Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory

Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu, *Department of Computer Science and Technology, Tsinghua University and Beijing National Research Center for Information Science and Technology (BNRist)*

https://www.usenix.org/conference/atc23/presentation/wang-jing

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Revisiting Secondary Indexing in LSM-based Storage Systems with Persistent Memory

Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

LSM-based storage systems are widely used for superior write performance on block devices. However, they currently fail to efficiently support secondary indexing, since a secondary index query operation usually needs to retrieve multiple small values, which scatter in multiple LSM components. In this work, we revisit secondary indexing in LSM-based storage systems with byte-addressable persistent memory (PM). Existing PM-based indexes are not directly competent for efficient secondary indexing. We propose PERSEID, an efficient PM-based secondary indexing mechanism for LSM-based storage systems, which takes into account both characteristics of PM and secondary indexing. PERSEID consists of (1) a specifically designed secondary index structure that achieves high-performance insertion and query, (2) a lightweight hybrid PM-DRAM and hash-based validation approach to filter out obsolete values with subtle overhead, and (3) two adapted optimizations on primary table searching issued from secondary indexes to accelerate non-index-only queries. Our evaluation shows that PERSEID outperforms existing PM-based indexes by 3-7× and achieves about two orders of magnitude performance of state-of-the-art LSM-based secondary indexing techniques even if on PM instead of disks.

1 Introduction

Log-Structured Merge trees (LSM-trees) feature outstanding write performance and thus have been widely adopted in modern key-value (KV) stores, such as RocksDB [22] and Cassandra [1]. Different from in-place update storage structures (e.g., B^+-Tree), LSM-trees buffer writes in memory and flush them to storage devices in batches periodically to avoid random writes, which enables high write performance and low device write amplification. Besides high write performance, many database applications also require high-performance queries on not only primary keys but also other specific values [11], thus necessitating secondary indexing techniques.

LSM-trees' attributes make it challenging to design efficient secondary indexing. Modern LSM-based storage systems typically store a secondary index as another LSM-tree [47] (e.g., a column family in RocksDB [44]). However, designed for block devices and optimized for write performance, LSM-trees are not competent data structures for secondary indexes which require high search performance. First, since secondary indexes usually only store primary keys instead of full records[1] as values, KV pairs in secondary indexes are small. LSM-trees' heavy lookup operations are inefficient for these small KV pairs. Second, secondary keys are not unique and can have multiple associated primary keys. LSM-trees' out-of-place write pattern will scatter these non-consecutive-arrived values (i.e., associated primary keys) to multiple pieces at different levels. Consequently, query operations need to search all levels in the LSM-based secondary index to fetch these value pieces. Besides the device I/O overhead, LSM-trees have non-negligible overheads of CPU and memory (i.e., indexing and Bloom filter) [17, 20, 34].

Moreover, the consistency of secondary indexes is another issue in LSM-based storage systems. As an LSM-based primary table adopts the blind-write pattern to update or delete records (appends new data without checking old data, versus read-modify-write in B^+-Trees) for high write performance, it is unable to delete the obsolete entry in a secondary index without acquiring the old secondary key. Consequently, when querying a secondary index, the system should validate each entry by checking the primary table before returning the results to users, which introduces many unnecessary but expensive lookups on the primary table for obsolete entries. Some systems fetch old records when updating or deleting records to keep secondary indexes up-to-date synchronously [9, 44], whereas this method discards the blind-write attribute and thus degrades the write performance.

Though many efforts have been made to optimize these predicaments [36, 40, 47, 50], they are difficult to solve the problems discussed above well, sacrificing either write per-

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

[1]For clarity, we use *record* to refer to a KV pair in the primary table, and *entry* to refer to a KV pair in a secondary index.

formance of the LSM-based storage systems or query performance of the secondary index.

As secondary indexing demands low-latency queries and the KV pairs of secondary indexes are small, we argue that leveraging persistent memory (PM) to provide a new solution for secondary indexing is promising. PM has many attractive advantages such as byte-addressability, DRAM-comparable access latency, and the ability of data persistency, which is well suited to secondary indexing. Though there are many state-of-the-art PM-based indexes [13, 25, 31, 33, 37, 38, 45, 46, 61, 62], none of them are designed for secondary indexing. Without considering the non-unique feature of secondary indexes and consistency in LSM-based KV stores, simply adopting existing general PM-based indexes as secondary indexes can overshadow their performance.

In this work, we propose PERSEID, a new persistent memory-based secondary indexing mechanism for LSM-based KV stores. PERSEID contains PS-Tree, a specifically designed data structure on PM for secondary indexes. PS-Tree can leverage state-of-the-art PM-based indexes and enhance them with a specific value layer, which considers the characteristics of both PM and secondary indexing. The value layer of PS-Tree works in a manner of blended log-structured approach and B^+-Tree leaf nodes, which is both PM-friendly and secondary-index-friendly. Specifically, new values are appended to value pages for efficient insertion on PM. During the value page split, multiple values (i.e., associated primary keys) that belong to the same secondary key are reorganized to store continuously for efficient querying.

Moreover, PERSEID retains the blind-write attribute of LSM-based KV stores for high write performance without sacrificing secondary index query performance. This is achieved by a lightweight hybrid PM-DRAM and hash-based validation approach in PERSEID. PERSEID uses a hash table on PM to record the latest version of primary keys. However, multiple random accesses on PM still incur high latencies. Thus, PERSEID adopts a small mirror of the validation hash table on DRAM which only contains useful information for validation. During validation, the volatile hash table absorbs random accesses to PM, and thus reduces the validation overhead. The small volatile hash table not only saves DRAM memory space but also reduces cache pollution.

PERSEID has a fairly low latency of index-only query[2] However, the overhead of non-index-only queries is still dominated by the LSM-based primary table. Therefore, we further propose two optimizations for non-index-only queries in PERSEID. First, as querying the primary table issued by the secondary index is an internal operation, we can locate KV pairs with additional auxiliary information much more efficiently, reducing cumbersome indexing operations. By matching the tiering compaction strategy [19, 41], we can further bypass Bloom filter checking. Second, as one secondary index query may need to search for multiple independent records in the primary table, we parallelize these searching operations with multiple threads. Since search latencies on the LSM-based primary table may vary largely, we apply a worker-active manner on parallel threads to avoid load imbalance among threads and improve utilization.

We implement PERSEID and evaluate it against state-of-the-art PM-based indexes and LSM-based secondary indexing techniques on PM. The evaluation results show that PERSEID outperforms exiting PM-based indexes by 3-7$\times$ for queries, and achieves about two orders of magnitude higher performance of state-of-the-art LSM-based secondary indexing techniques even if on PM instead of disks, while maintaining the high write performance of LSM-based storage systems.

In summary, this paper makes the following contributions:

- Analysis of the inefficiencies of LSM-based secondary indexing techniques and existing PM-based indexes when adopted as secondary indexes for LSM-based KV stores.
- PERSEID, an efficient PM-based secondary indexing mechanism, which includes a secondary index-friendly structure, a lightweight validation approach, and two optimizations on primary table searching issued from secondary indexes.
- Experiments that demonstrate the advantage of PERSEID.

2 Background

2.1 Log-Structured Merge Trees

The LSM-tree applies out-of-place updates and performs sequential writes, which achieves superior write performance compared to other in-place-update storage structures.

The LSM-tree has a multi-level structure on storage and each level comprises one or several sorted runs. The size of Level L_n is several times (e.g., 10) larger than Level L_{n-1}. Each sorted run contains sorted KV pairs and is further partitioned to multiple small components called *SSTables*. In LSM-trees, new key-value pairs are first buffered into a memory component called a *MemTable*. When the MemTable fills up, it turns into an immutable MemTable and gets flushed to storage as a sorted run. Since sorted runs have overlapping key ranges, a query operation needs to search multiple sorted runs. To limit the number of sorted runs and improve search efficiency, LSM-trees conduct compaction periodically to merge several components and remove obsolete KV pairs.

