PMDB will flush the filled up MemTable in a partition to a *Stash Space* as an SST. That is, each partition has a separate *Stash Space*. After the number of SSTs accumulated in the *Stash Space* reaches a limit, a partition compaction will be triggered to merge and sort these potentially overlapped SSTs since they cover the same key range in a partition. A set of new disjoint SSTs are created based on the existing disjoint key ranges in the partition as the result of compaction. Then these disjoint SSTs will be added into their corresponding existing key ranges. When adding a new SST to a key range, the existing SSTs in the key range will not be resorted and rewritten. Therefore, each key range may also include several overlapped SSTs. A range compaction may be triggered if a key range accumulates enough number of overlapped SSTs. As the result of range compaction, several smaller key ranges are formed in the original key range and each key range corresponding to one of the newly formed SSTs.

With this range-based data management, to maintain a good performance of range queries, compactions do not have to be triggered frequently. In the proposed two-stage compaction, each compaction will only involve SSTs with closely overlapped key ranges that further reducing the data rewrites introduced by the compaction. Therefore, the Put performance with compactions will be less impacted. However, a key range on storage including overlapping SSTs may impact the Get performance. Therefore, to improve Get performance PMDB also builds special indexes for overlapping SSTs on NVM.

Light-Weight NVM Index. PMDB uses a binary search tree in NVM to index both partitions and disjoint key ranges on storage. That is, some intermediate nodes of the binary search tree correspond to the key ranges of partitions and the leaf nodes correspond to disjoint key ranges in storage. The space overhead for the global binary search tree is negligible compared to those of IFTree and MemTable. The global binary search tree indexes the disjoint key ranges on storage. Since either a *Stash Space* or a key range may involve overlapping SSTs that decreases search efficiency for Get. To improve the search efficiency with overlapping SSTs, PMDB builds an Interval Filter Tree (IFTree) in each *Stash Space* and each disjoint key range as shown in Fig. 2 (represented as triangles). A node of an IFTree points to a data block in an SST. An IFTree consists of pointers to the sorted data blocks of overlapping SSTs based on their corresponding key ranges, and stores a bloom filter for each data block. When adding a new SST to a key range or a *Stash Space*, block information of the new SST will be added to its corresponding IFTree. The detailed design of IFTree will be further discussed later in Sections 3.3 and 4.1.

When searching a key, we follow the binary search tree to identify the partition and the key range under the partition that potentially hold the KV pair. Then, we search the key with the order of MemTable of the partition first, *Stash Space* of the partition and the corresponding key range. In the *Stash Space* or the key range, data blocks potentially including the key can be quickly identified via its corresponding IFTree. Also, unnecessary storage reads can be avoided with their bloom filters. An IFTree indexes the data blocks consisting of multiple KV pairs. Therefore, the total size of an IFTree is small. Besides, the false positive rate of a bloom filter is typically small. For example, in the default setting of LevelDB, the false positive rate of a bloom filter is about 0.03. The amortized number of storage reads for each search can be a little bigger than one.

Compared to SLM-DB which has per-key indexes in NVM, PMDB builds indexes in NVM based on data blocks on storage. Since a block have multiple key value pairs, the total size and update overheads of PMDB NVM index are much smaller. Besides, SLM-DB has a single-level structure on storage. The single-level structure intensifies the performance trade-off between Put and Range Query. An SST in the single-level structure may overlap with all of the existing SSTs. Thus compaction can be triggered frequently and each compaction may include a large amount of SSTs. PMDB inherits some designs from SLM-DB. For example, compactions in PMDB can also be triggered by invalid key ratios. Besides, PMDB uses a partition structure for storage so that SSTs only overlap with those in the same partition. PMDB also deploys a two-stage compaction to ensure that a compaction in PMDB only compacts SSTs with similar ranges.

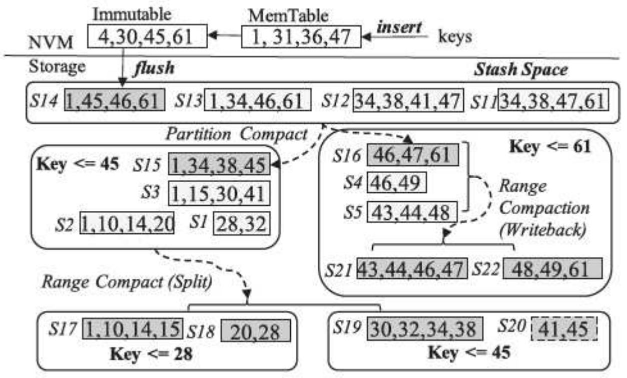


Fig. 3. Two-stage compaction.

3.2 Two-Stage Compaction

The purpose of the compaction is to improve the Get/Range Query efficiency and remove invalid KV pairs. Therefore, PMDB sets two thresholds to trigger a compaction: the number of storage I/Os to search a key in the worst case (*max_io*) and the invalid key ratio (*invalid_ratio*) (the same as used in SLM-DB). We will discuss how to set these two thresholds later. Besides, PMDB also utilizes an additional seek-based compaction to improve the performance of range queries. If range queries are executed frequently over a particular key range, PMDB will also more frequently compact the overlapping SSTs in the key range. With these compaction parameters, PMDB can be tuned to either reduce the number of rewrites during compactions or to improve the Get performance. The number of SSTs involved in each compaction is also reduced since a stash space or a key range will only include a limited number of overlapping SSTs.

Fig. 3 shows the two-stage compaction process within the key range of an MemTable in a partition. The dark gray boxes are new SSTs produced by compactions. We represent an SST with a format of $\{key_1, key_2,..., key_n\}$. For the convenience of presentation, we also mark each SST with an SST ID, e.g. *S1*, *S2*, etc. In PMDB, an SST ID is unique and incremental. New KV pairs inserted into this partition will be first buffered in MemTable. Each MemTable is implemented as a SkipList (the same as in [21], [22]). A MemTable will be flushed to *Stash Space* as an SST if its size reaches a threshold/limit. Then the two-stage compaction is executed with the following process.

