Read-Optimized Persistent Hash Index for Query Acceleration through Fingerprint Filtering and Lock-Free Prefetching

Renzhi Xiao[†], Dan Feng[*†‡], Yuchong Hu[*‡], Hong Jiang[§], Lin Wang[‡], Yucheng Zhang[†], Lanlan Cui[¶], Guanglei Xu[†], Fang Wang[†]

[†]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China*

[‡]*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*

[§]*Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, USA*

[¶]*School of Computer Science and Engineering, XI'AN University of Technology, Xi'an, China*

[*]*Correspongding Author: Dan Feng (dfeng@hust.edu.cn) and Yuchong Hu (yuchonghu@hust.edu.cn)*

{*rzxiao,wanglin2021,grayxu,wangfang*}*@hust.edu.cn*, hong.jiang@uta.edu, zhangyc_hust@126.com, cuilanlan@xaut.edu.cn

***Abstract*—Hash indexes are widely used in key-value storage systems due to their ability to perform rapid single-point queries. The persistent memory (PM) technology has received significant attention in both academia and industry due to its high performance, non-volatility, and large capacity characteristics. Currently, hash indexes tailored for persistent memories have been extensively researched. However, through an in-depth experimental study, we have discovered that existing persistent hash indexes suffer from low query performance. This is primarily due to persistent memory's higher read latency than DRAM's, which reduces the performance of both positive and negative queries in persistent hash indexes. Additionally, the former's higher read lock overhead further diminishes query performance.**

To address the above problems, we propose in this paper a Read-Optimized Persistent Hash Index, referred to as ROPHI, based on fingerprint filtering and lock-free prefetching. By employing a fingerprint filtering method, ROPHI introduces a DRAM-based Cuckoo filter to store fingerprints of keys on top of the PM-based hash table, effectively mitigating the time-consuming access overhead of persistent memory hash tables by accessing only the DRAM-based filter. Additionally, ROPHI employs lock-free prefetching for positive query acceleration, utilizing lock-free optimistic concurrent read techniques to avoid read lock overhead and high-speed cache prefetching techniques to reduce access overhead to persistent memory. Experimental results on the Intel Optane DC Persistent Memory Module (DCPMM) platform demonstrate that ROPHI significantly improves query performance over existing persistent hash index schemes. Specifically, ROPHI achieves an improvement of 2.67x-13.59x in negative query performance and 1.72x-7.86x in positive query performance. ROPHI outperforms the state-of-the-art SmartHT in positive query throughput by 34.5%, and in insertion and deletion throughput by 9.20% and 19.87% respectively, while sacrificing only 1.93% of negative query throughput. Additionally, it achieves a 5.07x improvement in recovery efficiency.

***Index Terms*—persistent memory, PM-based hash index, overall query performance, multi-threaded scalability**

This work was supported by National Key R&D Program of China (No. 2023YFB4502100), NSFC (No. 61821003, 62262042, U22A2027), Jiangxi Provincial Natural Science Foundation (No. 20224BAB202017), the Open Project Program of WNLO (No. 2021WNLOKF012), and Key Laboratory of Information Storage System Ministry of Education of China.

I. Introduction

With continuous technological and research advances, the persistent memory (PM) technology is poised to effectively bridge the chasm between traditional volatile memory and non-volatile storage paradigms. Unlike its volatile counterpart, which succumbs to data loss upon power cessation, persistent memory upholds data integrity even in the absence of power. This distinguishing trait promises numerous advantages, encompassing expedited boot times, fortified system resilience, and augmented data longevity. Moreover, persistent memory unlocks novel prospects in data-intensive applications by enabling expansive, byte-addressable storage for direct access by the CPU, obviating the latency encumbrance of conventional storage solutions.

Hash indexes, known for their rapid point-query lookup performance at a consistent pace, are pivotal components in memory-driven key-value (KV) storage systems such as Memcached and Redis. In instances where multiple keys converge to a singular location, triggering hash conflicts, measures such as rehashing or resizing become imperative when the hash table reaches capacity. The advent of PM has spurred a wave of research endeavors aimed at crafting efficient PM-based hash indexes. Notable among these innovations are CCEH [1], Dash [2], Level hashing [3], Clevel [4], PCLHT [5], and SmartHT [6].

When conducting queries in a hash table, a negative query verifies the absence of an element, while a positive query locates an existing element. The overall query functionality accommodates both negative and positive queries. However, the aforementioned persistent hash indexes still suffer from low query throughput performance. Our experimental investigation in Section II-C on end-to-end query performance indicates that persistent hash indexes exhibit lower query throughput than their DRAM-based counterparts. The negative query throughput of existing persistent hash indexes in DRAM exceeds their PM counterparts by 2.38x to 5.76x. The higher read latency of PM and the longer query path increase the time needed for negative queries, leading to lower performance. Similarly, the positive query throughput of persistent hash indexes in DRAM far surpasses that in PM, ranging from 2.18x to 4.58x higher. This discrepancy is due to PM's elevated read latency and increased read lock overhead, resulting in reduced positive query throughput for persistent hash indexes.