Two typical compaction strategy and their variants are commonly used in LSM-trees [19, 41]: The leveling strategy [22, 24] only allows each level (besides L_0) to have only one sorted run; The tiering strategy [48, 54] allows each level (besides L_0) to have multiple sorted runs to reduce the write amplification. Compared with leveling strategy, tiering strategy has a much smaller write amplification ratio and thus

[2]Index-only query is a common query technique: Users create a *covering index* that contains specific columns required by queries to avoid the cost of reading the primary table [5, 7, 44]. A non-index-only query searches the secondary index by secondary key to get primary keys and then retrieves full records from the primary table.

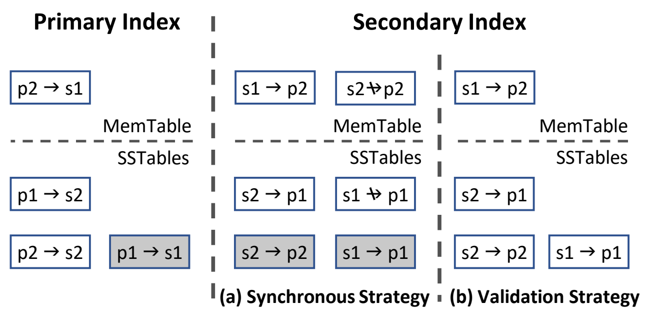


Figure 1: Stand-alone secondary indexing in LSM-based systems with Synchronous strategy and Validation strategy [47]. *The shaded entries indicate that they are invisible in the index.*

higher write performance. However, since query operations need to search multiple sorted runs in each level, LSM-trees with tiering strategy have much lower read performance.

2.2 Secondary Index in LSM-based Systems

Many applications require queries on specific values other than primary keys. Without an index based on specific values, database systems need to scan the whole table to find relevant data. Thus, secondary indexing is an indispensable technique in database systems. For example, in Facebook's database service for social graphs, secondary keys are heavily used, such as finding IDs who liked a specific photo [11, 44]. In this work, we mainly discuss stand-alone secondary indexes, which are separate index structures apart from the primary table and are commonly used in database systems [47]. A stand-alone secondary index maintains mappings from each secondary key to its associated primary keys. As secondary keys are not unique, a single secondary key can have multiple associated primary keys.

Consistency strategy. Since LSM-based KV stores update or delete records by out-of-place blind-writes, maintaining consistency of secondary indexes becomes a challenge in LSM-based storage systems. There are two strategies to handle this issue, *Synchronous* and *Validation*.

For *Synchronous* strategy, whenever a record is written in the primary table, the secondary index is maintained synchronously to reflect the latest and valid status (e.g., AsterixDB [9], MongoDB [4], MyRocks [44]). For example, as shown in Figure 1(a), when writing a new record { $p2 \rightarrow s1$ } (p denotes the primary key, s denotes the secondary key, and other fields are omitted for simplicity) into the primary table, the storage system also fetches the old record of $p2$ to get its old secondary key $s2$. Then the storage system inserts not only a new entry { $s1 \rightarrow p2$ } but also a tombstone to delete the obsolete entry { $s2 \rightarrow p2$ } in the secondary index. Nevertheless, this strategy discards the blind-write attribute and thus degrades the write performance which is the main advantage of LSM-based KV stores.

By contrast, as shown in Figure 1(b), *Validation* strategy only inserts the new entry { $s1 \rightarrow p2$ } but does not maintain the consistency of obsolete entries in secondary indexes (e.g., DELI [50], and secondary indexing proposed by Luo et al. [40]). However, secondary index query operations need to validate all relevant entries by checking the primary table to filter out obsolete mappings. Though previous work proposed some approaches to reduce the validation overhead, their benefits are limited. For example, DELI [50] lazily repairs the secondary index along with compaction of the primary table. Luo et al. [40] propose to store an extra timestamp for each entry in the secondary index and use a *primary key index* that only stores primary keys and their latest timestamp for validation. The primary key index is validated instead of the primary table. However, since the primary key index is also an LSM-tree, though it filters out unnecessary point lookups on the primary table, it still requires point lookups on itself.

Index type. As a secondary key can have multiple associated primary keys, LSM-based secondary indexes have two types surrounding this issue, including *composite index* and *posting list* [47]. The key in a *composite index* (i.e., *composite key*) is a concatenation of a secondary key and a primary key. The composite index is easy to implement and adopted by many applications [16, 44, 47]. However, it turns a secondary lookup operation into a prefix range search operation.

The *posting list* stores multiple associated primary keys in the value of a KV pair. When a new record is inserted, there are two update strategies. *Eager* update strategy conducts read-modify-write, fetching the old posting list and merging the new primary key to the posting list. *Lazy* update strategy blindly insert a new posting list which only includes the new primary key. It leaves posting lists merging to compaction. However, a secondary lookup needs to search all levels to fetch all relevant entries.

Limitations. Even though there are multiple strategies, types, and optimizations, LSM-based secondary indexes have to sacrifice either the write performance of storage systems or the secondary index query performance, which results from the incompatibility of inherent attributes of LSM-trees and characteristics of secondary indexes.

2.3 Persistent Memory

Persistent Memory (PM), also called Non-Volatile Memory (NVM) or Storage Class Memory (SCM), provides several attractive benefits for storage systems, such as byte-addressability, DRAM-comparable access latency, and data persistency. CPUs can access data on PM directly with load and store instructions. Compared to DRAM, PM has a much larger capacity and lower cost and power consumption. In addition to DDR bus-connected PM (e.g., Intel Optane DCPMM), the recent high-bandwidth and low-latency IO interconnection, Compute Express Link (CXL) [3, 29], brings a new form of SCM, CXL device-attached memory (e.g., Samsung's Memory-semantic SSD [8]).

However, PM also has some performance idiosyncrasies.

For example, the current commercial PM hardware (i.e., Intel Optane DCPMM) has physical media access granularity of 256 bytes, leading to high random access latency (about 3× of DRAM) and write amplification for small random writes, which needs to be considered when designing PM systems [14, 51, 53, 55, 57, 60]. These idiosyncrasies should be more obvious on CXL device-attached memory due to the physical media characteristics (e.g., flash page in CXL-SSD).

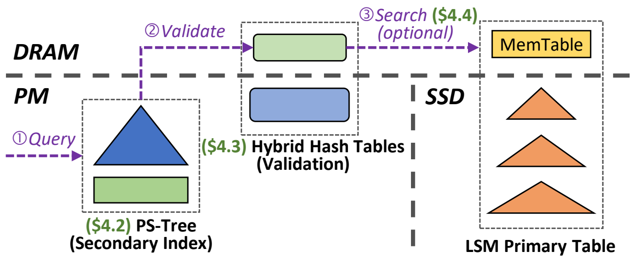
3 Motivation

Though recent work introduces some techniques to optimize secondary indexing in LSM-based systems, we find that *the performance of LSM-based secondary indexing is still unsatisfactory due to the incompatibility of inherent attributes of LSM-trees and characteristics of secondary indexing*. On the one hand, LSM-tree is not a competent data structure for secondary indexes, since the characteristics of secondary indexes exacerbate the deficiency of LSM-tree's read operations: (1) KV pairs are usually small in secondary indexes, to which LSM-tree's cumbersome lookup operations are unfriendly; (2) Secondary keys are not unique and can have multiple values, which LSM-tree's out-of-place update will exacerbate the query inefficiency. On the other hand, the blind-write attribute of LSM-based primary tables makes the consistency of secondary indexes troublesome.

Therefore, this motivates us to find a better solution for secondary indexes in LSM-based storage systems. As PM provides attractive features such as byte-addressability, DRAM-comparable access latency, and data persistency, we argue that it is promising to provide secondary indexing with PM.

