Pea Hash: A Performant Extendible Adaptive Hashing Index

ZHUOXUAN LIU and SHIMIN CHEN*, SKLP and Center for Advanced Computer Systems, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China

Hashing index is widely used to support efficient point operations. We observe that there is a conflict between performance and memory utilization goals. Existing hashing indices often have to trade off hash table access latency for better memory utilization. Moreover, many designs support only unique keys, and their performance is often suboptimal with skew workloads.

In this paper, we propose Pea Hash with two techniques to address the above two problems: (i) adaptive hashing strategy that holistically optimizes both access latency and memory utilization, and (ii) data-aware adaptive buckets that accommodate unique keys, and keys with various numbers of duplicates. We develop both an NVM-optimized Pea Hash and a DRAM-based Pea Hash index. Experiments on a machine equipped with Intel Optane DC Persistent memory show that compared to state-of-the-art NVM-optimized hashing indices, the NVM-optimized Pea Hash achieves up to 13.8x performance improvements with similar memory utilization. The DRAM-based Pea Hash outperforms existing in-DRAM hashing index designs, showing the generality of the proposed techniques.

CCS Concepts: • **Information systems** → **Data structures**; • **Hardware** → **Non-volatile memory**.

Additional Key Words and Phrases: Hash Index; Hashing Strategy; Duplicate Keys; Persistent Memory

ACM Reference Format:
Zhuoxuan Liu and Shimin Chen. 2023. Pea Hash: A Performant Extendible Adaptive Hashing Index. *Proc. ACM Manag. Data* 1, 1, Article 108 (May 2023), 25 pages. https://doi.org/10.1145/3588962

1 INTRODUCTION

Hashing index is widely used in database and big data systems [20, 22, 42, 44]. In main memory database systems, hashing index is used to build primary and/or secondary indices (e.g., on primary/foreign key columns), and to perform hash-based computation of join, aggregation, and deduplication operations [4, 21, 30]. In key-value store systems, such as Memcached [36, 48] and Redis [8, 43], hashing index plays the central role to implement the Get and Put operations. Therefore, it is important to study hashing index structures to better support the demands of database and big data systems.

Recent hashing index solutions exploit Non-Volatile Memory (NVM) [1, 7, 46] to manage the increasing amount of data. NVM can have much larger capacity than DRAM. A dual-socket server can be equipped with up to 6TB of Intel Optane DC Persistent Memory [13]. As the number of index entries increases, memory utilization and hash table resizing overhead become important design aspects to optimize. Level Hashing [52] proposes a top-bottom two-level structure to improve the memory utilization and reduce the resizing overhead. CCEH [37] and Dash [33] exploit Extendible

*Shimin Chen is the corresponding author.

Authors' address: Zhuoxuan Liu, liuzhuoxuan21b@ict.ac.cn; Shimin Chen, chensm@ict.ac.cn, SKLP and Center for Advanced Computer Systems, Institute of Computing Technology, CAS, No. 6 Ke Xue Yuan South Rd, Haidian District, Beijing, China and University of Chinese Academy of Sciences, No.1 Yanqihu East Rd, Huairou District, Beijing, China.

2836-6573/2023/5-ART108
https://doi.org/10.1145/3588962

Hashing [18] to amortize the resizing overhead across multiple segments. Apart from these NVM-optimized solutions, Viper [5] and Halo [24] devise hybrid designs that store hash tables in DRAM and key-value entries in NVM. Halo's in-DRAM structure is also based on Extendible Hashing.

We observe the following two main challenges in existing hashing index designs:

- *Conflict between performance and memory utilization goals*: Performance and memory utilization are two important design goals of hashing indices. It is desirable to achieve high memory utilization so that more index entries can be stored in the same amount of allocated memory space. However, optimization techniques to improve memory utilization often make performance worse. Techniques to fit more entries into a hash table often increase the number of locations that an entry can store in the hash table, e.g., by introducing 2-choice hashing schemes such as Cuckoo, allowing linear probing to store keys into multiple buckets, or adding stash/overflow buckets. Unfortunately, probing more locations incurs larger number of memory accesses, adversely impacting index performance. Consequently, existing hashing index designs have to make trade-offs between the two goals, e.g., in Level Hashing [52] and Dash [33], sacrificing performance for higher memory utilization.
- *Lack of efficient support for duplicate keys*: Duplicate keys are common in secondary indices (e.g., on foreign key columns) and hash-based join operations. However, existing hashing indices are designed to support mainly *unique* keys. One way to extend these structures to support duplicate keys is to store a pointer in the payload of a (key, payload) entry. Then the pointer can point to a buffer that stores the actual values of the index entries having the same key. However, this design can be inefficient. First, it incurs an extra (random) memory access to dereference the pointer for the payloads. Second, in real-world data sets, e.g., power-law graphs, there are keys with a large number of duplicates. The buffer to store payloads needs to be carefully designed to reduce memory access and space management overhead.