Partition Compaction. In Fig. 3, the *Stash Space* has four overlapping SSTs: *S14*, *S13*, *S12*, and *S11*. Once a partition compaction is triggered, the SSTs in the *Stash Space* will be merged. The newly generated SSTs, *S15* and *S16*, are disjoint, and created based on key range $k \leq 45$ and $45 < k \leq 61$. Two new SSTs, *S15* and *S16*, are produced since the compaction results include 7 key-value pairs and the maximal number of key-value pairs in an SST is 4 in our example. Finally, the two new SSTs are added to their corresponding key ranges without rewriting and merging with the existing SSTs. However, the data blocks in the new SSTs are updated in their corresponding IFTrees.

Range Compaction. A range compaction may be triggered to merge overlapping SSTs in a key range. A key range has a capacity limit (20 SSTs by default in PMDB) to reduce the time and space overheads of a compaction. A range compaction has two possible operations: write back or split. If a compaction produces more than 20 SSTs, a key range will be split into two ranges. Otherwise, the produced SSTs will be written back to the current key range. In Fig. 3 the range compaction for $45 < key \leq 61$ will execute the write back operation (*Range Compaction Writeback*). That is, new SSTs, *S21* and *S22*, are written back to the key range $45 < key \leq 61$. The compaction for the key range $key \leq 45$ performs a split (*Range Compaction Split*). That is, two smaller key ranges, $key \leq 28$ and $28 < key \leq 45$, are created using the median key 28 of the compaction results. The key range split does not introduce extra data rewrites since it is executed during a range compaction. A key range splitting will create a new key range on storage and a MemTable on NVM. When PMDB runs out of NVM space, key ranges will stop further splitting. PMDB can still serve writes and write back new key values to storage without range splitting. However, the number of SSTs in a key range will keep increasing leading to a worse write and read performance.

Trade-Offs Introduced by Compaction Parameters. The basic principle is that if frequent compactions are triggered, the performance of Get/Range Query gets better and a better space efficiency can be achieved while the Put efficiency will be worst. The compaction parameters, *max_io* and *invalid_ratio*, can trigger compactions for different purposes. They also introduce different efficiency trade-offs among Put, Get, Range Query and space.

The *max_io* is set to guarantee a worst-case Get/Range Query performance. When *max_io* is large, a key range or a *Stash Space* may accumulate more overlapping SSTs before triggering a compaction. PMDB will have a better Put efficiency since compactions are triggered less frequently. However, it decreases the Get/Range Query efficiency. It may also decrease space efficiency since more versions of KV pairs can exist at the same time.

The *invalid_ratio* is to improve the space efficiency. If the *invalid_ratio* is small, compactions will be triggered more frequently if with excessive updates. The Get/Range Query efficiency can also be improved since there will be fewer overlapping SSTs. However, the Put efficiency will be degraded. The *invalid_ratio* may have a significant performance influence in update-intensive workloads.

The characteristics of real-world workloads show that range queries has localities and certain key ranges may be queried more frequently [4], [15]. Therefore, PMDB also

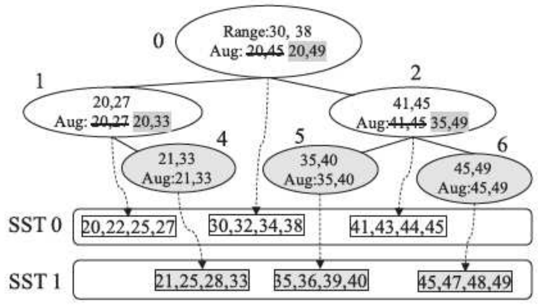


Fig. 4. Indexing overlapping SSTs using IFTree.

deployed a seek-based compaction. If a key range is being queried frequently, a compaction will also be triggered. The seek-based compaction improves the read performance, especially the performance of range queries. Besides, the seek-based compaction limits its impact on Put efficiency by only triggering compaction on frequently-queried ranges.

3.3 Light-Weight NVM Index with IFTrees

As shown in Fig. 2, PMDB indexes key ranges using a binary search tree. Each key range (pointed by a leaf node) and the stash space of a partition may include overlapping SSTs. IFTrees are used on NVM to index overlapping SSTs to increase the read efficiency.

Fig. 4 shows how an IFTree indexes the data blocks in overlapping SSTs. A node of IFTree includes the key range of each data block (Range) and an Augment information (Aug). It also contains a link that points to its corresponding data block (shown by a dashed link in Fig. 4). Considering at the beginning, there is only *SST 0*. SST 0 has three data blocks whose ranges are {20, 27}, {30, 38} and {41, 45}. Therefore, an IFTree is created with three nodes, *nodes 0, 1 and 2*, and sort them based on their start keys. The augment (Aug) of a tree node records the smallest and the largest keys in their subtrees. Therefore, with only SST 0, the augments of nodes 0, 1 and 2 are {20, 45}, {20, 27} and {41, 45} respectively. When *SST 1* is added including three more data blocks overlapping with those in *SST 0*. The IFTree will add three new nodes: *nodes 4, 5 and 6*. The Augments of exiting nodes will be updated based on the key ranges of data blocks in SST 1.

When searching a key, subtrees of a node can be fanned out based on its augment information. For example, the searching process of key 43 can start from *node 0*. The key range of *node 0* {30, 38} indicates that the indexed data block does not contain the target key 43. Before we search the children of *node 0*, we will further check its augment. The augment of *node 0* is {20, 49} which includes the target key 43. Therefore, we will further check its two children: *nodes 1 and 2* of *node 0*. However, neither the key range nor the augment of *node 1* includes the target key 43. Thus, the children of *node 1* will be removed from the follow-up searching process. Finally, we can identify that the only data block indexed by *node 2* (i.e., {41, 45}) includes the target key 43. Since we may search multiple overlapping data blocks for a given key, a bloom filter of its corresponding data block is also stored in the tree node to avoid unnecessary storage reads. Only if the bloom filter of a possible data block does

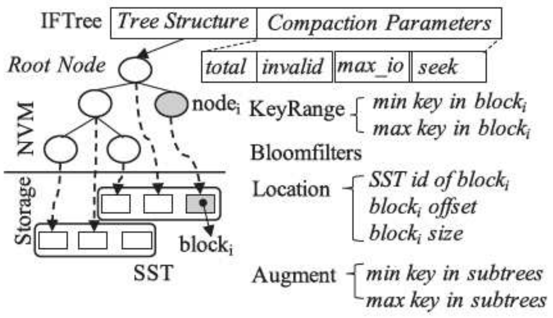


Fig. 5. IFTree implementation.

potentially include the searched key, the data block will be accessed from storage.