Motivated by the above analysis, we aim to propose a read-optimized persistent hash index, or ROPHI, to overcome the above performance shortfalls of PM-based hash indexes, ensuring high query performance for both negative and positive queries. ROPHI accelerates negative queries by employing a fingerprint filtering method. It introduces a DRAM-based Cuckoo filter alongside the PM-based hash table to store key fingerprints, effectively mitigating the time-consuming access overhead of persistent hash tables. To speedup positive queries, ROPHI employs lock-free prefetching, utilizing lock-free optimistic concurrent read techniques to avoid read lock overhead and high-speed cache prefetching techniques to reduce access overhead to PM. Experimental results on the Intel Optane DCPMM platform reveal that ROPHI significantly enhances query performance over existing persistent hash index schemes. ROPHI achieves substantial improvements, ranging from 2.67x to 13.59x in negative query performance and 1.72x to 7.86x in positive query performance. Moreover, it increases insertion throughput by 1.60x to 24.88x and deletion throughput by 3.73x to 16.95x. ROPHI outperforms the state-of-the-art SmartHT in positive query throughput by 34.5% while only sacrificing 1.93% of negative query throughput. Additionally, it increases the insertion and deletion throughputs by 9.20% and 19.87%, respectively, alongside a notable 5.07x improvement in recovery efficiency.

II. Background and Motivation

A. Persistent Hash Indexes

Advanced PM-based persistent hash indexes, exemplified by CCEH [1], Dash [2], Level hashing [3], Clevel [4], PCLHT [5], and SmartHT [6], surpass traditional memory hash indexes by overcoming capacity limitations and data loss issues. CCEH [1] expands dynamically via segment splitting to avoid full-table resizing but faces lower load factors. Dash [2] deals with the challenge of frequent segment splitting by adopting balanced strategies for insertion, replacement, and stashing, resulting in improved load factor albeit with increased PM reads. Level hashing [3] achieves economical resizing and minimal data consistency overhead with log-free failure atomicity operations tailored for persistent memory. Clevel [4] improves upon Level hashing with lock-free concurrency and non-blocking read-resizing. PCLHT [5] enhances read concurrency with a lock-free mode and maintains write concurrency correctness using bucket locks. SmartHT [6] is a persistent hash index that utilizes a hybrid DRAM-PM memory architecture, enhancing negative query performance with a DRAM-based Bloom filter.

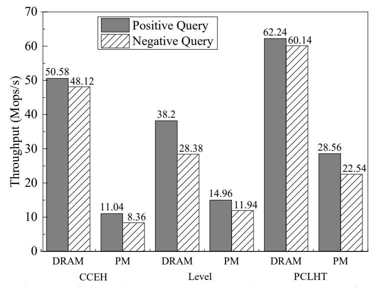

Fig. 1. Comparison of 32-threaded query throughput of different persistent hash indexes in both DRAM and PM memory.

The current PM-based persistent hash indexes excel in either negative or positive query optimization. However, none effectively addresses the challenge of optimizing both types of queries simultaneously, resulting in low overall query performance. Hence, there is a critical need for research to prioritize resolving the issue of overall query optimization.

B. Cuckoo Filter

The Cuckoo Filter [7] is a data structure that employs partial-key cuckoo hashing [8] to store item fingerprints, characterized by low storage overhead and efficient query performance. We utilize a simple four-slot hash bucket cuckoo filter, where each item can be stored in eight slot positions corresponding to two hash functions. In contrast to traditional single-slot cuckoo hashing, the Cuckoo filter sacrifices some query efficiency by exploring more alternative positions to enhance insertion efficiency and space utilization. Similar to traditional single-slot cuckoo hashing, Cuckoo filter insertion operates by directly inserting into idle alternative positions, or by performing a kick-out operation before insertion if no free positions are available. Because Cuckoo filters store only partial keys (fingerprints) of items rather than their full keys, they exhibit minimal storage overhead. A Cuckoo filter can determine key existence through fingerprint matching, enabling direct identification of non-existent keys. However, since different keys may have identical fingerprints, a false positive detection may occur where a key thought to exist does not correspond to actual data.

C. Overall Query Analysis

Existing persistent memory hash indexes such as Level Hashing, Clevel, CCEH, Dash, and PCLHT primarily face the challenge of insufficient overall query performance, encompassing both positive and negative query operations. For instance, in Figure 1, we compare the query performance of CCEH, Level, and PCLHT when entirely stored in DRAM or PM, with specific analysis as follows.

Observation 1. Reduced performance in negative queries for persistent hash indexes. The analysis in SmartHT [6] indicates that existing PM-based hash indexes achieve negative query performance ranging from 45.1% to 79.8% of their positive query counterparts. This discrepancy arises from the longer search paths inherent in negative queries. Furthermore, Intel Optane Persistent Memory exhibits higher read latency compared to DRAM, with sequential read latency being twice that of DRAM and random read latency three times greater [9]. Figure 1 illustrates that the DRAM negative query throughput of existing persistent hash indexes surpasses that of PM by 2.38 to 5.76 times. This difference can be attributed to the increased read latency of Intel Optane Persistent Memory, which extends the time required for negative queries in hash indexes utilizing this technology, ultimately resulting in reduced negative query performance.

Observation 2. Lower concurrent performance of positive queries for persistent hash indexes. The increased latency in persistent memory reads and the greater overhead of read locks diminish the concurrent performance of persistent hash indexes for positive queries. Figure 1 illustrates that the positive query throughput of the current persistent hash index in DRAM is 2.18 to 4.58 times higher than in PM. This discrepancy arises from the fact that PM's read latency is 2 to 3 times greater than that of DRAM [9]. Since positive queries target elements stored in the PM-based hash index, each query operation incurs multiple instances of heightened persistent memory read access overhead, thereby reducing query efficiency. Moreover, the added overhead of lock-based concurrency control [4] further dampens the concurrent performance of positive queries.