In this paper, we propose Pea Hash with two innovative techniques to address the above challenges.

- *Adaptive hashing strategy*: Pea Hash follows previous designs [24, 33, 37] to exploit the classic dynamic hash table — Extendible Hashing [18]. In Extendible hashing, there is a global directory. Each directory entry points to a segment, which consists of an array of buckets. Segment is the unit for hash table resizing. We observe that a segment is actually a hash table by itself. The hashing scheme inside a segment does not affect the global extendible hashing design. Therefore, we propose to adaptively modify the hashing strategy in a segment in order to resolve the conflicts between performance and memory utilization goals. Our goal is to optimize performance under different memory utilization ratios. When the number of entries in a segment is low, we employ simpler hashing strategies that reduce the number of memory accesses to achieve high performance. When the number of entries in the segment increases, we adaptively change the hashing strategy to consider more memory locations for a key to improve memory utilization. Specifically, we begin with the single hashing strategy. When the segment is about to resize, we switch to 2-choice hashing, then to 2-choice hashing with stash buckets. In this way, the design enjoys high performance when the memory utilization is low, and it obtains maximal utilization similar to the most sophisticated strategy.
- *Data-aware adaptive buckets*: A bucket in Pea Hash contains an array of (key, value) entries. To deal with duplicate keys, we propose to adapt the bucket design for the following three cases. (1) There is no duplicate. We store (key, value) entries in the bucket. (2) The number of duplicates is low. The bucket can accommodate a small number of duplicates. It is easy to extend index operations (e.g., search/delete) to look for multiple entries given a key. (3) The number of duplicates is high. We detect this case by examining the keys when the bucket is full. For the

key with many duplicates, we merge all the duplicate entries into one (key, pointer) entry. The pointer points to a value bucket, which stores the multiple values of the key. We encode this change in the bucket header so that index operations can distinguish between (key, value) and (key, pointer) entries. We carefully design the value bucket structure to reduce memory access and space management costs.

We exploit the above two techniques in the design of an NVM-optimized Pea Hash index. NVM bandwidth is (e.g., 2 ∼ 3x) lower than DRAM, and persisting data from CPU cache to NVM with special instructions (e.g., `CLWB` and `SFENCE`) incurs significant overhead [25, 31]. Therefore, our design aims to reduce the number of NVM accesses, especially persist operations as much as possible. We follow previous work [32, 33, 52] to employ fingerprints to accelerate in-bucket key search, and handle typical insert/delete/update operations with atomic NVM writes without logging. We employ the entry moving technique previously proposed in LB+-Trees [32] so that insertions are more likely to find an empty slot in the same cache line as the bucket header, and persist both the header and the new entry with a single operation, thereby significantly reducing the the number of persist operations. Moreover, we employ optimistic locking to support concurrent accesses, and design a light-weight NVM space management module to reduce NVM memory allocation overhead. Most of our designs are generally pertinent for any NVM implementation. For Intel Optane DCPMM in our experimental machine, we set the bucket size to 256B, which is the internal data transfer size in Intel Optane DCPMM.

In addition to the NVM-optimized design, we develop a DRAM-based Pea Hash index in order to show that the proposed techniques are general purpose for in-memory hashing indices. The implementation slightly modifies the NVM-optimized Pea Hash index by removing the persist operations and allocating space in DRAM. Please note that the DRAM-based Pea Hash index can be also used as the in-DRAM hash table in a hybrid DRAM-NVM hashing solution (e.g., Viper [5] and Halo [24]). Since the DRAM-based Pea Hash is sufficient to demonstrate the generality of our solution, we do not develop and evaluate such a hybrid solution.