Since the IFTree indexes data blocks instead of each KV pair, the sizes of binary search tree and IFTree are reduced. PMDB can save more NVM space for write buffers (i.e., more partitions). More repetitive updates can be merged in NVM further reducing the data rewrites on storage. Besides, an IFTree is also updated less frequently since it will only be updated for each data block instead of each KV pair. Therefore, the update overhead of the NVM index is also reduced and it can avoid impacting the write performance.

4 PMDB Implementation

This section discusses the implementation of PMDB including the implementation of IFTree and the range-based data management. We also introduce the interfaces of PMDB.

4.1 IFTree

IFTree is a critical data structure to index the data blocks in overlapping SSTs in a *Stash Space* or in each disjoint key range. Essentially, IFTree is a specially-designed augmented interval tree [12]. To be self-balance, an augmented interval tree can be implemented based on AVL-Tree [35] or red-black tree [17]. In PMDB, IFTree is implemented by red-back tree since a red-black tree can achieve better insert performance and less additional memory cost [17].

In the augmented interval tree implemented based on red-black tree, node augments do not need to be modified for re-color operations since the tree structure does not change. During a rotation, augments of the involved nodes will be updated according to their new children [12].

Fig. 5 shows the implementation of an IFTree. An SST includes multiple data blocks (we assume the default block size is 4KB) [14]. Therefore, an IFTree is a tree structure with each node of the tree referring to a data block in an SST (shown by dashed pointers in Fig. 5).

Searching A Key. Algorithm 1 shows the process of searching a key. The search process in an IFTree starts from the root node with a Breath-First Search (BFS). When BFS arrives at $node_i$, the KeyRange and bloom filter of $node_i$ will determine if the target key exists in the data block. That is, only if the key range includes the target key and the bloom filter tested positive, we will access and check the data block. Otherwise, the node's augment including the min and max keys of the data blocks in its subtrees and it is used to make the fan out decision of its subtree. If the target key is not in the range of augment of $node_i$, the subtrees of $node_i$ will be excluded from the follow-up BFS.

Adding A New Data Block. IFTree is a particular type of binary search tree. The nodes of an IFTree are sorted based on the start keys of the indexed data blocks. When add a new $block_i$, a new $node_i$ is created. The place to add the $node_i$ can be identified by binary searching the IFTree based on the max value of the corresponding key range of the data block. Since the nodes visited during this binary search are ancestors of $node_i$, the augments of visited nodes should be updated using the key range of $node_i$.

Algorithm 1. Searching a Key in IFTree

```
def search(key, locations)
to_visit.append(root)
while not to_visit.empty do
  node = to_visit.pop()
  if node and key in node.range, node.bloomfilter then
    locations.append(node.location)
    to_visit.append(node.left)
    to_visit.append(node.right)
  end if
end while
```

Updating Compaction Parameters. Both a disjoint key range and the *Stash Space* of a partition use an IFTree to index the data blocks of overlapping SSTs. Therefore, PMDB also records the parameters to trigger compactions in the IFTree. Compaction can be triggered, either in a key range or a *Stash Space* by three thresholds, *max_io*, *invalid_ratio* and *seek*. The *max_io* is the required number of storage I/Os to search a key in the worst case among the overlapping SSTs. The *invalid_ratio* is the ratio of invalid keys that can be calculated with the number of invalid keys (*invalid*) and the total number of keys (*total*). The *seek* is the number of range queries searched in the IFTree. The *max_io* and *invalid_ratio* will be updated after adding data blocks of a new SST.

Calculating the exact *max_io* and *invalid_ratio* can be difficult. A key range may include both disjoint SSTs from the last range compaction and newly added overlapping SSTs. Besides, IFTree indexes data blocks instead of KV pairs. It is almost impossible to accurately identify whether a KV pair is a new insert or an update. On the other hand, PMDB does not require the exact *max_io* and *invalid_ratio* to trigger compactions. Therefore, we estimate *max_io* and *invalid_ratio* as follows. In most conditions, a newly generated SST will overlap with most of the existing SSTs in a key range or in a *Stash Space*. Accordingly, for each new SST, we increment the *max_io* of the IFTree by one. We also estimate the *invalid_ratio* based on the bloom filters of data blocks. For each new data block, PMDB identifies all of the overlapping blocks. Then PMDB checks the bloom filters of overlapping blocks for each new key. If the bloom filter of a block indicates that the key may exist (tested positive), the *invalid* is incremented by one. The *invalid_ratio* is calculated by *invalid* and *total* which is the total inserted KV pairs in the key range.

Space Overhead. In the build-in benchmarks, dbbench [4], of LevelDB and RocksDB, the default sizes of keys, values and data blocks are 16B, 100B and 4KB respectively. A 4KB

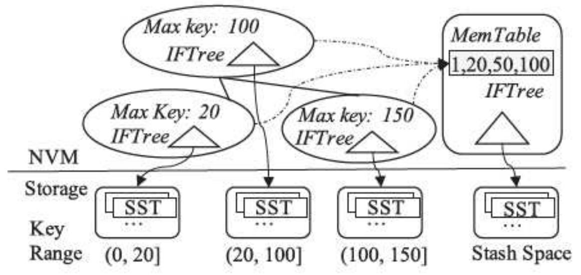


Fig. 6. Data structure in a partition.

data block includes 32 KV pairs. Besides, by default, a bloom filter includes 10 bits per key. Therefore, KeyRange of a node is 32 bytes including two keys (min and max). The size of a bloom filter is 40 bytes. The location address will be of 16 bytes including an 8-byte SST ID, a 4-byte block offset, and 4-byte block size. The augment of a node is also 32 bytes including two keys. Other tree structure information includes a 1-bit node color, an 8-byte left/right child pointer. Therefore, for a 4KB data block, the size of a node of an IFTree is 136 bytes, and the total size of an IFTree is about 3.32% (136B/4KB) of the total data set size. The required NVM space can be less if we use a larger block size.

4.2 Partitions and Key Ranges

To support Get and range queries, PMDB uses a Binary Search Tree (BST) on NVM to index disjoint key ranges in storage. Multiple disjoint key ranges are grouped by a partition (represented by an intermediate node) and each partition has a dedicated MemTable.