The cuckoo filter excels in dynamic insertion and deletion, facilitating rapid queries on larger-scale indexes. Leveraging insights from SmartHT [6], situating the filter in DRAM reduces path length and optimizes read lock overhead, achieving a balance between positive and negative queries to elevate overall query performance.

III. The Design of ROPHI

This section explores the design of ROPHI, a read-optimized persistent hash index that combines fingerprint filtering and lock-free prefetching to accelerate overall queries.

A. *ROPHI Overview*

ROPHI is a persistent hash table designed for hybrid memory consisting of both DRAM and PM. The Cuckoo filter is placed in DRAM to store fingerprints, accelerating negative query operations. Meanwhile, the chained hash table is placed in PM to facilitate fast persistence of key-value pairs. Figure 2 illustrates the architecture of the ROPHI system based on hybrid memory.

The DRAM-based Cuckoo filter of ROPHI uses a cuckoo hashing structure with each hash bucket containing four slots. Each element is mapped to the Cuckoo filter through two independent hash functions. The first hash function computes the hash value, h1, of the element, while the second hash function computes the hash value, h2, by performing an XOR operation on h1 and the hash value of the element's fingerprint. By XORing the hash value of the element's fingerprint, elements are distributed more evenly throughout the Cuckoo filter. In the Cuckoo filter, if the feedback is that an element does not exist, it is determined that the element indeed does not exist, thereby eliminating false negatives. However, if the feedback indicates that the element exists, there is a degree of uncertainty because different elements may have the same fingerprint, thus there is a possibility of false positives. Therefore, when the Cuckoo Filter reports the existence of an element, further confirmation through the PM-based hash table is necessary to verify whether the corresponding element truly exists.

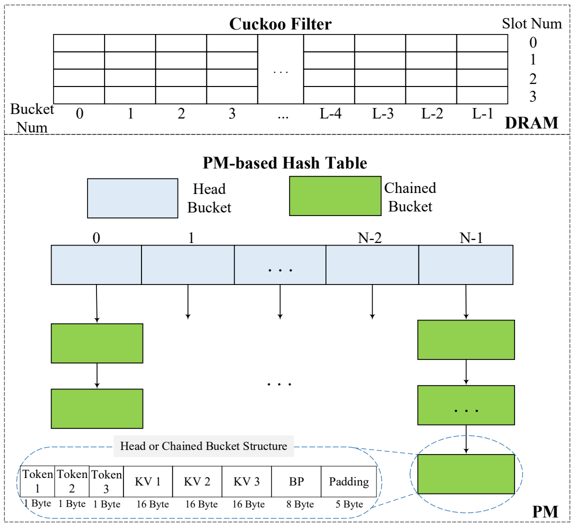

Fig. 2. The architecture of ROPHI.

The PM-based hash table of ROPHI adopts the same structure as SmartHT [6], utilizing chained hashing to resolve hash collisions. To ensure correctness in multi-threaded concurrency, the system employs a mutex write-lock mechanism. To reduce the overhead of data consistency guarantees, an atomic merge-flush mechanism is introduced. Additionally, optimization is performed by designing the head bucket to shorten the average chain length, thereby enhancing the positive query performance of the PM-based hash table. ROPHI utilizes lock-free optimistic read technology to avoid additional read lock overhead and employs prefetching techniques to reduce accesses to the PM-based hash table, enhancing positive query concurrency performance.

Each hash bucket has a size of 64 bytes, comprising three key-value pair slots (each with a size of 8 bytes for key and value), one 8-byte bucket pointer (BP) to the next bucket, and three 1-byte tokens. Furthermore, there are 5 bytes of padding information to maintain alignment.

B. *Overall Query Acceleration Method*

To synergistically optimize both negative and positive query performance and enhance overall query performance, ROPHI proposes an overall query acceleration method based on fingerprint filtering and lock-free prefetching.

1) ***Negative Query Acceleration Based on Fingerprint Filtering:*** Fingerprints are generated by using a simplified representation of keys, such as using one or two-byte partial keys to represent the full key. Traditional Bloom filters slightly outperform Cuckoo filters in negative query efficiency due to their efficient bit array structure. However, they do not support dynamic deletion operations for elements. Instead, ROPHI utilizes a space-efficient Cuckoo filter [7] that supports dynamic deletion operations to store fingerprints. This filter not only offers high space utilization efficiency but also achieves slightly higher positive query efficiency compared to traditional Bloom filters. The Cuckoo filter efficiently supports fingerprint insertion, deletion, and query operations through cuckoo hashing of its partial keys.

Since inserting into the Cuckoo filter may trigger a large number of eviction operations, resulting in additional writes, storing the filter in PM is not ideal. Therefore, ROPHI adopts a memory-aware filter placement strategy by storing the Cuckoo filter in DRAM instead of PM. This not only leverages the lower read latency of DRAM to enhance query performance but also alleviates the issue of write amplification in the Cuckoo filter, thus helping to maintain the longevity of the persistent memory system.