We conduct extensive experiments on a server with Intel Optane DCPMM to evaluate Pea Hash. We compare the NVM-optimized Pea Hash with three state-of-the-art NVM-optimized hashing indices, i.e., Level Hashing [52], CCEH [37], and Dash [33]. Experimental results show that Pea Hash achieves up to 13.8x speedup, while attaining similar memory utilization and reducing the recovery time by an order of magnitude. Moreover, we compare the DRAM-based Pea Hash with DRAM-based CCEH and Dash, CLHT [14], a CPU cache optimized hash table, and Level Hashing, which has shown better in-DRAM performance than several previous hash tables, including BCH [19]. Experimental results show that Pea Hash achieves up to 7.0x improvement.

Contributions. The contribution of this work is three-fold. First, we summarize the common design challenges of hashing indices, and identify two main challenges that have not been addressed before. Second, we propose Pea Hash, an innovative hashing index. We develop two adaptive techniques to deal with the two main challenges, and combine a number of features for performance, persistence, and scalability. Finally, we conduct extensive experiments to compare Pea Hash and state-of-the-art hash tables. Experimental results exhibit significant benefits of our proposed design. We have made Pea Hash code publicly available[1].

Outline. The rest of the paper is organized as follows. Section 2 examines the main design considerations. Section 3 proposes Pea Hash. Section 4 presents the performance evaluation. Section 5 finally concludes the paper.

[1]https://github.com/schencoding/peahash

2 DESIGN CONSIDERATIONS

We examine the two main aspects of the hashing index design, i.e., performance and memory utilization, for better understanding the design challenges.

2.1 Performance

We consider the design choices for (1) the processing of individual index operations; (2) the resizing of hash tables; and (3) the handling of duplicate keys. (3) is often overlooked in previous studies.

2.1.1 Collision Handling in Normal Operations. A hashing index consists of an array of hash buckets [17, 27, 35, 41]. It employs hash functions to map keys to bucket IDs. Compared to tree-based indices, an ideal hashing index is capable of achieving O(1) time complexity for common operations, i.e., search, insert, and delete. The main obstacle to achieving the ideal performance is hash collision, i.e., multiple entries are mapped to the same bucket. Popular techniques to handle hash collisions can be categorized as follows:

(1) *Multiple entries in a bucket.* A common technique is to set the bucket size to be large enough to hold multiple key-value entries [14–16, 33, 52]. Another technique is chained hashing [14, 27, 35], which holds zero or more entries in a linked list in each bucket. Since pointer chasing may incur poor CPU cache behaviors, chained hashing is less popular in recent hashing index solutions. Multiple entries and chained hashing can also be combined in a design [14].

(2) *Multiple subsequent buckets for a collision entry.* A representative technique is linear probing [27, 35, 41]. If the bucket for an entry to insert is occupied, linear probing scans the subsequent buckets linearly until an empty bucket is found. Unfortunately, this technique may incur large overhead because of primary clustering, where there exist long sequences of occupied buckets. To mitigate this problem, Robin Hood hashing [10] and Hopscotch hashing [23] reduce or bound the distance between the location to store an entry and the hashed location. Alternatively, quadratic or other functions can be used to compute the subsequent bucket to probe.

(3) *2-choice hashing.* Two (independent) hash functions[2] are employed to compute two candidate buckets for a given key [3, 9]. Then an insert stores an entry to one of the buckets. A search or delete needs to probe both buckets. It is shown that instead of a single hash function, if we employ two hash functions, the number of collision entries per bucket can be dramatically reduced [3]. Cuckoo hashing [6, 16, 19, 26, 39, 40, 47] enhances 2-choice hashing with a technique to displace entries for insertions. If both computed buckets for a key to insert are full, cuckoo hashing randomly picks one entry E in the two buckets and displaces E to E's other bucket B to make room for the new entry. If this other bucket B is also full, then cuckoo hashing performs displacement on B recursively. However, the displacement design may result in endless loops [47] and incur cascading writes [15], leading to poor performance. Therefore, several recent designs perform one-hop displacement, i.e., at most one displacement without cascading [15].

(4) *Stash or overflow buckets.* Collision entries can be stored into a global stash or overflow area [15, 26, 27, 35]. Then a search needs to probe both the target bucket and the stash. Compared to the global stash, local stash is shared among subsets of buckets. Path hashing [51] builds a binary tree of buckets. The hash bucket array is the leaf nodes in the tree. Each non-leaf node is a stash bucket shared across all the leaf nodes in the subtree rooted by the non-leaf node. In level hashing [52], there are two levels of hash buckets. Every bottom level bucket is a stash bucket shared by two adjacent top level buckets.