Though there are many state-of-the-art PM-based index structures, they are not specifically designed for secondary indexes. To adopt them as secondary indexes (e.g., support the multi-value feature), naive approaches include the composite index or using a conventional allocator to organize posting lists (§2.2). However, *simply adopting these naive approaches to use existing PM-based indexes as secondary indexes will overshadow their superior advantages*.

Why not use a PM-based composite index? Though this method is straightforward and easy to implement in LSM-based systems, it is not ideal for tree-based persistent indexes. First, when adding or removing a primary key for a secondary key, a value update operation turns into a new composite key insert or delete operation for composite indexes. Insert and delete operations are more expensive than update operations in a PM-based tree index because they may cause shift operations or structural modification operations (SMO). Second, composite indexes store every pair of mappings as an individual KV pair, expanding the number of KV pairs, which increases the height of the tree index and thus degrades its query performance. Third, storing the same secondary keys repeatedly in multiple composite keys wastes PM space, which can be a dominant overhead for some real-world databases [59].



DRAM
PM
SSD
①Query
②Validate
③Search (§4.4) (optional)
MemTable
(§4.2) PS-Tree (Secondary Index)
(§4.3) Hybrid Hash Tables (Validation)
LSM Primary Table

Figure 2: The overall architecture with Perseid.

Why not use a conventional allocator? One may use a conventional allocator, such as a log-structured approach or a slab-based allocator, to allocate space for values out of the indexes. Nevertheless, they are not suitable for values of secondary keys. The log-structured approach is friendly to PM for its sequential-write pattern. One may use the log-structured approach to append new values in the log and use pointers to link associated values belonging to the same secondary key. However, it will scatter values (primary keys) associated with the same secondary key and thus reduce the query performance due to poor data locality. Slab-based allocators are widely used for volatile memory, but they are not suitable for PM and secondary indexes. First, these general-purpose allocators usually have high overheads on PM since they conduct expensive mechanisms for crash consistency (e.g., logging) and perform many small writes on their metadata which is necessary for recovery [6]. Second, slab-based allocators have low memory utilization due to the memory fragmentation issue [49], which cannot be eliminated by restarting on PM [18]. These issues are more severe for secondary indexes. In secondary indexes, the value of a secondary key is changed by inserting or removing primary keys, which means the value size (the total size of associated primary keys) changes constantly. This characteristic not only needs frequent reallocation but also exacerbates the fragmentation issue.

Our experiments (§5.2) show that these naive approaches on PM-based indexes lead to several times performance degradation. It thus motivates us to explore a new PM-based secondary indexing mechanism for LSM-based KV stores. In addition, an efficient validation approach is required to retain the blind-write attribute of LSM-based KV stores.

4 Perseid Design

4.1 Overview

Motivated by the analysis above, we propose PERSEID, a PM-based secondary indexing mechanism for LSM-based storage systems, which overcomes traditional LSM-based secondary indexes' deficiencies. Figure 2 shows the overall architecture of an LSM-based storage system with PERSEID.

- PERSEID contains a PM-based secondary index, `PS-Tree`, which is both PM and secondary index friendly: by adopting log-structured insertion, PS-Tree achieves fast insertion on PM; by storing primary keys which associate to the same secondary key closer and further rearranging them adjacent, PS-Tree supports efficient query operations (§4.2).

- PERSEID retains the blind-write attribute of the LSM primary table for write performance without sacrificing query performance by introducing a lightweight hybrid PM-DRAM and hash-based validation approach. The validation approach contains a persistent hash table to record version information of primary keys, and a volatile and lite hash table to absorb random accesses to PM. (§4.3).
- To accelerate non-index-only queries, PERSEID adapts two optimizations on primary table searching issued from secondary indexes. PERSEID filters out irrelevant component searching with sequence numbers and parallelizes primary table searching in an efficient way (§4.4).

4.2 PS-Tree Design

PERSEID introduces PS-Tree, a PM-based secondary index, which is designed considering the multi-value feature and PM characteristics. We first present PS-Tree's structure (§4.2.1), and then describe its operations (§4.2.2).

4.2.1 Structure

The overall structure of PS-Tree is shown in Figure 3. PS-Tree consists of two layers, *SKey Layer* for indexing secondary keys and *PKey Layer* for storing values. Specifically, the SKey Layer resembles a normal in-memory index and can leverage an existing high-performance PM-based index (e.g., P-Masstree [33, 43] and FAST&FAIR [25]). The PKey Layer stores variable-number values (i.e., primary keys and other user-specified values) of secondary keys in a manner of blended B^+-Tree leaf nodes and log-structured approaches, which combines the advantages of the two approaches. The value of a secondary key in the SKey Layer is a pointer, which points to corresponding primary keys in the PKey Layer. Each pointer is a combination of the address of the PKey Page and an offset within the page.

In the PKey Layer, primary key entries (*PKey Entries*) are stored in *PKey Pages*. Each PKey entry has an 8-byte metadata header and a primary key. The header consists of a 2-byte size, a 1-bit obsolete flag, and a 47-bit sequence number (SQN) of the primary key. The SQN is internally used for multi-version concurrency control (MVCC) in LSM-based KV stores [22, 24]. Each new record (including updates and deletes) in the primary table gets a monotonically increased SQN. PERSEID leverages the SQN mechanism to guarantee data consistency among the primary table and secondary indexes, and also for validation which will be described in §4.3. PKey Pages are aligned to PM physical media access granularity (e.g., 256 bytes of Intel Optane DCPMM [57]). PS-Tree inserts PKey Entries into PKey Pages in a log-structured manner to reduce the write overhead and ease crash consistency on PM.

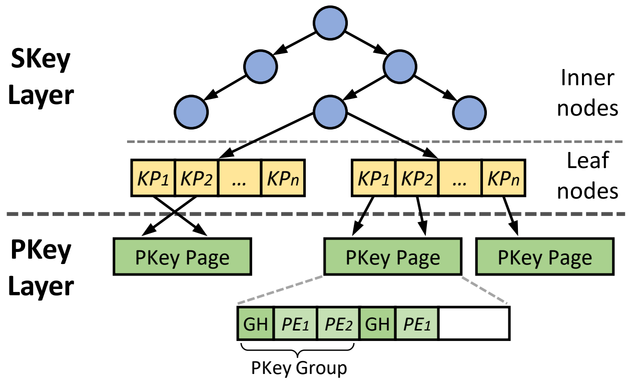


Figure 3: The structure of PS-Tree. *KP: Key Pointer pair, GH: Group Header, and PE: PKey Entry.*

Nevertheless, traditional log-structured approaches scatter different values of the same secondary key in the log, resulting in poor data locality and degraded query performance. To improve data locality, PS-Tree stores PKey Entries of contiguous SKeys in the same PKey Page, similar to the leaf node in a B^+-Tree. Furthermore, during the PKey Page split, PS-Tree rearranges PKey Entries that belong to the same secondary keys to store continuously as a *PKey Group*. Each PKey Group has an 8-byte *Group Header* and one or multiple PKey Entries. The lower 48 bits of a group header are the address of the previous PKey Group of the same secondary key or null if the current group is the last one. Thus all PKey Groups belonging to one secondary key are linked as a list. The remaining 16 bits store the number of total entries and the number of obsolete entries in the group.

4.2.2 Basic Operations

PS-Tree considers features of both secondary indexing and persistent memory. Compared with DRAM, PM has limited write bandwidth and the write amplification issue. Therefore, PS-Tree adopts log-structured insertion and copy-on-write split for efficient writes and lightweight crash consistency mechanisms. To avoid high latencies of multiple random accesses of multiple values on PM during query operations, PS-Tree reorganizes values of the secondary index and conducts lazy garbage collection during the PKey Page split.

Log-structured Insert. Algorithm 1 describes the process of the insert operation. First, PS-Tree searches for the SKey and its pointer in the SKey Layer. From the pointer, PS-Tree locates the previous PKey Group and the corresponding PKey Page (Line 1-3). If the SKey is not found, then the PKey Page is located from the pointer of the previous SKey which is just smaller than this new SKey (Line 5).