Fig. 6 shows the data structure of a partition. A node of the BST stores the maximum (max) key of a key range in storage. Besides, to further improve the search efficiency on overlapping SSTs in a key range, each BST leaf node (corresponding to a disjoint key range) also includes an IFTree. Key ranges in the same partition (represented by an intermediate node called partition node) will share an MemTable. When searching a key, the search process will then be executed with the ordering of MemTable of a partition, IFTree of the *Stash Space*, and the IFTree of the key range covering the target key.

PMDB creates partitions by splitting from a single partition. Thus, workloads will be balanced among different partitions if the initial portion of the workload represents the whole workload distribution. Otherwise, the partition can be initially set up based on the understanding of the key distribution of the workload. In this paper we simply assume the former case.

PMDB deploys MemTable whose size is the same to an SST for each partition. Assume that each partition includes at most 10 disjoint key ranges. Each key ranges can store at most 10 SSTs. The NVM space requirements for all MemTables will be about 1% of the total data set.

Overall, the minimal NVM space requirements of PMDB for both IFTree and MemTables can be about 5% of total data set. PMDB can be tuned for more NVM budgets by decreasing the maximal numbers of overlapping SSTs in a key range on storage.

We implement BST and IFTrees using Persistent Memory Development Kits (PMDK) [20]. To ensure data consistency, *libmemobj transactions* [19] are used to modify the NVM data structure. With much slower storage (compared with NVM speed), the KV store's overall performance will be dominated by the number of storage accesses. The overhead to ensure data consistency on NVM will not significantly impact the overall performance. We have verified that the performance overhead to ensure data consistency on NVM is not significant by comparing with the overall write performance of KV store with and without transaction guarantee.

4.3 Key-Value Store Interfaces

As a KV store, PMDB provides the following interfaces.

Put is to store a new KV pair. A new KV pair will be inserted into the corresponding MemTable in a partition. Deletions are executed using **Put** with a delete mark in the *Value*. Invalid KV pairs will be dropped during compaction.

Get is to search a KV pair. PMDB will search the BST to find the key range that possibly holding the KV pair. Then, it searches the target key with the order of MemTable of the partition containing the possible key range, *Stash Space* of the partition that may store the target key, and the possible key range containing the target key.

Iterator is for a range query. It consists of a *Seek* and a *Next* function. When executing a range query, PMDB performs the *Seek* using a start key and then call *Next* several times. An iterator in PMDB consists of sub-iterators of a MemTable in a partition, the SSTs in *Stash Space*, the SSTs in a disjoint key range. During a *Seek*, PMDB will locate the positions of the sub-iterators using the start key. For each of the *Next*, PMDB will iterate all sub-iterators simultaneously and choose the smallest key as the next key.

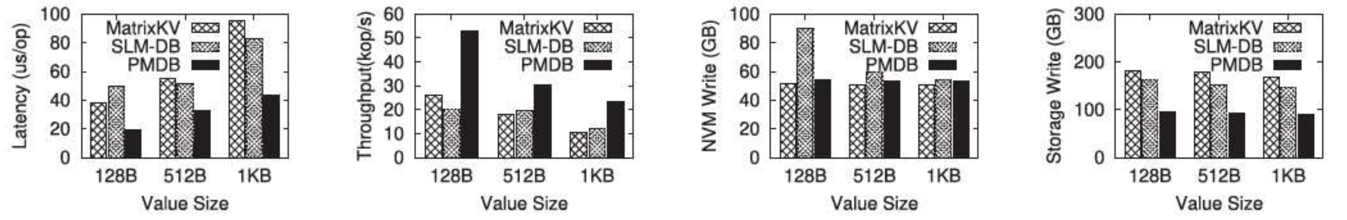
Recover is called after the system restarts after a failure. PMDB maintains a root object at a specific location. The root object stores the root node of the BST such that the index of KV store can be recovered.

5 Performance Evaluation

5.1 Experimental Setup

Limited by the available resources, we use NVDIMM [33] and spin for a specified duration with a time stamp counter in its write and read interfaces to emulate different NVM devices which is similar to prior works [6], [38]. We set write and read latency of the NVM to 500ns (5$\times$ that of DRAM) and 200ns (2$\times$ that of DRAM) respectively [27], [29].

We use an SSD to store SSTs through an ext4 file[25]. The SSD (Dell 400-AFKX, SATA interface) has a 600Mbps data transfer rate and a 480GB capacity. Following the approach of some existing systems [21], [22], we implement PMDB by modifying LevelDB [13]. We compare PMDB to two state-of-the-art KV stores designed for NVM-Storage systems: MatrixKV [40] and SLM-DB [21]. We use MatrixKV to represent KV stores using LSM-Tree based structure in storage since it outperforms other designs [22], [24], [40]. SLM-DB keeps one level in storage and builds an index for each KV pair in NVM. We use two threads, one for executing benchmarks, and the other for background compaction. In all experiments, we disable data compression and enable bloom filters (10 bits per key). We set the default SST size as 2MB, block size as 4KB, and an internal block cache as 8MB.



(a) Latency (b) Throughput (c) NVM write (d) Storage write

Fig. 7. Put (Random Write).

Since SLM-DB is not open-sourced, we also implement SLM-DB based on LevelDB. In our implementation, an entry in the B+ Tree is 32 bytes, including a 16-byte key and 16-byte location information (i.e., SST ID, block offset, and block size). The size of the location information in SLM-DB is the same as that in PMDB. In SLM-DB, we use similar configurations in [21] including the live key ratio (0.7), the leaf node threshold (10), and the sequential degree threshold (0.8). Since SLM-DB builds an index for each KV pair, bloom filters are not needed.

In PMDB, the *max_io* and *invalid_ratio* are set to 10 and 0.3 respectively which are the same as used in SLM-DB. The seek count to trigger seek-based compaction is set to 3. The largest possible number of SSTs in a key range is set to 20. Besides, the default NVM partition size is the same as the maximal file size (2MB).

5.2 Micro-Benchmarks

We use the built-in benchmark tool, dbbench [4], to compare the performance of MatrixKV, SLM-DB, and PMDB by running different benchmarks including random workloads and mixed skewed workloads. We also discuss the recovery and space overheads, and analyze the sensitivity of PMDB by varying the configurations.