[2]While this can be generalized to multiple-choice hashing, 2-choice hashing is the most popular in practice. 2-choice achieves good memory utilization and is better performing than multiple-choice hashing.

Recent hashing index solutions often combine several of the above techniques. PFHT [15] employs multi-entry buckets, 2-choice hashing with one-hop displacement, and global stash. CCEH [37] uses multi-entry buckets and linear probing with bounded probe distance. Level hashing [52] differs from PFHT by using a local stash design with two-level buckets. Dash [33] employs multi-entry buckets, local stash per segment of buckets, and a variant of 2-choice hashing with one-hop displacement, which computes the second hash bucket ID as the first bucket ID plus one.

2.1.2 Hash Table Resizing. A resizing operation is typically invoked when the hashing index fails to insert a new entry. The standard resizing approach allocates a new hash table whose size is larger (e.g., twice as large as the original table), rehashes *all* existing entries into the new hash table, then deletes the original table. However, this often incurs the long tail of hashing access latency.

Level hashing [52] proposes an in-place resizing design. Suppose the top and bottom levels in level hashing consist of 2N and N buckets, respectively. The resizing scheme allocates a new top level of 4N buckets, preserves the old top level as the new bottom level, then rehashes only entries in the old bottom level. It reduces the number of rehashed entries to about 1/3 of the standard scheme.

CCEH [37] and Dash [33] exploit Extendible Hashing [18] that divides the hash table into a number of hash segments and an array of segment pointers (a.k.a. directory). When a key fails to be inserted into a segment, the segment is split into two segments without affecting other segments. Only the entries in the single segment need to be rehashed, thereby significantly reducing the latency of individual resize operations.

2.1.3 Duplicate Key Support. It is important to support index entries with duplicate keys in practice. First, duplicate keys are frequently seen in secondary indices in database systems [20, 42]. Secondary indices are often created on non-primary-key columns in order to accelerate the evaluation of filtering predicates. It is also a common practice to create secondary indices on foreign key columns in order to efficiently enforce referential integrity when a referenced tuple is deleted or a referenced primary key is modified. Second, hashing indices play a key role in hash-based query processing algorithms [21], such as hash join operations. It is quite common to see multiple tuples with the same join key.

Unfortunately, existing hashing indices [14, 15, 33, 37, 52] mainly focus on unique keys. An insert with a duplicate key either is ignored or becomes an update to the existing (key, value) pair. One way to get around the problem is as follows. Instead of storing (key, value), we store (key, value pointer) in a hashing index, where the value pointer points to a buffer that contains all the values with the given key. Then the insert, search, and delete operations can be modified to access values through value pointers. However, this straightforward scheme incurs the cost of at least one extra pointer dereference for every index operation, and the overhead of memory allocation for the value buffers.

2.2 Memory Utilization

Memory utilization is another important aspect of the design. With higher utilization, a hashing index stores a larger number of index entries and sees fewer resizing operations.

2.2.1 Conflict: Performance vs. Memory Utilization. We observe that there is a conflict between performance and memory utilization in hashing index designs. The common collision handling schemes as discussed in Section 2.1.1 improve memory utilization, but degrade the hashing index performance. For example, for technique (1), a search has to check multiple entries in a bucket. For techniques (2) and (3), a search may need to examine two or more buckets in order to locate a key. For technique (4), a search not only checks the hashed bucket for a key, but also has to

visit the stash. Insert and delete operations incur similar overhead. Hence, it is desirable but very challenging to resolve such conflict.

2.2.2 Average Utility. The term, load factor, measures the instantaneous memory utilization of a hashing index [27, 35]. It is defined as the number of inserted entries divided by the total entry slots in the hashing index at a particular instant. Previous studies mainly consider the maximum load factor of a design, i.e., the load factor immediately before resizing.

However, we find that maximum load factor does not capture the full behavior of a hashing index design. First, the load factor is essentially a time varying function. It increases as index entries are dynamically inserted until an insert fails and triggers the hash table resizing operation. After resizing, there is a sharp drop of the load factor. Then the load factor will increase again with new inserts. The maximum load factor does not reflect this dynamic behavior. Second, the number of index entries cannot fully represent the memory space used. There are often in-bucket metadata and other auxiliary structures, such as the directory in Extendible Hashing [33, 37].