When inserting elements into ROPHI, the key-value pairs are first inserted into the hash table in PM, followed by inserting the fingerprints of the keys into the Cuckoo filter in DRAM. Since the Cuckoo filter always returns true for cases where an element is determined to not exist, eliminating false negatives, ROPHI can execute negative query operations by only querying the Cuckoo filter in DRAM. This allows ROPHI to return a result indicating that the element does not exist without accessing the higher-latency PM-based hash table.

The relationship between the number of fingerprint bits in the Cuckoo filter and the false positive rate and load factor is represented by Equation 1, where b denotes the number of fingerprint bits per entry, p denotes the false positive rate, and α represents the load factor of the Cuckoo filter. In this study, fingerprints of 1.5 bytes (i.e., b = 12) size are stored in the Cuckoo filter instead of storing the full keys, ensuring high query performance while maintaining a low false positive rate. With a load factor of 95% for this Cuckoo filter storing 200 million elements, with each bucket having a size of 4 slots, Equation 1 yields a false positive rate p of approximately 0.00296. Since different keys may have the same fingerprint, further confirmation in the PM-based hash table is required when the Cuckoo filter indicates that a key exists.

$$b = \frac{(log_2(\frac{1}{p}) + 3)}{\alpha} \tag{1}$$

*2) **Positive Query Acceleration through Lock-Free Prefetching**:* ROPHI employs lock-free prefetching to boost positive query performance in the hash table. Unlike SmartHT, which relies on shared read locks in PM-based hash tables for multi-threaded read concurrency, ROPHI's PM-based hash table adopts a lock-free optimistic read approach, eliminating additional overhead from read locks. Specifically, during each read thread, a copy of the element to be read is first stored. Upon return, if the read value matches the copy, the lookup is successful; otherwise, if other threads have modified the value, a re-lookup is required. Given the higher read latency overhead of PM, ROPHI employs a prefetching technique to minimize access to higher-latency PM by utilizing lower-latency cache.

C. ROPHI Operations

*1) **Search**:* The search process in ROPHI begins by querying the DRAM-based Cuckoo filter using fingerprint filtering. Subsequently, it determines whether to proceed with retrieving the PM-based hash table utilizing lock-free prefetching. If the Cuckoo filter indicates that the key to be searched does not exist, it returns NONE directly; otherwise, a search in the PM-based hash table is necessary to prevent false positives. Since the read latency of DRAM is lower than that of PM, ROPHI can access only the DRAM-based Cuckoo filter during negative queries, thereby accelerating negative query performance by reducing accesses to the higher-latency PM-based hash table. When the Cuckoo filter indicates that the key exists, the lock-free prefetching method is employed to enhance the query performance of the PM-based hash table. The pseudocode for the search operation of ROPHI is shown in Algorithm 1.

Algorithm 1: Search

```
1  if the Cuckoo filter (CF) contains the key then
2      Compute the hash value (HV) and bucket number
         (BN) based on the provided key.;
3      bucket_PTR = &Buckets[BN];
4      while bucket_PTR != NULL do
5          for $i \leftarrow 0$ to 2 do
6              copy_val = bucket_PTR→slot[i].value;
7              if bucket_PTR→token[i] == 1 and
                 bucket_PTR→slot[i].key == key then
8                  if bucket_PTR→slot[i].value ==
                     copy_val then
9                      return bucket_PTR→slot[i].value;
10                 end
11                 else
12                     return NONE;
13                 end
14             end
15         end
16         bucket_PTR = bucket_PTR→next;
17         if bucket_PTR != NULL then
18             __builtin_prefetch(bucket_PTR, 0, 0);
19         end
20     end
21 end
22 return NONE;
```

*2) **Insertion**:* ROPHI first inserts the key-value pair into the PM-based hash table, followed by inserting the fingerprint of the key into the Cuckoo filter in DRAM. Each hash bucket in the PM-based hash table can hold a maximum of 3 key-value pairs, with each key and value being 8 bytes in size. The insertion process begins by hashing the key to determine the corresponding hash bucket and then locking the bucket area where the hash bucket resides. The entire insertion process of the PM hash table adopts an ordered atomic persistence method to ensure data consistency. This involves using memory fence (MFENCE) instructions to order the flushing sequence of cache lines with CLFLUSH instructions. ROPHI ensures the correctness of insertion through lock-based concurrency control. Algorithm 2 provides the pseudocode for the insertion operation of ROPHI.

Algorithm 2: Insert

```
1  Calculate specific BN based on the given key;
2  bucket_PTR = &Buckets[BN];
3  if Is_Exists_in_ROPHI(bucket_PTR, key) then
4      return False;
5  end
6  // Insert the key-value pair into ROPHI if it is not exist
7  (bucket_PTR, emptySlotNum) = Find_EmptySlot();
8  lock(Locks[BN/LockSize]);
9  if bucket_PTR != NULL and Is_Valid(emptySlotNum)
    then
10     bucket_PTR→slot[emptySlotNum].value=value;
11     bucket_PTR→slot[emptySlotNum].key = key;
12     bucket_PTR→token[emptySlotNum] = 1;
13     Persist this bucket to PM using CLFLUSH;
14     Ensure persistent ordering of PM using MFENCE;
15     CuckooFilter_Insert(key's fingerprint);
16     unlock(Locks[BN/LockSize]);
17     return True;
18 end
19 unlock(Locks[BN/LockSize]);
20 return False;
```

*3) **Deletion**:* The deletion operation in ROPHI involves first setting the token of the key-value pair to be deleted in the PM-based hash table to 0. Then, a cache line flush instruction is used to persist the token value to PM. Finally, the fingerprint of the key is removed from the Cuckoo filter in DRAM to prevent misjudgment. Similar to the insertion operation, ROPHI utilizes hash bucket area locks to achieve multi-threaded concurrency control. Before locking the specific hash bucket area, the Is_Exists_in_ROPHI function is used to determine if the key to be deleted exists. This function employs an optimization strategy for the DRAM Cuckoo filter; if the key does not exist, it returns directly. If the key exists, it is searched for in the specific hash bucket chain and then deleted.