5.2.1 Random Workloads

Random Writes (Put). We first discuss the write performance of each design. We run random writes with 50GB in a total data set which is similar to the prior work in [31], [39]. We vary the *Value* sizes among 128B (400 million KV pairs), 512B (100 million KV pairs), and 1KB (50 million KV pairs). The NVM size is set to 10% of the total data size which is close to the configurations used in [22], [40].

We keep the same total NVM budget for each system. PMDB and SLM-DB will use NVM to store both index structure and MemTables. However, when the *Value* size is 128B, the size of B+ Tree in SLM-DB is about 25% (32B/128B) of the total data set. The 5GB NVM is not large enough to store the B+ Tree index of SLM-DB. PMDB builds an index for each data block (4KB). The index structure on NVM of PMDB is about 3% of the total data set size which can be stored in the 5GB NVM. To compare PMDB and SLM-DB with small KV pairs, in the evaluation with 128-byte *Values*, we assume the index of SLM-DB can be stored in NVM and the write buffer size of SLM-DB is the same as that of PMDB.

Fig. 7 shows the performance of the random writes including the latency (Fig. 7a) and throughput (Fig. 7b). We also measure the sizes of the total data written to NVM (Fig. 7c) and to storage (Fig. 7d). Fig. 7d indicates that PMDB achieves the smallest amount of data written to storage. Compared to MatrixKV and SLM-DB, PMDB reduces the total amount of data written to storage by 47.17% – 48.32% and 39.46% – 42.9% respectively.

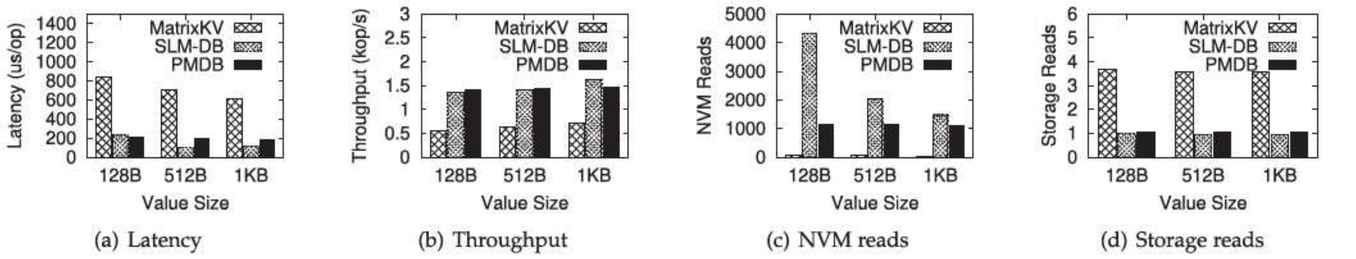
Fig. 7c shows that MatrixKV achieves the least amount of NVM writes since it only stores Level 0 in NVM. All compactions in MatrixKV will not generate any updates in NVM. SLM-DB and PMDB maintain index structures in NVM. The index will be updated during Puts and compactions. The total size of NVM writes in SLM-DB is significantly influenced by the number of KV pairs. With a given total data set size, a smaller *Value* means a larger number of KV pairs. Since SLM-DB stores an index entry for each KV pair, more KV pairs lead to a larger NVM writing size. PMDB only builds an index referred to each data block. Therefore, the *Value* size has less influence on its NVM writing size.

When the *Value* size is 128 bytes, the total NVM writing size, including writing to MemTables and updating the index, of PMDB is only about 4.61% higher than that of MatrixKV. Compared to SLM-DB, PMDB reduces the NVM writing size by 40.24%. The size of NVM writes due to index updates is reduced by 90.12%. When the *Value* size is larger (512 bytes and 1KB), the NVM writing size of SLM-DB becomes smaller. PMDB still reduces the total size of NVM writes by 9.92% and 5.32% for 512-byte and 1KB *Value* sizes respectively.

However, Figs. 7a and 7b show that the performance of SLM-DB is not better than that of MatrixKV with a *Value* size of 128-bytes due to the larger performance overhead for index updates. PMDB achieves the best write performance since it reduces the writing sizes to NVM and to storage simultaneously. PMDB reduces the write latency with different *Value* sizes by 50.14% – 54.89% and 48.74% – 61.63% compared to those of MatrixKV and SLM-DB respectively. The throughput of PMDB is 2.02× – 2.28× and 1.92× – 2.61× higher than those of MatrixKV and SLM-DB.

Random Reads (Get). We present the results of random read evaluations after random write experiments. Similar to [21], the number of reads covers about 20% of the total KV pairs to reduce the total execution time. We show the performance of random reads including latency (Fig. 8a) and throughput (Fig. 8b). Similar to [8], [9], we also measure the number of storage reads for each Get operation (Fig. 8d). Besides, we also measure the number of NVM reads per read to demonstrate the overhead of the index structure (Fig. 8c).

Fig. 8d indicates that MatrixKV requires the most number of storage reads (3.54 – 3.66) for searching a key since a



(a) Latency (b) Throughput (c) NVM reads (d) Storage reads

Fig. 8. Get (Random Read).

read may check multiple SSTs and each SST needs multiple storage I/O accesses. With the global B+Tree, SLM-DB needs the least number of storage reads (0.97 – 0.99) to search a key. Although PMDB using overlapping SSTs in each key range, it can also search a key with a close efficiency to SLM-DB (1.03 – 1.05) with the help of IFTrees.

Fig. 8c shows that MatrixKV has the least number of NVM reads (31 – 61 per key) since it does not have an index structure in NVM. Its NVM reads are only for checking SSTs in L_0. SLM-DB has the most NVM reads since it searches the B+ Tree in NVM for each key lookup. The size of B+ Tree becomes larger with more KV pairs inserted. Since PMDB indexes data blocks of 4KB size instead of each key, the size of IFTrees is less influenced by the total number of KV pairs. Compared to SLM-DB, PMDB reduces the total number of NVM reads by 29.03% – 74.24%.