Second, PS-Tree appends a new PKey Group in that PKey Page (Line 11-12). The new PKey Group contains one entry with the new PKey and other values if specified, and the header points to the previous PKey Group if exists.

Third, the new pointer of the SKey (i.e., the address of

Algorithm 1: Insert(SKey sk, PKey pk, Slice val, SeqNumber seq)

```
search for the leaf_node and pointer ptr of sk in SKey Layer;
if ptr ≠ NULL then // found sk
    pkey_page = pointer.pkey_page;
else
    pkey_page = leaf_node→get_prev_pkey_page(sk);
end
if pkey_page is full then
    pkey_page split;
    goto Line 1;
end
construct a PKeyGroup pg with pk, val, seq, and ptr;
new_ptr = pkey_page→append(pg);
leaf_node→upsert(sk, new_ptr);
```

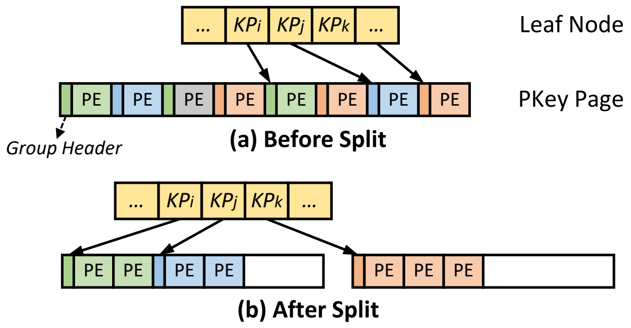


Figure 4: An example of PKey Page split. *PEs (PKey Entries) with the same color belong to the same secondary key; PE in gray are obsolete.*

the new PKey Group) is updated or inserted in the SKey Layer (Line 13). Thus, the insert request usually performs an update operation in the SKey Layer. PKey Entries of a secondary key are always linked in the order of recency to facilitate query operations which usually require the most latest entries [11,47].

Search. The search operation in PS-Tree starts with searching for the secondary key and its pointer in the SKey Layer. Then, from the latest PKey Group indicated by the pointer, primary keys and other user-specified values can be retrieved in the order of recency. PERSEID adopts the *Validation* strategy (§2.2) for its high ingestion performance. Therefore, all primary keys are first validated before returning. Obsolete PKey Entries are marked as obsolete by setting their obsolete flags, which will be removed physically when the PKey Page splits. The LSM-based primary table supports MVCC by attaching one snapshot and using reference counters to protect components from being deleted [22, 24]. In PS-Tree, we adopt an epoch-based approach: readers publish their snapshot numbers during query operations, and obsolete entries whose sequence number is larger than any reader's snapshot number are guaranteed not to be removed physically.

Update and Delete. PS-Tree has no update or delete operations (from the point of view of secondary indexes rather than the data structure). Since updating the primary key of a record in the primary table is commonly not supported in database systems, there is no need to update values (primary keys) in secondary indexes. With the Validation strategy, PS-Tree does not delete the obsolete entries synchronously with the primary table. PS-Tree leaves obsolete entry cleaning to garbage collection.

Locality-aware PKey Page Split with Garbage Collection. When a PKey Page does not have enough space for a new entry, it splits into two new PKey Page in a copy-on-write manner. Since insertions are performed in a log-structured manner, the PKey Entries which belong to one SKey may scatter discontinuously. Querying these entries may need multiple random accesses on PM. As PM has non-negligible read latencies compared to DRAM (e.g., about 300 ns with Intel Optane DCPMM [57]), query operations can have high overheads. Therefore, as shown in Figure 4, to improve locality, PS-Tree reorganizes PKey Entries when the PKey Page splits. PS-Tree rearranges PKey Entries belonging to the same SKey in one PKey Group, so these entries are stored continuously, and the storage overhead of the Group Header is reduced since multiple PKey Entries share one Group Header.

Besides, entries not marked as obsolete in the current PKey Page are validated by a lightweight approach (described in §4.3) and obsolete entries are physically removed during reorganization to reduce space overhead. Since a secondary key may have many primary keys which occupy more than one PKey Page, for those PKey Entries not in the current PKey Page, PS-Tree lazily garbage collects them only when the number of obsolete entries exceeds half of the number of total entries in that PKey Group. To support MVCC, PS-Tree retains obsolete entries whose sequence number is larger than the minimum snapshot number of concurrent readers. Obsolete entries may retain long if there exists a long-running queries. PERSEID can be enhanced with similar techniques in recent work [30, 35] to handle long-lived snapshots. After rearranging valid entries to the new PKey Pages, pointers of related SKeys are updated and the old PKey Page is freed.

Crash Consistency. PERSEID relies on the existing write-ahead-log (WAL) of the LSM-based primary table to guarantee atomic durability among the primary table and secondary indexes. During recovery with WAL, PERSEID redoes uncompleted operation to the PS-Tree.

PS-Tree also handles its own crash consistency issues. Insert operations are committed only when the pointers in SKey Layer are updated. If the system crashes before updating pointers but after allocating a new PKey Page, then the PKey Page is unreachable. After restart, a background thread will scan the allocated pages and PS-Tree to find and reclaim unreachable pages. Besides, PS-Tree allows concurrent insertion in one PKey Page. A thread obtains the space to write by compare-and-swap the tail pointer of the PKey Page. Thus, the space may leak if any thread obtains it but does not update the pointer in SKey Layer when the system crashes. `PS-Tree` tolerates this situation and leaves these leakages to page splitting which naturally reclaims these leaked spaces.

4.3 Hybrid PM-DRAM Validation

PERSEID adopts the Validation strategy (see §2.2) for high write performance, which necessitates a lightweight validation approach. Since update-intensive workloads are quite common nowadays [11, 12], if the validation approach is heavy, validating a large number of obsolete entries brings no outcomes but generates huge overhead.

PERSEID introduces a lightweight validation approach based on the requirement of validation. PERSEID adopts a hash table on PM storing version information for primary keys. The hash table is indexed by the primary key and stores its latest sequence number (§4.2.1). Nevertheless, even though point lookups on a PM-based hash table are much faster than on a tree, the validation time is comparable to the query time of `PS-Tree`. This is because one secondary key has multiple primary keys to validate, and PM has non-negligible random access latency. Simply placing the hash table on DRAM will occupy a large memory footprint. However, as validation only needs to validate whether a version of a primary key is valid, but not obtaining the specific latest version number, PERSEID builds another volatile hash table on DRAM which only stores versions for primary keys that have been updated or deleted. In this way, PERSEID only needs to query the small volatile hash table and thus the validation overhead is further reduced.

Figure 5 illustrates the hybrid PM-DRAM validation approach. The values in the hash tables consist of the sequence number of the record (6-byte) and a 2-byte counter. The counter is used to determine whether a primary key has obsolete versions. There is a slight difference in the counters of the two hash tables. In the volatile hash table, each counter indicates the number of *logically* existing entries related to a primary key in the secondary index. By contrast, each counter in the persistent hash table indicates the number of *physically* existing entries in the secondary index. Next, we describe the validation approach in detail according to operations.