D. Consistency Guarantee and Recovery

Consistency Guarantee: To address data inconsistency issues caused by CPU out-of-order writes, ROPHI must ensure data consistency in the event of system failures. However, due to performance optimizations, the sequence of flushing volatile cache lines to PM may not align with the program's write order, necessitating measures for consistency assurance. In ROPHI, where keys and values are both 8 bytes, aligning with the atomic write unit of persistent memory, cache line flush and memory fence instructions are employed to regulate the update sequence of key-value pairs, ensuring consistency during data writes to PM.

Recovery: To ensure correct recovery in the event of system failures, ROPHI adopts an ordered persistence atomic method, similar to previous research such as Clevel [4]. Memory fence instructions are added after cache line flush instructions to ensure ordered persistence of write operations. ROPHI inherits the PM-based hash table from SmartHT [6] to ensure data consistency in insertion and deletion operations. After a failure occurs, the fingerprints stored in the Cuckoo filter in DRAM are lost, requiring scanning of the entire PM-based hash table to rebuild the Cuckoo filter in DRAM. During recovery from a failure, ROPHI can scan the entire hash table, release unused persistent memory space, and reinsert the fingerprints of valid key-value pairs into the Cuckoo filter in DRAM.

IV. Performance Evaluation

A. Experiment Setup

Server Hardware Configuration: Intel Optane DCPMM were chosen as the persistent memory medium, with the evaluation encompassing ROPHI and existing persistent hash indexes. The evaluation was conducted on a Linux server featuring 2 CPU sockets, each accommodating 18 CPU cores. The server architecture comprised 2 Intel Xeon Gold 6240M CPUs, operating at 2.6GHz, complemented by 4 modules of 128GB DCPMM, 128GB of DRAM memory, and a 24.75MB last-level L3 cache. Ubuntu 20.04.1 served as the operating system, leveraging kernel version 5.15.0. The total memory capacity of the DCPMM totaled 512GB. All experiments were executed utilizing the ext4 Direct Access (DAX) file system and the application direct mode of Intel Optane DCPMM. To mitigate NUMA effects, all PM modules were consolidated within a single CPU socket. The implementation of ROPHI incorporated the libvmem library provided by the Persistent Memory Development Kit (PMDK).

Comparisons: We compared CCEH [1], Dash [2], Level Hashing [3], Clevel [4], and PCLHT [5], utilizing implementations outlined in HashEvaluation [10]. Both ROPHI and SmartHT [6] were implemented using the libvmem library. However, while SmartHT optimizes negative queries, ROPHI optimizes both positive and negative queries, resulting in higher overall query performance.

Workloads: The experiment used the PiBench [11] benchmark framework to generate workloads similar to SmartHT [6], incorporating two overall query workloads. This benchmark assessed the performance of positive queries (keys exist) and negative queries (keys do not exist), as well as multi-threaded concurrent insertions and deletions. Furthermore, the Processor Counter Monitor (PCM) interface from the processor counter monitoring library was utilized to scrutinize various operations from a low-level viewpoint, including read and write byte counts and cache hit rates. Details of the operation workloads are provided below.

- **Positive Query.** In the loading phase, insert 200 million key-value pairs. In the running phase, replace all inserts with queries.
- **Negative Query.** The loading phase set is identical to that of positive queries. During the running phase, query 200 million key-value pairs that are not in the hash table.
- **Overall Query A and B.** Insert 200 million key-value pairs during the loading phase, and execute 200 million

query operations during the running phase. Overall query A consists of 30% positive queries and 70% negative query operations, while overall query B consists of 70% positive queries and 30% negative query operations.

- **Insertion.** During the loading phase, do not insert any key-value pairs. During the running phase, insert 200 million key-value pairs.
- **Deletion.** Change all insert operations from the aforementioned insertion workload to deletion operations. The number of operations remains 200 million KV pairs.

All latency and throughput tests are based on the average of five consecutive runs, with a skew factor of 0.2 for all skewed workloads. Both key and value sizes are set to 8 bytes.

B. Query Performance Under Different Workloads

Figures 3 and 4 evaluate the query performance of various persistent hash indexes under different query workloads with uniform and skewed distributions.

In terms of parallel positive queries, ROPHI demonstrates superior performance compared to existing persistent hash indexes under both uniform and skewed distributions. Figures 3(a) and 4(a) illustrate that ROPHI achieves positive query throughputs of 60.12 Mops/s and 105.06 Mops/s with 64 threads, respectively, representing a performance improvement of 1.72x-7.86x over existing persistent hash indexes. The superior positive query performance of ROPHI is attributed to its PM-based hash table, which enhances positive query performance through lock-free prefetching techniques.