Figs. 8a and 8b show that compared to MatrixKV, PMDB reduces the read latency by 61.54% – 51.81%. The throughput of PMDB is $2.61\times$ – $2.06\times$ higher than that of MatrixKV. PMDB achieves a comparable read performance to SLM-DB. When the number of KV pairs is large, e.g., with 128-byte *Value*, the read latency of PMDB can be close to that of SLM-DB since the index search in SLM-DB introduces a significant performance overhead. However, with 1KB *Value*, PMDB has a higher read latency by 11.13% than SLM-DB since the B+ Tree size in SLM-DB becomes smaller compared to that of 128-byte *Value*.

Range Queries. This experiment evaluates the performance of range queries. We set the *Value* size to 512 bytes since the real workloads are dominated by small *Values* [2], [4]. We pre-load KV stores with 50GB data and execute three types of operations including *Seek*, *Range8* and *Range64*. Seek *Seek* means only a seek operation is performed. Seek is to find the start point of a range query. In this case, bloom filters in IFTree will not help. *Range8* and *Range64* mean a short-range query of 8 keys and a longer range query of 64 keys respectively. Note that the start keys of range queries are randomly chosen/distributed. Therefore, seek-based compaction in PMDB will have very limited benefits.

This evaluation only shows the throughput (Fig. 9a). Besides, the performance of range queries is dominated by storage I/Os since a range query requires multiple storage I/Os. Therefore, we also show the number of storage I/Os for each range query in Fig. 9b. Fig. 9b indicates that SLM-DB achieves the highest efficiency for *Seek*. With the B+ Tree, SLM-DB can execute a *Seek* with only one storage I/O. MatrixKV and PMDB need multiple storage I/Os to construct iterators with overlapping SSTs. However, a single *Seek* is less used. It is typically followed by several *Nexts* in a range query.

Range8 in Fig. 9b shows the efficiency of KV stores for short-range queries. Since consecutive KV pairs may be in different SSTs, each *Next* of SLM-DB may lead to a random number of storage I/Os. After a *Seek*, MatrixKV and PMDB have built iterators with multiple data blocks. *Next* in a short-range may not need additional storage I/Os. However, for a longer range query (*Range64*), SLM-DB can have a closer efficiency with MatrixKV and PMDB since the previously read data blocks will be cached in memory. Some *Next* function can be satisfied by the cached data blocks.

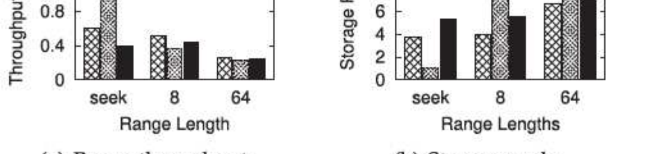
Fig. 9a shows that PMDB achieves a higher throughput by 16.82% for *Range8* than that of SLM-DB. For the longer range queries (*Range64*), PMDB achieves a close performance to that of MatrixKV. The throughput of *Range64* of PMDB is still 7.23% higher than that of SLM-DB.

5.2.2 Mixed Workloads

Real-world workloads are highly skewed and include mixed types of operations [2], [4]. Therefore, we further discuss KV store performance in skewed workloads with mixed operations as discussed in some existing studies [4], [36].

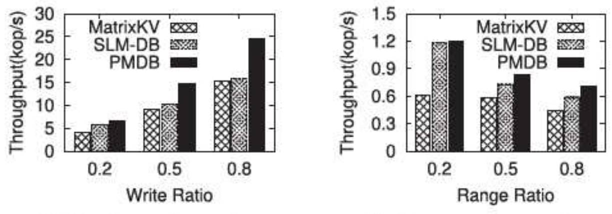
Similar to [36], we pre-load 50GB KV pairs randomly with a *Value* size of 512 bytes. Since PMDB is less efficient for short-range queries, we especially set the length of range queries to 8. Then we run mixed workloads with different ratios of the number of writes (Puts) and the number of reads (Gets), and short-range queries. The skewness of the workloads is defined in the same way as in [4], [36]. That is, it is the ratio of the accesses of writes, reads, and the start keys of range queries to a set of hottest keys. In our evaluation, we set the hottest key ratio and the workload skewness to 1% and 50% respectively as used in [36]. That is, 50% of total accesses are for the 1% of hottest keys.

First, we vary the write ratio to represent the write-dominated scenarios (Fig. 10a). We keep the same ratio of reads and range queries. For example, if the write ratio (i.e., the percentage of writes in total operations) is 0.2, both read ratio (the percentage of reads in total operations) and range



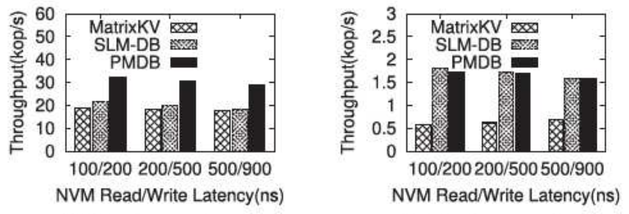
(a) Range throughput (b) Storage reads

Fig. 9. Range queries.



(a) Varying write ratios (b) Varying range ratios

Fig. 10. Throughputs of mixed workloads.



(a) Random write (b) Random read

Fig. 11. Write/Read performance on HDD.

ratio (the percentage of range queries in total operations) will be 0.4. Results in Fig. 10a indicate that PMDB outperforms the other two KV stores in all workloads. The throughput of PMDB is $1.16\times - 1.54\times$ and $1.51\times - 1.62\times$ higher than those of SLM-DB and MatrixKV respectively.

Range queries can influence the performance of PMDB significantly. Therefore, we further evaluate KV stores in workloads including only reads and range queries with different range ratios (the ratio of the number of range queries to the number of reads) (Fig. 10b). As shown in Fig. 10b, PMDB can improve the efficiency of range queries with seek-based compaction. When the range ratio is 0.2, the overall performance is dominated by reads, the throughput of PMDB is $1.96\times$ higher than that of MatrixKV, and similar to that of SLM-DB. When the range ratio is 0.8, the throughput of PMDB is $1.21\times$ and $1.61\times$ higher than that of SLM-DB and MatrixKV respectively.

5.2.3 Recovery and Space Overheads

Recovery. We measure the time to recover a KV store after loading 50GB KV pairs with 512-byte *Value*. The results indicate that all KV stores can achieve fast recovery. The time required to recover the KV store is less than one second.