Insert. When a new record (including update and delete) is inserted into the primary table, the primary key is inserted or updated with its sequence number into the persistent hash table. If the persistent hash table does not contain this primary key before, its counter is set to one, which means this primary key has only one version and no obsolete entries of this primary key exist in the secondary index. For example in Figure 5, at *t2*, key *c* is inserted for the first time, and it is inserted into the persistent hash table. Otherwise, the primary key's counter in the persistent hash table is increased by one; besides, the primary key is inserted or updated with its sequence number into the volatile hash table, and the counter

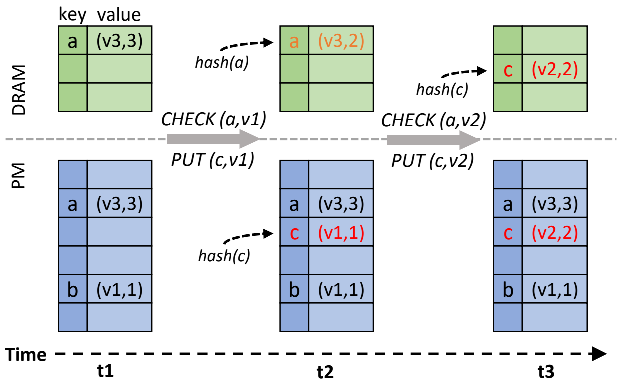


Figure 5: Hybrid PM-DRAM hash-based validation.

in the volatile hash table is set to two if it's an insertion or increased by one if it's an update. For example, when key *c* is updated with a new version *v2* at *t3* in Figure 5, the entry in the persistent hash table is updated and a new entry is inserted into the volatile hash table.

Validate. The secondary index validates an entry by querying the volatile hash table. Specifically, the entry is valid if the sequence number of this entry matches the latest sequence number stored in the hash table, or the hash table does not contain the primary key which means there are no obsolete entries of this primary key. Otherwise, the entry is not valid and PERSEID marks the entry as obsolete and decreases the counter of the entry in the volatile hash table by one. For example, when key *a* is checked with an obsolete version *v1* at *t2* in Figure 5, the result is false, and then the counter is decreased from 3 to 2. If the counter is decreased to 1, which means all obsolete entries have been marked, the entry is removed from the volatile hash table to restrict the hash table size. For example, when key *a* is checked with an obsolete version *v2* at *t3* in Figure 5, the counter is decreased to one, the validation return false and the entry is removed. A rare case is that the volatile hash table reports a new sequence number larger than the current reader's snapshot number, which means a concurrent writer has updated this primary key. In this case, we cannot directly confirm whether this entry is valid in this snapshot, so we have to validate it by the primary table. During validation for secondary index queries, PERSEID only operate with the volatile validation hash table. Thus, the validation overhead is quite small.

Garbage Collection. During the PKey Page split, entries that are not marked as obsolete are also validated to remove obsolete entries (§4.2.2). Since this step physically removes obsolete entries, PERSEID decreases the corresponding counters in the persistent hash table. If a counter is decreased to one, PERSEID removes the corresponding hash pairs from the volatile hash table.

Recovery. When the system restarts from a crash or a normal shutdown, the volatile hash table needs to be recovered. PERSEID iterates the whole persistent hash table and inserts primary keys whose counter is greater than one into the volatile hash table. Now the counters in the volatile hash table are numbers of physically existing entries, which may be larger than the actual numbers of logically existing entries. Therefore, some false positive primary keys may exist in the volatile hash table. However, this does not affect the validation accuracy and these primary keys can be removed by garbage collection.

4.4 Non-Index-Only Query Optimizations

Though the PERSEID significantly reduces the overhead of secondary indexing, the overhead of non-index-only queries (require full records) is still dominated by the LSM-based primary table. Thus, PERSEID further introduces two optimizations for non-index-only queries.

4.4.1 Locating Components with Sequence Number

A secondary index query operation may need to search the primary LSM table multiple times for all its associated records. LSM-trees have mediocre read performance due to the multi-level structure. Besides device I/Os, if data is cached in memory or using fast storage devices, LSM-trees have non-negligible overheads on probing components (i.e., indexing and checking Bloom filters) [17, 20, 60]. Moreover, the read performance gets worse with the tiering compaction strategy since more components (SSTables) need to check and read.

Nevertheless, we find that many components are unnecessary to probe in searching processes issued from the secondary index. Previous work uses zone maps, which store the minimum and maximum values of an attribute, to skip irrelevant data blocks or components during searching [9, 10, 47]. We found that this technique can also be used by secondary indexes to search the primary table. Since we have already recorded the sequence numbers of primary keys in the secondary index, the sequence number can be used as an additional attribute to skip irrelevant components. PERSEID builds a zone map that records a sequence number range (i.e., the minimum and maximum sequence numbers of records) for each component (including MemTables).

Moreover, since tiering compaction does not rewrite SSTables in the higher level (except for the last level), for a range partition, the sequence number ranges of different levels and even different sorted runs are strictly divided. For primary tables adopting the tiering strategy, with the primary key to search SSTables horizontally and the additional sequence number to search sorted runs vertically, PERSEID can locate the exact component that contains the record directly. Besides, since PERSEID already validates the version so it must exist in the component, PERSEID can further skip the Bloom filter checking. Thus, the overheads on indexing and checking Bloom filters are almost eliminated.

This optimization does not fit with leveling strategy perfectly. The sequence number ranges in different levels have overlaps because compaction rewrites SSTables in the higher level and generates new SSTables with blended sequence numbers from multiple levels. However, since most LSM-base KV stores adopt the tiering strategy on L_0 at least [22, 24], this optimization is still effective to some extent.

4.4.2 Parallel Primary Table Searching

A single secondary key usually has multiple associated primary keys, and queries on these primary keys are independent. Therefore, using multiple threads to accelerate primary table searching is a natural optimization method. One naive approach is to assign primary keys to each thread in a round-robin fashion. However, point lookups on LSM-trees may have a large latency gap, since some KV pairs can be fetched from MemTable or block cache directly and others may reside at a relatively high level and need several disk I/Os due to Bloom filter false positives. The naive approach will result in a load imbalance among parallel threads where some threads have finished their tasks and become idle while others are still stuck and there may still exist some unfinished tasks.

To avoid this situation, we apply a *worker-active* fashion. PERSEID publishes primary keys into a shared queue as tasks, and each parallel thread fetches one task from the queue. In this way, though each thread may complete a different number of tasks, parallel threads are fully utilized.

5 Evaluation

In this section, we evaluate PERSEID against existing PM-based indexes with naive approaches and state-of-the-art LSM-based secondary indexing techniques [40, 47]. After describing the experimental setup (§5.1), we evaluate these secondary indexing mechanisms with micro benchmarks to show their performance on basic operations (§5.2). Then, we evaluate these systems' overall performance with mixed workloads (§5.3) and recovery time (§5.4).

5.1 Experimental Setup

Platform. Our experiments are conducted on a server with an 18-core Intel Xeon Gold 5220 CPU, which runs Ubuntu 20.04 LTS with Linux 5.4. The system is equipped with 64 GB DRAM, two 128 GB Intel Optane DC Persistent Memory in AppDirect mode, and a 480 GB Intel Optane 905P SSD.
Compared Systems. We implement `PS-Tree` of PERSEID based on two typical PM-based indexes, a B^+-Tree FAST&FAIR [25] and a trie-like P-Masstree [33, 43]. We compare PERSEID against the two original PM-based indexes, and LSM-based secondary index (denoted as `LSMSI`) approaches of *LevelDB++* [47]. The compared PM-based indexes are implemented as secondary indexes via the composite index approach and the log-structured approach (denoted as `FAST&FAIR-composite`, `FAST&FAIR-log`, `P-Masstree-composite`, `P-Masstree-log`, respectively). For the log-structured approach, we provide enough space

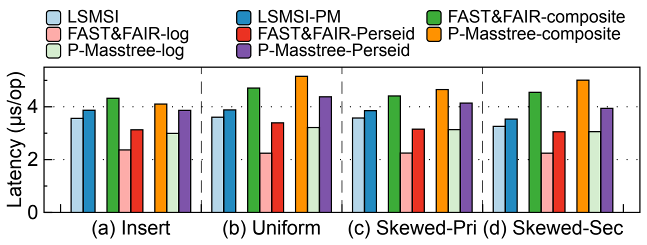


Figure 6: Insert and upsert performance.

and disable garbage collection to avoid its influence and present the ideal performance [51]. We enhance other PM-based indexes with PERSEID's hybrid PM-DRAM validation approach (§4.3) and `LSMSI` with the primary key index [40] (§2.2) for validation. For a fair comparison, we also implement LSM-based secondary indexing approaches on PM. We use *PebblesDB* [48], a state-of-the-art tiering-based KV store, as the primary table.