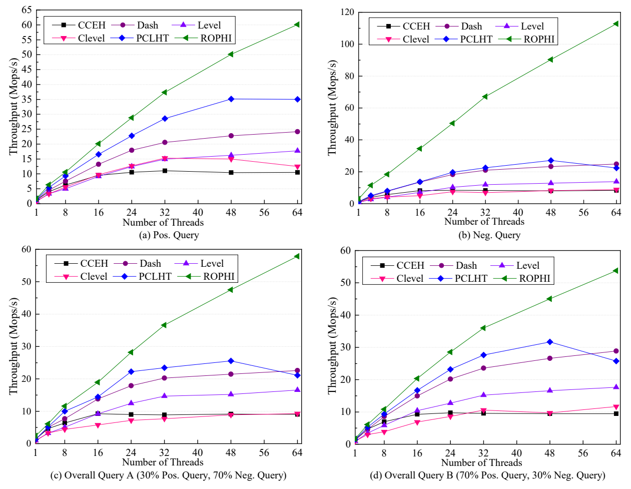

Fig. 3. Multi-threaded query scalability for uniform workloads.

In terms of parallel negative queries, ROPHI significantly outperforms existing persistent hash indexes, as illustrated in Figures 3(b) and 4(b). With 64 threads, ROPHI achieves negative query throughputs of 112.82 Mops/s and 122.02 Mops/s under uniform and skewed distributions, respectively, representing performance improvements of 2.67x-13.59x over existing persistent hash indexes. The superior negative query performance of ROPHI stems from its fingerprint-based negative query acceleration strategy, which reduces accesses to the high-latency PM hash table by solely accessing the DRAM cuckoo filter. This cuckoo filter can proactively determine the

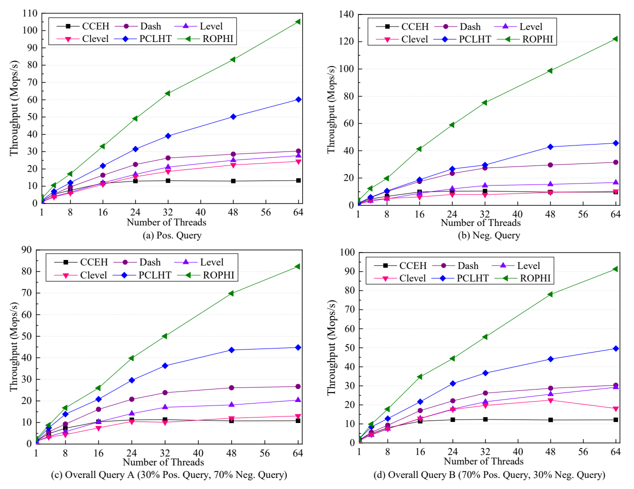

Fig. 4. Multi-threaded query scalability for skewed workloads.

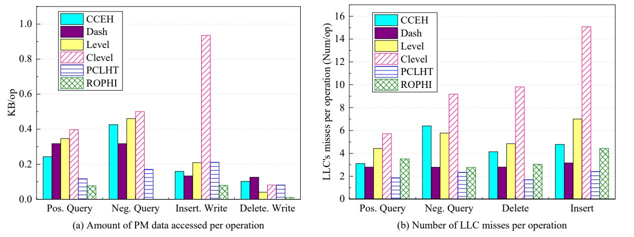

Fig. 5. The number of read/write bytes to PM and last-level cache (LLC) misses for each operation in various persistent hash indexes.

non-existence of key-value entries, resulting in zero data reads from persistent memory and favorable last-level cache misses, as depicted in Figure 5. However, existing persistent hash indexes suffer from longer access paths to persistent memory hash tables, leading to lower efficiency in negative queries.

In Overall Query A, comprising 30% positive queries and 70% negative queries, ROPHI exhibits outstanding query performance, surpassing existing persistent hash indexes, as detailed in Figures 3(c) and 4(c). With 64 threads, ROPHI achieves mixed query throughputs of 57.85 Mops/s and 82.35 Mops/s under uniform and skewed distributions, respectively, representing performance improvements of 1.84x-7.63x over existing persistent hash indexes. Similarly, Figures 3(d) and 4(d) demonstrate ROPHI's high query performance in Overall Query B, with mixed query throughputs of 53.75 Mops/s and 91.34 Mops/s under uniform and skewed distributions with 64 threads, respectively, representing performance improvements of 1.84x-7.55x over existing persistent hash indexes. ROPHI achieves superior mixed query performance by employing an overall query acceleration strategy based on fingerprint filtering and lock-free prefetching, effectively optimizing both negative and positive queries.

C. Insertion and Deletion Performance

Insert. In terms of insertion parallelism, ROPHI demonstrates high performance under both uniform and skewed distributions, as illustrated in Figures 6(a) and 7(a). With 64

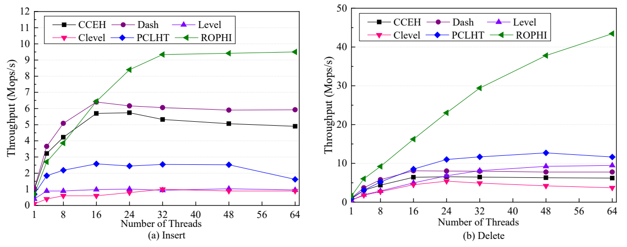

Fig. 6. Multi-threaded insertion and deletion scalability for uniform workloads.