Space Overhead. We measure the NVM and storage overheads after completing a random write workload with 50GB KV pairs with 512-byte *Value*. The storage space required for MatrixKV, SLM-DB and PMDB is 44.92GB, 45.31GB, and 45.22GB respectively. Since invalid key ratios can trigger compaction in both SLM-DB and PMDB, they do not introduce significant storage overhead than that of MatrixKV using leveled compaction.

The results of required NVM space indicate that MatrixKV introduces the least NVM space overhead (0.08GB) since it does not build indexes in NVM. Compared to SLM-DB (2.73GB) with indexes to every KV pair, PMDB (1.21GB) reduces the NVM space by 55.67% by constructing indexes based on data blocks. A data block can store more KV pairs if the *Value* size is small. When the *Value* size is 128 bytes, PMDB reduces the NVM space used for indexes by 88.31% compared to that of SLM-DB. The reduction of used NVM space can be more significant if we enable data compression or use larger blocks with sizes greater than 4KB.

5.2.4 Sensitivity Analysis

NVM Performance. There are multiple types of NVM with different performance [21], [29], [34]. Therefore, we vary NVM latency by changing the delay time in the write and read interfaces of the reserved memory. Fig. 11 shows the read and write throughput of three KV stores with different NVM latencies. The results indicate that the performance of PMDB is less sensitive on the NVM performance compared to SLM-DB. From 100/200ns to 500/900ns, the write and read throughout of PMDB is decreased by 9.53% and 5.98%. However, the write and read throughput of SLM-DB is decreased by 16.2% and 9.98%.

NVM Space. We decrease and increase the total available NVM space to 5% (5% NVM) and 20% (20% NVM) from the default 10% (10% NVM) of total size of data set respectively. Note that with 512-byte *value*, the B+ Tree of SLM-DB will need about 7.32% NVM to store B+ Tree. Similar to the previous evaluations, we set the write buffer size of SLM-DB the same as that of PMDB and assume SLM-DB has enough NVM to store B+ Tree. Then we run random writes (50GB with 512-byte *Value*) and random reads with different available NVM space. The results indicate that PMDB can still provide the best write performance with a comparable read performance to SLM-DB with different available NVM sizes. The write throughput (kop/s) of PMDB is $2.23\times$ and $2.56\times$ higher than that of MatrixKV and SLM-DB respectively.

5.2.5 Configuration Parameters

Max_io and Invalid_ratio. The max_io and invalid_ratio determine the frequency of compactions. They influence the trade-offs among write performance, read performance and storage space overhead. First of all, we keep the default invalid_ratio (0.3) and vary the max_io to 1, 10 and 20. When max_io = 1, PMDB will do read-merge-write compaction in storage. It achieves a close write and range performance to MaxtixKV and read performance to SLM-DB. When max_io = 10, the write throughput of PMDB is increased by $2.36\times$ compared to that of max_io = 1. The random read throughput is not significantly decreased with the help of bloom filters. The performance of short-range queries is decreased by 45.26% as we discussed. When max_io = 20, the write performance of PMDB is further increased by $1.7\times$ compared to that of max_io = 10. However, the read performance is decreased by 17.96%.

Next, we keep max_io = 10 and increases invalid_ratio from 0.3 to 0.5. The invalid_ratio has limited performance impact without enough number of updates. Therefore, we run random updates after loading KV pairs. After invalid_ratio is increased to 0.5 from 0.3, the write performance increases by 32.26%. However, the required storage space is increased by 19.22%.

Block Size. PMDB uses IFTrees to represent data blocks in an SST of a key range. We study the performance of random writes and random reads that influenced by varying block sizes among 4KB, 16KB and 64KB. The results indicate that a large block size will improve the write performance. With

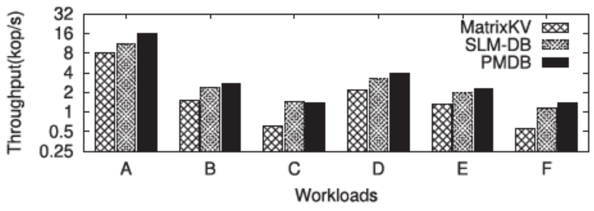


Fig. 12. Throughput of YCSB workloads.

larger blocks, the size of IFTrees becomes relatively smaller leaving more NVM space used by write buffers. Besides, these IFTrees will be updated less frequently during a compaction. When the block sizes are set to 16KB and 64KB, the write throughput of PMDB is increased by 15.37% and 42.11% respectively. However, if the block size is too large, the read performance will be decreased due to a large data transfer size for one storage read. The read throughput of PMDB is decreased by 14.78% after we increase the block size from 4KB to 64KB.

Partitions. We further run random writes without partitions. The whole available NVM space will be used as a single partition with one MemTable. The results indicate that the write throughput of PMDB without partitions is decreased by 21.45% compared to PMDB with partitions due to the frequently-triggered compactions.

5.3 Yahoo! Cloud Service Benchmark (YCSB)

Same as some of the prior work [21], [36], we further evaluate PMDB using core workloads from YCSB [7] including workloads A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads (latest values), 5% writes), E (95% ranges, 5%writes) and F (50% reads, 50% read-modify-writes). The total data size used in YCSB is 50GB. The key and value sizes are 16 bytes and 512 bytes respectively. Fig. 12 shows the throughputs of three KV stores in six core workloads. Note that the results are presented with a log scale to make them more readable. Although MaxtrixKV and SLM-DB achieves best performance for range queries and Gets respectively, PMDB ensures the best performance in workloads with mixed operations. In write-intensive workload A, the throughput of PMDB is 1.91× and 1.46× higher than those of MatrixKV and SLM-DB respectively. In read-intensive workloads (B, D, E, F), the throughput of PMDB is 1.27× − 2.49× and 1.15× − 1.21× higher than those of MatrixKV and SLM-DB respectively. In workload C (100% reads), the throughput of PMDB is comparable (2.95% smaller) to that of SLM-DB, and it is still 2.32× higher than that of MatrixKV.

6 Related Work

Write-Optimized Key-Value Stores. KV stores play significant roles in big data applications [8], [23], [37]. LSM-Tree based approach has been widely used in KV stores to serve write-intensive workloads [26], [32], [37]. Trade-offs among write, read and space amplifications in an LSM-Tree have been studied [8], [9] with compaction methods [31], [32], [39].