In this study, we propose the following new metric to succinctly describe the memory utilization across the dynamic process:

$$\text{Average Utility} := \frac{\sum_{k=1}^{n} N(k)}{\sum_{k=1}^{n} S(k)}$$

where $N(k)$ denotes the memory space taken by the inserted key-value entries after the k-th insert, $S(k)$ denotes the total space used in the hash table after the k-th insert, and n is the total number of inserted entries.

We compute the average utility for the baseline hash table that consists of an array of one-entry hash buckets without in-bucket metadata and auxiliary structures. Suppose the hash table contains M buckets, each (key, value) entry takes S_e bytes, and its maximum load factor is α. We consider $n{=}\alpha M$ before resizing:

$$\text{Average Utility} = \frac{\sum_{k=1}^{n} kS_e}{\sum_{k=1}^{n} MS_e} = \frac{n+1}{2M} \approx 0.5\alpha$$

A resizing doubles the hash table size to $2MS_e$ bytes. Then for $n{=}2\alpha M$, we have

$$\text{Average Utility} = \frac{\sum_{k=1}^{n} kS_e}{\sum_{k=1}^{\alpha M} MS_e + \sum_{k=\alpha M+1}^{2\alpha M} 2MS_e} = \frac{2\alpha M+1}{3M} \approx 0.67\alpha$$

For $n \gg alphaM$, the hash table goes through a number of resizing. We can show that the average utility is approaching 0.75α.

We can extend the above computation if a hashing index design has a constant factor f of extra space overhead for in-bucket metadata and auxiliary structure. That is, if all the entry slots take MS_e bytes, then the total space, including the extra space overhead, is $MS_e(1 + f)$. For such a hashing index, we can show that when $n \gg alphaM$, the average utility is approaching $\frac{0.75\alpha}{1+f}$.

We report the average utility of the studied hashing strategies in Section 4.

2.3 Design Challenges

We compare the state-of-the-art hashing indices in Table 1. Level hashing [52] has been shown to outperform previous designs, including PFHT [15] and BCH [19]. CCEH [37] and Dash [33] are the two most recent NVM-optimized hashing index solutions. They both exploit extendible hashing to reduce the resizing latency. CLHT [14] is a CPU cache optimized hash table. It uses cache-line sized buckets and chained hashing.

From Section 2.1–2.2, we observe two main challenges in existing hashing index designs:

- **Conflict between performance and utilization**: Collision handling techniques improve memory utilization but incur more memory accesses, essentially sacrificing hash table performance when the load is low.

Table 1. Comparing state-of-the-art hashing indices.

	Level	CCEH	Dash	CLHT	***Pea***
Average Utility	✓	–	✓	–	✓
Performance at Low Load	×	×	×	✓	✓
Performance at High Load	×	–	✓	×	✓
Support for duplicate keys	×	×	×	×	✓
Resizing Latency	×	✓	✓	×	✓

note: × poor, – ok, ✓good.

- **Lack of optimization for duplicate keys**: Existing designs mainly target unique keys. The solution to store (key, value pointer) in a hashing index incurs extra overhead of pointer dereference and memory allocation.

Our solution, Pea Hash, aims to address the two challenges, while building on ideas of state-of-the-art hashing indices.

3 PEA HASH

We propose Pea Hash, a Performant extendible adaptive hashing index, to address the aforementioned challenges with two main optimization techniques:

- Adaptive hashing strategies balance memory utilization and performance.
- Data-aware adaptive buckets accelerate index operations for both unique and duplicate keys.

In the following, we overview the Pea Hash structure in Section 3.1. Then, we focus on the two optimization techniques in Section 3.2 and 3.3, respectively. Finally, we present the NVM-optimized Pea Hash index in Section 3.4. To show the generality of our approach, we also implement a DRAM-based Pea Hash index. Since it is very similar to the NVM-optimized design, we briefly describe the DRAM-based Pea Hash index in the overview in Section 3.1.

3.1 Design Overview

Structure overview. Figure 1(a) depicts the structure of Pea Hash. The primary structure is an extendible hash table, which consists of a directory and a set of main segments. We adaptively choose the hashing strategy for every main segment based on the load factor of the segment. Below the primary structure is a set of stash segments, which are used when stash is employed in the chosen hashing strategy. The value segments on the right store the values of the same (duplicate) key. A set of auxiliary structures are used for concurrency control and segment management. The main, stash, and value segments are of the same size to simplify NVM space management.