Workloads. Since common benchmarks for key-value stores such as YCSB [15] don't have operations on secondary indexes, as in previous work [36, 40, 47], we implemented a secondary index workload generator based on an open-source twitter-like workload generator [2] for evaluation. With this generator, we generate several microbenchmark workloads and mixed workloads. The primary key (e.g., ID) and secondary key (e.g., UserID) are randomly generated 64-bit integers. The key space of primary keys and secondary keys is 100 million and 4 million, respectively. Thus the average number of records per secondary key is about 25. The size of each record is 1KB.

KV Store Configurations. For the primary table, according to configuration tuning guide [23], MemTable size is set to 64 MB and the Bloom filters are set to 10 bits per key. As our workloads will generate a primary table larger than 100 GB, we set a 16-GB block cache for the primary table and a 1-GB block cache for the LSM-based secondary index. Compression is turned off to reduce other influencing factors.

5.2 Microbenchmarks

In this section, we evaluate the basic single-threaded performance and scalability of compared secondary indexing mechanisms.

5.2.1 Insert and Update

The *Insert* workload (i.e., no updates) has 100 million unique records. Figure 6(a) shows the average latency of insert operations of each secondary index. PERSEID performs about 10-38% faster than the corresponding composite indexes, but 25% slower than the *ideal* log-structured approach without garbage collection due to the page split overhead in `PS-Tree`. The composite index approach results in inferior performance as we analyzed in §3. Other approaches have higher performance due to the sequential-write pattern.

The upsert workloads contain 100 million insert operations and 100 million update operations. Operations are shuffled to avoid all newer entries being valid in secondary indexes. In the *Uniform* workload, both primary keys and secondary keys follow a uniform distribution. In the *Skewed-Pri* workload, primary keys follow a Zipfian distribution with the skewness parameter 0.99, and secondary keys are selected randomly. In the *Skewed-Sec* workload, secondary keys follow a Zipfian distribution (parameter 0.99), and primary keys are uniform. Thus, hot secondary keys have lots of associated primary keys, which represent low-cardinality columns.

Compared with other secondary indexes, composite indexes perform even worse in upsert workloads. This is because, with additional upsert operations, composite indexes have more KV pairs and larger tree heights. By contrast, `PS-Tree` and the log-structured approach do not increase the number of KV pairs in the index part.

5.2.2 Query

In this experiment, we evaluate the performance of index-only queries of each system after loading the insert workload or upsert workloads. Index-only query reflects the performance of a secondary index itself and is a common query technique (i.e., covering index [5, 7]) to avoid looking up the primary table. We show two different selectivities by specifying limit *N* (10 and 200) on return results. The most recent and valid *N* entries are returned. For limit of 200, the actual average number of returned entries per query is 25 and 142 for the *Skewed-Pri* and *Skewed-Sec*, respectively.

Figure 7 shows the results of index-only query performance. From the results, we have the following observations.

First, PM-based indexes have significantly lower latencies than LSM-based secondary indexes. Putting `LSMSI` on PM (`LSMSI-PM`) has very limited improvement, which is because `LSMSI` already benefits from block cache and OS page cache. Even so, `LSMSI` is still inefficient due to the high overhead of indexing and Bloom filter checking. Besides, `LSMSI` has a high overhead on validating the primary key index.

Second, PERSEID outperforms existing PM-based indexes with the composite index and the log-structured approach by up to 4.5× and 4.3×, respectively. The log-structured approach has poor locality since relevant values are scattered across the whole log and require multiple random accesses to fetch them all. Composite indexes are inferior due to the larger number of KV pairs in the indexes and range-scan operations as we analyzed in §3. They are especially inefficient under the *Skewed-Sec* workload with a large limit (e.g., 200), where they fetch a large number of entries and fail to enjoy the cache effect. By contrast, the performance of PERSEID is much more stable across different workloads, owing to the locality-aware design of `PS-Tree`. For a limit of 10, PM-based secondary indexes benefit from higher cache hit ratios under the *Skewed-Sec* workload, thus achieving better performance than other upsert workloads. Composite indexes also occupy about 4×

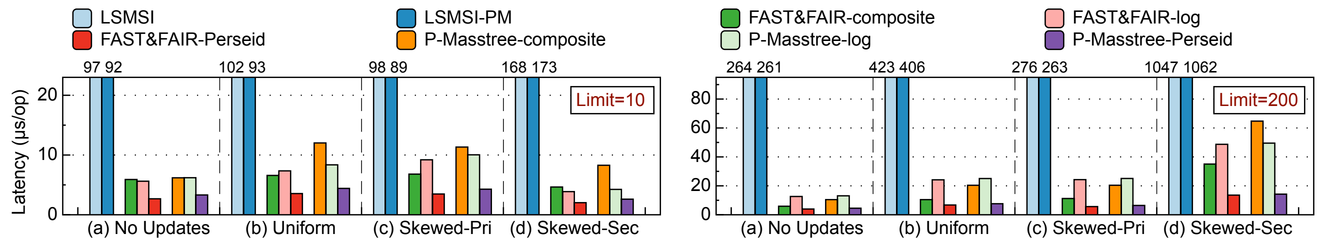


Figure 7: Index-only query performance.

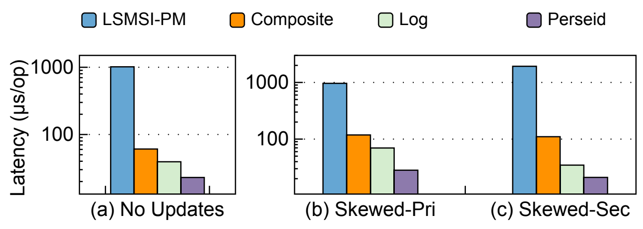


Figure 8: Index-only range query performance.

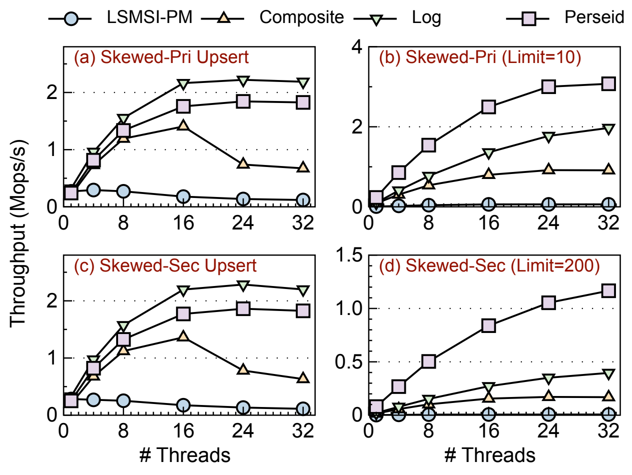


Figure 9: Multi-threaded performance.

more PM space than `PS-Tree`, which is because they repeatedly store secondary keys and have more index nodes. In addition, `P-Masstree-composite` has higher latencies than `FAST&FAIR-composite`, because trie-based indexes are less efficient than B$^+$-Trees in range search since their leaf nodes do not have sibling pointers pointing to neighbor nodes.

Third, under upsert workloads, all systems need to validate more primary keys to exclude obsolete entries, which also contributes to the higher overheads than under insert workloads. For `LSMSI`, since the primary key index needs multiple heavy point lookups on LSM-trees, validating the primary key index accounts for the lion's share of the total cost of an index-only query. `LSMSI` has lower latencies under the *Skewed-Pri* workload than other upsert workloads since the primary key index enjoys the data locality on primary keys. By contrast, PERSEID (and other PM-based indexes) validates on a volatile hash table, which takes up less than half of the total cost. The overhead on PERSEID increased little due to the locality-aware design of `PS-Tree` and the lightweight validation approach.