Key-Value Stores on NVM-Storage Systems. Several existing studies build KV stores for NVM-Storage hybrid systems. LSM-Tree based approaches including NoveLSM [22], NVMRocks [24], and MatrixKV [40] store a small portion of data in NVM and KV pairs in storage using a leveled structure. SLM-DB [21] stores KV pairs in storage with a single level structure and maintains a persistent B+ Tree [18] index in NVM for every KV pair. LightKV [16] has partitions on storage, but it still builds per-key indexes on NVM. Besides, it only uses size-tiered compaction in each partition which will lead to compacting a large amount of data involved in a single compaction.

7 Conclusion

In this article, we propose PMDB, a range-based KV store for hybrid NVM-Storage systems. PMDB can provide high performance for both write and read operations. We have compared PMDB to other state-of-the-art schemes designed for hybrid NVM-Storage systems. The results indicate that PMDB outperforms the other KV stores by 1.16× − 2.49 × . Besides, the performance of PMDB can be tuned for either write or read-intensive workloads.

References

[1] J. Arulraj and A. Pavlo, “How to build a non-volatile memory database management system,” in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1753–1758.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 53–64, 2012.

[3] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, “{SILK}: Preventing latency spikes in log-structured merge key-value stores,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 753–766.

[4] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, “Characterizing, modeling, and benchmarking rocksDB key-value workloads at facebook,” in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 209–223.

[5] A. Chen, “A review of emerging non-volatile memory (NVM) technologies and applications,” *Solid-Statist. Electron.*, vol. 125, pp. 25–38, 2016.

[6] H. Chenetal, “Efficient and available in-memory KV-store with hybrid erasure coding and replication,” *ACM Trans. Storage*, vol. 13, no. 3, pp. 1–30, 2017.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[8] N. Dayan, M. Athanassoulis, and S. Idreos, “Monkey: Optimal navigable key-value store,” in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 79–94.

[9] N. Dayan and S. Idreos, “Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging,” in *Proc. Int. Conf. Manage. Data*, 2018, pp. 505–520.

[10] G. DeCandia et al., “Dynamo: Amazon’s highly available key-value store,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, 2007.

[11] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, “Optimizing space amplification in RocksDB,” in *Proc. 8th Biennial Conf. Innov. Data Syst. Res.*, 2017, Art. no. 3.

[12] OpenGenus Foundation, “Interval trees: One step beyond BST,” 2020. [Online]. Available: https://iq.opengenus.org/interval-tree/

[13] S. Ghemawat and J. Dean, “LevelDB, A fast key-value storage library,” 2012. [Online]. Available: https://github.com/google/leveldb

[14] S. Ghemawat and J. Dean, “Sorted tables, LevelDB implementation,” 2012. [Online]. Available: https://github.com/google/leveldb/blob/master/doc/impl.md#sorted-tables

[15] E. Gilad et al., “Evendb: Optimizing key-value storage for spatial locality,” in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[16] S. Han, D. Jiang, and J. Xiong, “LightKV: A cross media key value store with persistent memory to cut long tail latency,” in *Proc. 36th Int. Conf. Massive Storage Syst. Technol.*, 2020.

[17] S. Hanke, “The performance of concurrent red-black tree algorithms,” in *Proc. Int. Workshop Algorithm Eng.*, 1999, pp. 286–300.

[18] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 187–200.

[19] Intel, "Transaction in PMDK libmemobj," 2015. [Online]. Available: https://pmem.io/2015/06/15/transactions.html

[20] Intel, "Tree map in PMDK," 2020. [Online]. Available https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/tree_map

[21] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 191–205.

[22] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMS for nonvolatile memory with novelsm," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 993–1005.

[23] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "{SLIK}: Scalable low-latency indexes for a key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 57–70.

[24] J. Li, A. Pavlo, and S. Dong, "NVMRocks: RocksDB on non-volatile memory systems," 2017.

[25] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proc. Linux Symp.*, 2007, pp. 21–33.

[26] F. Mei, Q. Cao, H. Jiang, and J. Li, "SifrDB: A unified solution for write-optimized key-value stores in large datacenter," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 477–489.

[27] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proc. 1st ACM SIGOPS Conf. Timely Results Oper. Syst.*, 2013, Art. no. 1.

[28] S. A. Noghabietal ., "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[29] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 371–386.

[30] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[31] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 497–514.

[32] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SlimDB: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endowment*, vol. 10, no. 13, pp. 2037–2048, 2017.

[33] A. Sainio, "NVDIMM: Changes are here so what's next," *Memory Comput. Summit*, 2016.

[34] R. Strenz, "Review and outlook on embedded NVM technologies–from evolution to revolution," in *Proc. IEEE Int. Memory Workshop*, 2020, pp. 1–4.

[35] Tutorialspoint, "Data structure and algorithms - AVL trees," 2020. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

[36] F. Wu, M.-H. Yang, B. Zhang, and D. H. C. Du, "AC-key: Adaptive caching for LSM-based key-value stores," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 603–615.

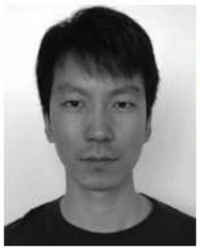
[37] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 71–82.

[38] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 167–181.

[39] T. Yao et al., "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.*, 2017.

[40] T. Yao et al., "MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 17–31.

[41] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for non-volatile main memories and disks," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 207–219.



Baoquan Zhang received the PhD degree in computer science from the University of Minnesota - Twin Cities advised by professor David H. C. Du. His research interests include memory/storage systems, such as key-value stores, RAID systems, non-volatile memory, etc.

Haoyu Gong received the BS degree in computer science from the Huazhong University of Science and Technology, in 2019. She is currently working toward the PhD degree in computer science with the University of Minnesota – Twin Cities advised by professor David Du. Her research interests include file and storage system designs for emerging storage technologies such as ZNS, storage class memories.

David H.C. Du (Fellow, IEEE) is currently the Qwest Chair Professor in computer science and engineering with the University of Minnesota - Twin Cities. His current research interests include focuses on intelligent and large storage systems. He serves on editorial boards of several international journals. He was a program director (IPA) with National Science Foundation (NSF) CISE/CNS Division from 2006 to 2008. He has served as Conference Chair, Program Committee Chair, and General Chair for several major conferences in database, security and parallel processing.