Extendible hashing. We follow CCEH [37] and Dash [33] to employ extendible hashing [18] in Pea Hash. The directory in Figure 1(a) is an array of 2^G elements, where G is the global depth. Each element contains the segment ID and the local depth L of a main segment. $L \leq G$ is always true. 2^{G-L} elements in the directory point to the same main segment. A hash table visit uses G bits in the hash code computed from the key to retrieve the directory element. Then, it visits the associated main segment as an independent sub hash table. When an insert to a segment S fails, extendible hashing splits S into $K = 2^k$ segments (e.g., K=2 or 4). It allocates K new segments, S_1, ..., S_K, and rehashes S's entries to the new segments. Then it adjusts the directory. In the common case where S's local depth $L + k \leq G$, there are already $2^{G-L} \geq 2^k = K$ directory elements associated with S. Then the existing directory elements can be modified to point to S_1, ..., S_K. The local depth of the new segments are all set to L+k. In case where $L + k > G$, the directory needs to be expanded so that the new global depth $G' = L + k$, and then the common case can be applied. In this way, extendible hashing amortizes the resizing cost across multiple segments, significantly reducing the tail latency of hash table accesses.

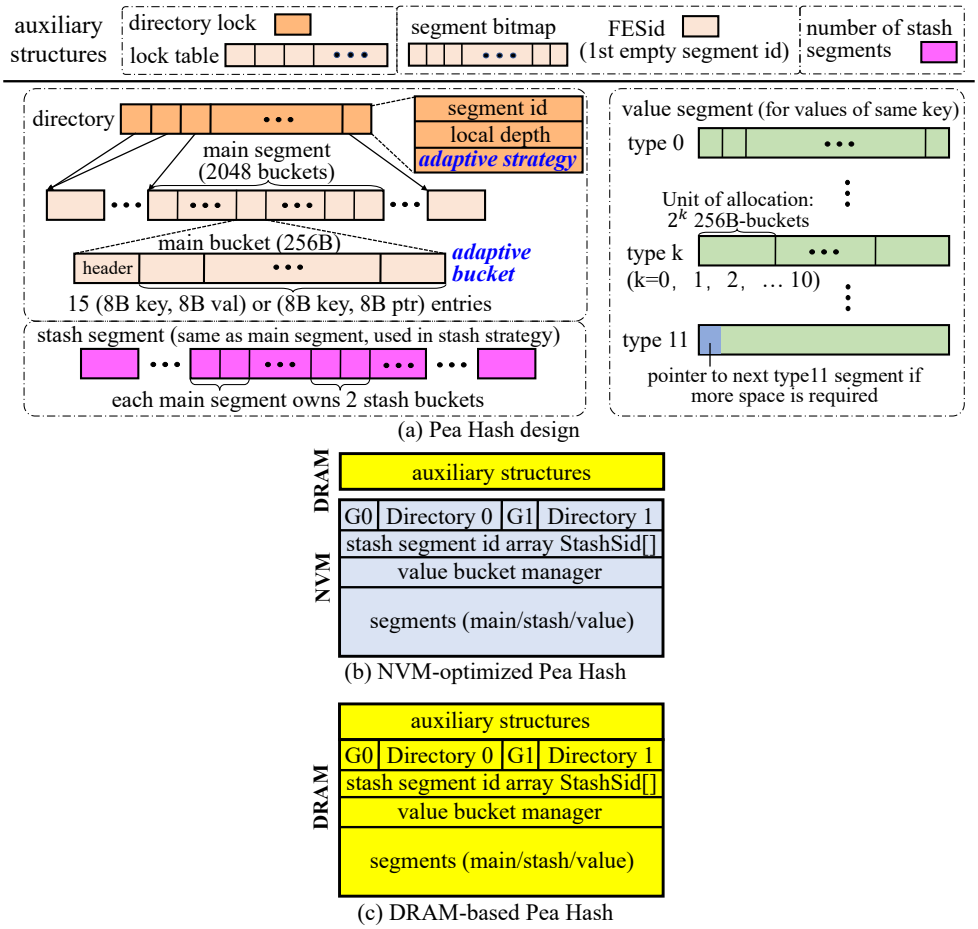


Fig. 1. Pea Hash overview.

Adaptive hashing strategy. We observe that when the load factor is low, it is feasible to employ simple hashing strategies that obtain high performance. When the load factor is high, we have to consider more sophisticated collision handling techniques. From this observation, we propose to adaptively set the hashing strategy for each main segment in its associated directory element(s) based on its load factor. We choose a sequence of hashing strategies, HS_1, HS_2, ..., HS_k, with increasing memory utilization but potentially decreasing performance. The initial strategy is HS_1. When an insertion fails under HS_i, we switch the strategy to HS_{i+1}. Segment split is invoked only when HS_k fails. Our design aims to achieve good performance under any load factors.