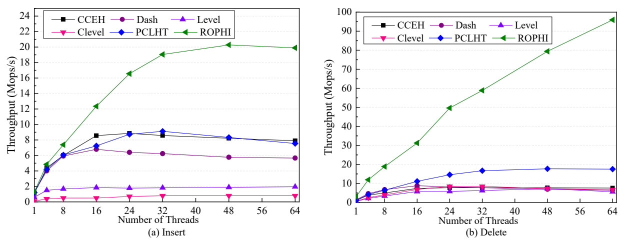

Fig. 7. Multi-threaded insertion and deletion scalability for skewed workloads.

threads, ROPHI achieves insertion parallel throughputs of 9.5 Mops/s and 19.9 Mops/s under uniform and skewed distributions, respectively, representing performance improvements of 1.60x-24.88x over existing persistent hash indexes. Although under uniform distribution, ROPHI's insertion performance may be slightly lower than CCEH and Dash, particularly below 16 threads, due to the overhead of inserting cuckoo filter fingerprints. However, in multi-threaded scenarios, especially with 64 threads, ROPHI exhibits superior insertion performance due to its excellent query performance and optimized merge-flushing of insertion operations.

Delete. Figures 6(b) and 7(b) illustrate the deletion concurrent throughput of various persistent hash indexes. Under 64 threads, ROPHI achieves deletion performance of 43.44 Mops/s and 95.93 Mops/s under uniform and skewed distributions, respectively. Compared to CCEH, Dash, Level, Clevel, and PCLHT, ROPHI's deletion performance is enhanced by a factor of 3.73x-16.95x. This enhancement can be attributed to ROPHI's superior query performance and its efficient lazy deletion mechanism, which minimizes the amount of data written to PM. This mechanism involves resetting tokens to 0 and atomically persisting them, as illustrated in Figure 5(a).

D. *Query Tail Latency*

In storage systems, reducing tail latency is crucial for enhancing user experience. This study evaluates the query tail latency of persistent hash indexes across various percentiles under 32 threads, as depicted in Figure 8.

ROPHI demonstrates relatively lower tail latency in positive queries, with its p99.99 positive query tail latency efficiency being approximately 99.27% of PCLHT's. Compared to CCEH, Dash, Level, and Clevel, ROPHI improves p99.99 positive query tail latency efficiency by 1.25 to 7.42 times, as illustrated in Figure 8(a). ROPHI excels in negative query tail latency, as depicted in Figure 8(b), with tail latencies of 1.02 microseconds and 3.96 microseconds at p99.9 and p99.99, respectively. This represents improvements of 5.24x-30.43x

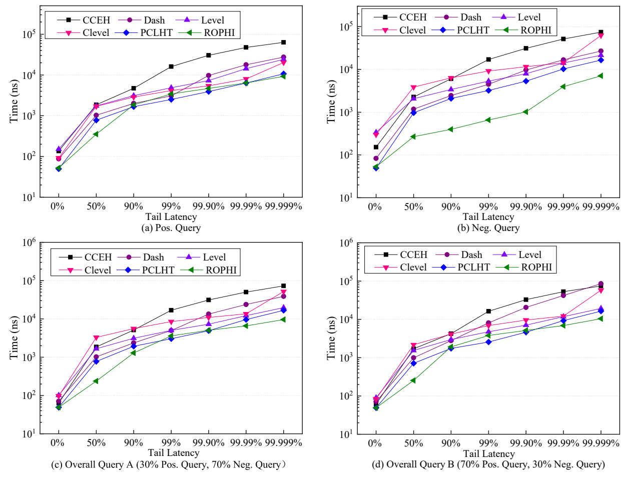

Fig. 8. Tail latency of queries at various percentiles in a uniform distribution scenario with 32 threads.

and 2.59x-12.95x over existing persistent hash indexes. The primary reason for ROPHI's high negative query tail latency performance lies in its fingerprint filtering-based negative query optimization strategy, which effectively reduces the access overhead of the PM-based hash table.

Additionally, Figure 8(c) demonstrates ROPHI's excellent tail latency efficiency in Overall Query A, with a p99.99 query tail latency of 6.58 microseconds, representing improvements of 1.46x-7.61x over existing persistent hash indexes. Similarly, Figure 8(d) also illustrates ROPHI's favorable query tail latency efficiency in Overall Query B, with a p99.99 query tail latency of 6.89 microseconds, which is 1.34x-7.68x higher than existing persistent hash indexes.

E. *Comparative Analysis against SmartHT*

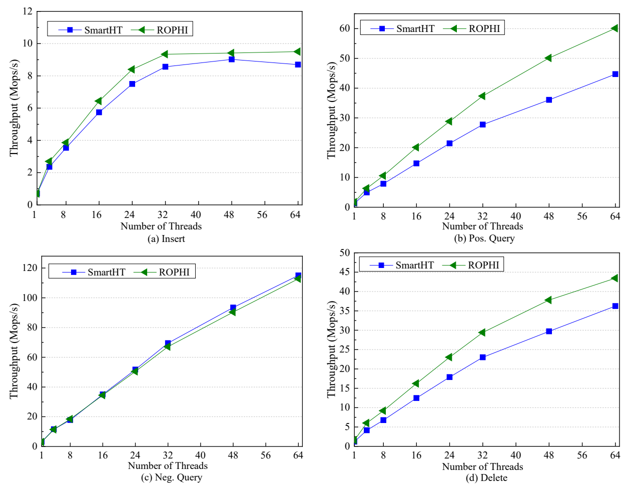

Fig. 9. Performance of ROPHI and SmartHT under uniform distribution with multiple threads.