5.2.3 Range Query

In the following experiments, we show results of the LSM-based secondary index on PM (`LSMSI-PM`), and PM-based secondary indexes based on P-Masstree as representatives. We evaluate the range queries of these secondary indexes. Each range query searches for 20 secondary keys and retrieves 5 latest associated primary keys of each secondary key.

The results are shown in Figure 8. Range queries need to search more KV pairs from ten different secondary keys, showing a more pronounced difference between these secondary indexes than low-limit query operations. PERSEID outperforms `LSMSI-PM`, the Composite P-Masstree, and the log-structured approach by up to 92×, 5.2×, and 1.6×, respectively under the *Skewed-Sec* workload. Though `LSMSI-PM` benefits from PM access latency and DRAM caching, it still has a fairly high latency. This is because the range operation in LSM needs to merge-sort multiple iterators of components. The composite index needs to perform more range search than PERSEID in the index since PERSEID groups primary keys outside of the index.

5.2.4 Multi-threaded Performance

Figure 9 shows the multi-threaded performance of compared secondary indexes. We take the results of *Skewed-Pri* and *Skewed-Sec* workloads as representatives. For *Skewed-Sec*, we show the result with the limit of 200, and the result with the limit of 10 is similar to that of *Skewed-Pri*. For upsert operations, PERSEID scales up to 24 threads, achieving 2.8× and 16× the upsert throughput of the composite P-Masstree and `LSMSI-PM`, and slightly slower than the ideal log-structured approach. For query operations, PERSEID scales well and achieves 7× and 3× query throughput of `P-Masstree-composite` and `P-Masstree-log` under the *Skewed-Sec* workload due to the locality-aware design of `PS-Tree`. `LSMSI` has poor scalability due to its coarse-granularity lock and non-concurrent logging mechanism. Though using the same index structure (P-Masstree), because

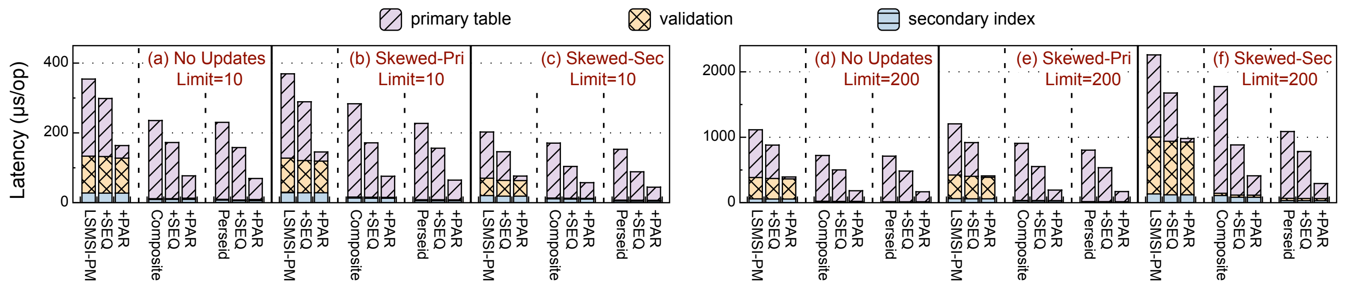


Figure 10: Non-index-only query performance. *The primary table time on +PAR only shows the time not covered by other parts.*

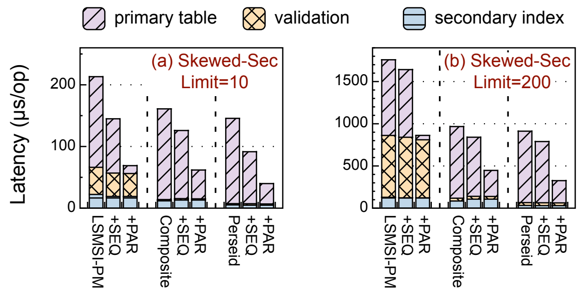


Figure 11: Non-index-only query performance on Leveling-based LSM table.

the composite index turns update operations into insert operations, the index operations limit its write scalability; and because it expands the number of KV pairs and thus has a bigger tree height, the increased index overhead limits its query scalability. As for the log-structured approach, the poor data locality restricts its query performance, especially for large-range queries.

5.2.5 Non-Index-Only Query

We next evaluate the non-index-only query operations. Besides the basic compared secondary indexes, we also enhance them by applying the two optimizations (§4.4), sequence number zone map (+*SEQ*) and parallel primary table searching (+*PAR*) sequentially. In this experiment, we use 4 threads for parallel primary table searching. Figure 10 shows the performance and time breakdown of non-index-only query operations. Note that the breakdown of primary table time on +*PAR* only shows the time not covered by the secondary index and validation. PERSEID brings considerable improvements against the `LSMSI-PM`, even if it has the two optimizations applied. PERSEID outperforms `LSMSI-PM` by up to 62% and 2.3×, when without and with optimizations on primary table lookups (the sequence number zone map and parallel primary table searching), respectively.

Though the primary key index indeed reduces unnecessary point lookup operations on the primary table for `LSMSI-PM`, with advanced low-latency storage devices and sufficient DRAM caching, it also has significant overhead. On the contrary, the hybrid PM-DRAM validation of PERSEID reduces the primary table lookups with subtle extra overhead.

PERSEID's optimizations on primary table searching can also boost other compared secondary indexing. The zone map improves the overall query performance of the KV store with PERSEID by about 50%, and the parallel primary table searching further improves by up to 3.1×. However, the numbers are only 20-36% and up to 2.4× for `LSMSI-PM`, respectively. This is because these optimizations only accelerate the primary table lookups, but the `LSMSI-PM` still has huge overheads. In addition, `LSMSI-PM` has to conduct the heavy validation first then it can pass the lookup tasks to parallel worker threads. Therefore, parallel threads cannot work adequately for `LSMSI-PM`.

We also implement secondary indexes and conduct the experiments on a leveling-based LSM primary table (LevelDB [24]). Figure 11 shows the results of *Skewed-Sec* as an example. The main difference is that the sequence number zone map is less effective on leveling-based LSM primary tables. However, the zone map is still effective when the limit is small, since the latest few records stay in MemTables or SSTables in lower levels like L_0, and these components can be filtered by sequence number with a high probability.

5.3 Mixed Workloads

Workload	Operation Ratios			
	Upsert	Get	Index-Only Query	Non-Index-Only Query
Write-Heavy	70%	10%	10%	10%
Balanced	45%	10%	25%	20%
Read-Heavy	20%	20%	40%	20%

Table 1: Mixed workloads description.

In this section, we evaluate PERSEID, the composite P-Masstree, and `LSMSI-PM` under mixed workloads. The mixed workloads consist of interleaved and various types of operations, which are more representative of real-world workloads. Each workload has 40 million operations, containing both *Skewed-Pri* and *Skewed-Sec* operations. Table 1 describes these workloads' traits. Each system is prefilled with 80 million records before performing the workloads. We also enable PERSEID's optimizations on primary table searching (i.e.,

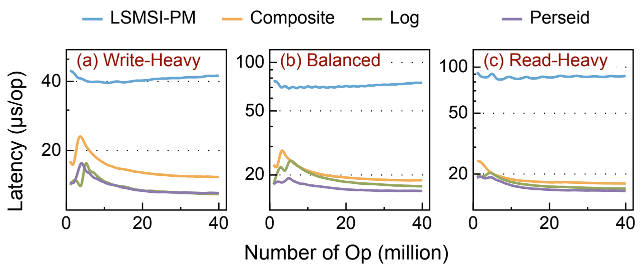


Figure 12: Performance of mixed workloads.

the sequence number zone map and parallel primary table searching) for all systems.