Data-aware adaptive buckets. As shown in Figure 1(a), each slot in a main bucket can be either a (8B key, 8B value) or a (8B key, 8B pointer). There are three cases: 1) A unique key is stored as a (key, value) entry; 2) A key with a few duplicates can be stored as multiple (key, value) entries; and 3) A key with many duplicates is stored as a (key, pointer) entry, where the pointer points to a value buffer in the value segments containing all the values of the key. Note that Case 1 and 2 reduce the pointer dereference overhead, and the value segments reduce the space allocation overhead in Case 3. We design the buckets to be data skew aware and to adaptively handle the three cases well.

NVM-optimized Pea Hash index. Figure 1(b) shows the memory layout for the NVM-optimized Pea Hash index. We allocated two (global depth, directory) in NVM to support directory expansion. Each main/stash/value segment is 512KB large. A directory entry is 8B large. It consists of (6B segment ID, 1B local depth, 1B hashing strategy), which supports up to $2^{48}\times$ 512KB = 2^{67}B of total hash table space. The segments area contains an array of segments. A main segment consists of 2048 buckets. We set the bucket size to 256B, DCPMM's internal data transfer size. A bucket is composed of a 16B header and fifteen 16B slots. A stash bucket has the same structure as a main bucket. Each main segment is associated with two stash buckets if stash is used in the chosen hashing strategy. The locations of the stash segments are recorded in the StashSid array. The value segments are organized to efficiently allocate space for value buffers of different sizes. All the structures are stored in NVM except the auxiliary structures, which are not critical for index persistence and are placed in DRAM.

Dram-based Pea Hash index. Figure 1(c) shows the memory layout of the DRAM-based Pea Hash index. We slightly modify the NVM-optimized Pea Hash index by removing the NVM persist operations, and allocating all data structures in DRAM.

3.2 Adaptive Hashing Strategy

Existing hashing index designs focus on a fixed hashing strategy [14, 15, 33, 37, 52]. In order to achieve high memory utilization, existing designs employ high-cost collision handling techniques. However, such high-cost techniques are also used when the load factor is low, which is unnecessary. Our idea is to gradually employ more sophisticated techniques as the load factor increases in order to achieve good performance for any load factor.

The transition from a simpler hashing strategy to a more sophisticated one subjects to certain qualifications. In the following, we first propose a containment relationship among hashing strategies, clarify the transition requirement in terms of the containment relationship, and present our solution in Pea Hash.

Containment relationship. We consider the following relationship among hashing strategies:

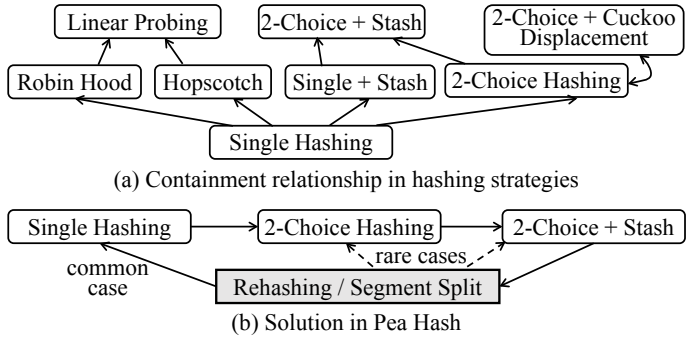
Definition 3.1 (Hashing Strategy Containment). *Hashing strategy HS_A is contained in hashing strategy HS_B, denoted as $HS_A \preceq HS_B$, if for any hash table instance T that uses HS_A, all the index entries in T can be correctly retrieved using HS_B.*

In other words, HS_A can be viewed as a special case of HS_B. For example, single hashing $\preceq$ 2-choice hashing. Single hashing refers to the simplest hashing strategy, where there is an array of hashing buckets and a single hash function maps a key to a bucket ID. 2-choice hashing computes 2 hash functions to map a key to 2 bucket IDs, and then it visits both buckets [3, 9]. If the first hash function is the one in single hashing, then a hash table instance using single hashing is also a valid instance in 2-choice hashing in that any existing key is found in its first computed bucket.