1) Throughput Evaluation: The ROPHI and the state-of-the-art SmartHT methods each have their own strengths and weaknesses, as illustrated in Figures 9. Under the uniform distribution scenario, compared to SmartHT, ROPHI demonstrates higher concurrency in insertions, positive queries, and deletions, but slightly lags in negative query performance. Specifically, with 64 threads as an example, ROPHI improves positive query throughput by 34.50% and insertion throughput by 9.20%, while also enhancing deletion throughput by 19.87%, at the expense of only a 1.93% decrease in negative query throughput compared to SmartHT.

2) Failure Recovery: During system failures or power outages, it is necessary to restore the filter metadata in the DRAM of both ROPHI and SmartHT. A comparative analysis of the recovery time for ROPHI and SmartHT at the scale of inserting 50 million, 100 million, 150 million, and 200 million key-value pairs is presented in Figure 10. ROPHI demonstrates improved recovery performance compared to SmartHT. Specifically, the recovery times for ROPHI at the scale of 50 million, 100 million, 150 million, and 200 million key-value pairs are 4.6 seconds, 11.3 seconds, 19.1 seconds, and 29.9 seconds, respectively. These represent efficiency improvements of 5.07x, 4.31x, 3.94x, and 3.42x over SmartHT, respectively. ROPHI's superior recovery performance is attributed to its enhanced positive query and insertion efficiency.

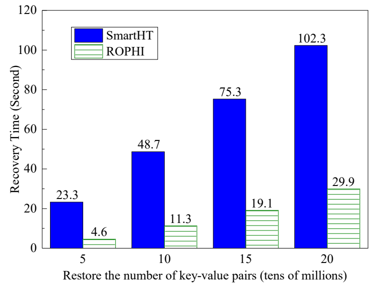

Fig. 10. Comparison of Recovery Time between ROPHI and SmartHT.

V. Related Work

PM-based Hash Indexes. Similar to Dash [2], OP-HMEH [12] and EEPH [13] are also variants of scalable hash tables based on DCPMM. While PCLHT [5], like other persistent hash indexes, primarily optimizes positive query performance at the expense of neglecting improvements in negative query performance, and SmartHT [6] primarily optimizes negative query performance with limited improvement in positive query performance, our work focuses on enhancing both positive and negative queries to provide high overall query performance. Inspired by SmartHT, our approach significantly improves negative query performance by placing a cuckoo filter in DRAM while optimizing positive query performance.

Hybrid DRAM-PM Memory Key-Value Stores. ChameleonDB [14] and Halo [15] all employ a hybrid DRAM-PM memory architecture, leveraging the low-latency access performance of DRAM and the fast persistence features of PM. However, while these systems boost performance by placing all indexes in DRAM, they incur additional recovery time overhead. Similarly, ROPHI adopts this hybrid memory architecture, but it utilizes only a small-sized DRAM to store cuckoo filters, optimizing hash table operation performance while reducing recovery overhead.

VI. Conclusion

To tackle the challenge of low overall query performance in persistent hash indexes, we propose ROPHI, a read-optimized persistent hash index method leveraging fingerprint filtering and lock-free prefetching. ROPHI boosts overall query performance by optimizing both negative and positive queries. For enhancing negative query efficiency, ROPHI employs a fingerprint filtering-based acceleration approach. Additionally, ROPHI integrates a lock-free prefetching strategy to expedite positive queries. The experimental results clearly illustrate ROPHI's superiority in achieving significantly improved overall query performance compared to existing persistent hash indexes.

References

[1] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Berkeley, CA, USA: USENIX, 2019, pp. 31–44.

[2] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, 2020.

[3] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *OSDI*. Berkeley, CA, USA: USENIX, 2018, pp. 461–476.

[4] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. Berkeley, CA, USA: USENIX, 2020, pp. 799–812.

[5] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2019, pp. 462–477.

[6] R. Xiao, H. Jiang, D. Feng, Y. Hu, W. Tong, K. Liu, Y. Zhang, X. Wei, and Z. Li, "Accelerating persistent hash indexes via reducing negative searches," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 174–181.

[7] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. New York, NY, USA: ACM, 2014, pp. 75–88.

[8] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[9] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Berkeley, CA, USA: USENIX, 2020, pp. 169–182.

[10] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: an experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 785–798, 2021.

[11] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.

[12] X. Zou, F. Wang, D. Feng, J. Zhu, R. Xiao, and N. Su, "A write-optimal and concurrent persistent dynamic hashing with radix tree assistance," *Journal of Systems Architecture*, vol. 125, p. 102462, 2022.

[13] Q. Chen, H. Hu, C. Deng, D. Liu, S. Li, B. Tang, T. Yao, and W. Xia, "Eeph: An efficient extendible perfect hashing for hybrid pmem-dram," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 1366–1378.

[14] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*. New York, NY, USA: ACM, 2021, pp. 194–209.

[15] D. Hu, Z. Chen, W. Che, J. Sun, and H. Chen, "Halo: A hybrid pmem-dram persistent hash index with fast recovery," in *Proceedings of the 2022 International Conference on Management of Data*. New York, NY, USA: ACM, 2022, pp. 1049–1063.