Figure 12 reports the average operation latencies every million operations. At the beginning of the Write-Heavy workload and the Balanced workload, PM-based secondary indexes have a spike in latency, which is mainly caused by seek-driven compaction in the LSM primary table. PERSEID outperforms `LSMSI-PM` significantly under different mixed workloads. Even though the overhead of the primary table dominates the whole operations, PERSEID still has visible advantages against the other PM-based indexes. Note that `PS-Tree` has much less capacity overhead than the composite index. As we set the limit on return results to 10 for query operations, the log-structured approach is not affected too much by its poor data locality.

5.4 Recovery Time

We evaluate the recovery time of PERSEID and `LSMSI-PM` after the Zipfian upsert workload that contains 200 million upsert operations with a single thread. Since we only need to recover the volatile validation hash table in PERSEID, it takes 2.7 seconds to scan the persistent hash table and rebuild the volatile hash table. By contrast, it takes 2.3 seconds and 1.4 seconds to recover the LSM-based secondary index and the primary key index, respectively. Their recovery time is mainly spent on rebuilding MemTables from logs and varies with the size of MemTables.

6 Related Work

Secondary Indexing in LSM-based KV stores. Qader et al. [47] conduct a comparative study on secondary indexing techniques in LSM-based systems. They conclude and evaluate several common secondary indexing techniques, including filter-based embedded index, composite index, and posting list. DELI [50] proposes an index maintenance approach that defers expensive index repair to compaction of the primary table. Luo et al. [40] propose several techniques for LSM-based secondary indexes, improving data ingestion and query performance. However, their techniques are mainly saving random device I/Os for traditional disk devices which reduce random reads at the cost of more sequential reads. Based on key-value separation [39], SineKV [36] keeps both the primary index and secondary indexes pointing to the record values. Thus, secondary index queries can get records directly without searching the primary index. However, SineKV has to discard the blind-write attribute and maintain index consistency synchronously. Cuckoo Index [32] enhances the filter-based indexing with a cuckoo filter. However, as a filter-based index, Cuckoo Index does not support range queries.

Though there are many proposed optimizations, LSM-based secondary indexing is not efficient enough due to the nature of LSM-trees. In this work, we revisit the design of the secondary index with persistent memory.

PM-based indexes. There has been plenty of research on high-performance PM indexes [13, 25, 31, 33, 42, 45, 46, 52, 56, 61]. These general-purpose indexing are not directly competent for efficient secondary indexing.

Improving LSM-based KV stores with PM. There is a lot of work optimizing LSM-based KV stores with PM. NoveLSM [27] introduces a large mutable MemTable on PM to lower compaction frequency and avoid logging. SLM-DB [26] utilizes a B$^+$-Tree on PM to index KV pairs on disks; SSTables on disks are organized in a single level, which reduces the compaction requirements. MatrixKV [58] places level L_0 on PM and adopts fine-granularity and parallel column compaction to reduce write stalls in LSM-trees. Facebook redesigns the block cache on PM to reduce the DRAM usage and thus reduce the total cost of ownership (TCO) [21, 28]. Different from these efforts, this work revisits the secondary indexing for LSM-based KV stores with PM.

7 Conclusion

In this paper, we revisit secondary indexing in LSM-based storage systems with PM. We propose PERSEID, an efficient PM-based secondary indexing mechanism for LSM-based storage systems. PERSEID overcomes the deficiencies of traditional LSM-based secondary indexing and existing PM-based indexes with naive approaches. `PS-Tree` achieves much higher query performance than state-of-the-art LSM-based secondary indexing techniques and existing PM-based indexes without sacrificing the write performance of LSM-based storage systems. The prototype of PERSEID is open-source at https://github.com/thustorage/perseid.

Acknowledgements

We sincerely thank our shepherd Baptiste Lepers and the anonymous reviewers for their valuable feedback. This work is supported by National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 61832011, U22B2023, 62022051), and Huawei.

References

[1] Apache cassandra. https://cassandra.apache.org/, 2022.

[2] Chirp: A Twitter-like workload generator. http://alumni.cs.ucr.edu/~ameno002/benchmark/, 2022.

[3] Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, 2022.

[4] Mongodb. https://www.mongodb.com, 2022.

[5] MySQL Glossary for Covering Index. https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_covering_index, 2022.

[6] Persistent Memory Development Kit. https://pmem.io/pmdk/, 2022.

[7] PostgreSQL: Documentation: Index-Only Scans and Covering Indexes. https://www.postgresql.org/docs/current/indexes-index-only-scans.html, 2022.

[8] Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit 2022. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022/, 2022.

[9] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source bdms. *Proc. VLDB Endow.*, 7(14):1905–1916, oct 2014.

[10] Sattam Alsubaiee, Michael J. Carey, and Chen Li. Lsm-based storage and indexing: An old idea with timely benefits. In *Second International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data*, GeoRich'15, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1185–1196, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[13] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.

[14] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[15] 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, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[17] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171. USENIX Association, November 2020.

[18] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. Nvalloc: Rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 115–127, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 365–378, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Facebook. Rocksdb. `https://rocksdb.org/`, 2022.

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

[24] Sanjay Ghemawat and Jeff Dean. Leveldb. `https://github.com/google/leveldb`, 2022.

[25] 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, Oakland, CA, February 2018. USENIX Association.

[26] 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, Boston, MA, February 2019. USENIX Association.

[27] 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, Boston, MA, July 2018. USENIX Association.

[28] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 821–837. USENIX Association, July 2021.

[29] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Power-optimized deployment of key-value stores using storage class memory. *ACM Trans. Storage*, 18(2), mar 2022.

[30] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. Rethink the scan in mvcc databases. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 938–950, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424–439, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G. Andersen. Cuckoo index: A lightweight secondary index structure. *Proc. VLDB Endow.*, 13(13):3559–3572, sep 2020.

[33] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 425–441. USENIX Association, November 2020.

[36] Fei Li, Youyou Lu, Zhe Yang, and Jiwu Shu. Sinekv: Decoupled secondary indexing for lsm-based key-value stores. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1112–1122, 2020.

[37] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+trees: Optimizing persistent index performance on 3dxpoint memory. *Proc. VLDB Endow.*, 13(7):1078–1090, mar 2020.

[38] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.*, 13(10):1147–1161, April 2020.

[39] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association.

[40] Chen Luo and Michael J. Carey. Efficient data ingestion and query processing for lsm-based storage systems. *Proc. VLDB Endow.*, 12(5):531–543, jan 2019.

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

[42] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 1–16. USENIX Association, February 2021.

[43] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 183–196, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.*, 13(12):3217–3230, aug 2020.

[45] 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, Boston, MA, February 2019. USENIX Association.

[46] 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, page 371–386, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 551–566, New York, NY, USA, 2018. Association for Computing Machinery.

[48] 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, page 497–514, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 1–16, USA, 2014. USENIX Association.

[50] Yuzhe Tang, Arun Iyengar, Wei Tan, Liana Fong, Ling Liu, and Balaji Palanisamy. Deferred lightweight indexing for log-structured key-value stores. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 11–20, 2015.

[51] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. Pacman: An efficient compaction approach for Log-Structured Key-Value store on persistent memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 773–788, Carlsbad, CA, July 2022. USENIX Association.

[52] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box approach to NUMA-Aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 93–111. USENIX Association, July 2021.

[53] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. Replicating persistent memory key-value stores with efficient rdma abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, July 2023.

[54] 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, Santa Clara, CA, July 2015. USENIX Association.

[55] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, and Jiwu Shu. Petps: Supporting huge embedding models with persistent memory. *Proc. VLDB Endow.*, 16(5):1013–1022, jan 2023.

[57] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[58] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[59] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Order-preserving key compression for inmemory search trees. SIGMOD '20, page 1601–1615, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: A key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 194–209, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: Differential indexing for persistent memory. *Proc. VLDB Endow.*, 13(4):421–434, December 2019.

[62] 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, Carlsbad, CA, October 2018. USENIX Association.