Figure 2(a) depicts the containment relationship as directed edges among hashing strategies. We consider combinations of the collision handling techniques in Category (2)–(4) in Section 2.1.1. Note that we assume that multi-entry buckets in Category (1) are employed for all the hashing strategies, which is the case in all state-of-the-art hashing indices [14, 15, 33, 37, 52]. We see the following containment relationship in Figure 2(a).

First, single hashing can be viewed as a special case of 2-choice hashing, single+stash, and linear probing variants.

Second, both Robin Hood [10] and Hopscotch [23] are contained in linear probing. While the two variants reduce or bound the distance between the bucket to store an entry and the hashed bucket, the resulting hash tables can still be accessed by linear probing.



(a) Containment relationship in hashing strategies

(b) Solution in Pea Hash

Fig. 2. Adaptive hashing strategy.

Third, a hashing strategy is contained by the same strategy with stash. For example, 2-choice $\preceq$ 2-choice+stash. The first strategy can be viewed as a special case where the stash contains zero entries.

Finally, 2-choice hashing and 2-choice+CD (cuckoo displacement) are equivalent in terms of containment. It is clear that 2-choice $\preceq$ 2-choice+CD. Moreover, CD can displace an entry to its other bucket to make room for a new entry. This does not change the fact that a key is still hashed to one of two buckets. Hence, 2-choice+CD $\preceq$ 2-choice.

Transition requirements. The sequence of hashing strategies, HS_1, HS_2, ..., HS_k, in adaptive hashing strategy must satisfy:

(a) For i=1,...$k-1$, $HS_i \preceq HS_{i+1}$;

(b) For i=1,...$k-1$, HS_{i+1} is expected to have higher memory utilization than HS_i.

Requirement (a) ensures that the transition cost is low because none of the existing index entries under HS_i need to be redistributed for HS_{i+1}. This minimizes the number of random memory accesses. Requirement (b) is important because the transition to HS_{i+1} is triggered when an insertion fails under HS_i. Since HS_{i+1} has higher memory utilization, it can accommodate more inserts.

Our solution. As shown in Figure 2(b) Pea Hash chooses single hashing, 2-choice hashing, and 2-choice+stash as the sequence in adaptive hashing strategy for a main segment. It is easy to verify that this sequence satisfies the transition requirements. Note that we avoid cuckoo displacement, which may incur a lot of expensive NVM persist operations in NVM-optimized hashing indices [52].

When an insert to a segment fails, Pea Hash triggers the change of the hashing strategy for the segment. Single hashing is employed when the load factor is low. It minimizes the number of memory accesses, obtaining the fastest index operations. For 2-choice hashing, we employ balanced inserts [15] to insert to the less occupied bucket as it leads to better memory utilization [3]. For 2-choice+stash, a main segment with 2048 buckets has two stash buckets in the stash segments, as shown in Figure 1(a). Hence, stash segments occupy less than 0.1% of the space used in Pea Hash. We record the stash segment IDs in an array `StashSid[]`. Given a main segment ID `sid`, the first stash bucket is at bucket (`2*sid%2048`) in stash segment `StashSid[sid/1024]`.

When an insertion fails under 2-choice+stash, we perform segment split for the main segment. As the load factor of 2-choice+stash is over twice as high as that of single hashing, we split the segment into four segments so that in most cases, single hashing can be employed for the resulting new segments, as shown in Figure 2(b). In our experiments, this common case occurs 99.4% of the time, while the transition to 2-choice occurs 0.6% of the time. We do not see the other rare case. Consequently, our chosen adaptive hashing strategy can be applied in the segments after split.

3.3 Data-aware Adaptive Buckets

Design goals. We achieve two goals in the design of data-aware adaptive buckets. First, we would like to achieve good performance for unique keys and keys with only a few duplicates. Pea Hash does not allocate a value buffer for each key. Instead, it tries to place entries in the main bucket as much as possible. This minimizes the number of pointer dereferences.

Second, we would like to achieve good performance for popular duplicate keys. Pea hash places (key, pointer) entries in the main bucket, and strives to store the values in a contiguous value buffer rather than in a linked list, in order to improve the performance for retrieving the values. Moreover, it places the (key, pointer) entries close to the header in the main bucket so that they tend to be in the same cache line as the header. This accelerates the retrieval of the value buffer pointer for a popular key.

Flexible bucket structure. Figure 3(a) depicts the structure of a main/stash bucket. We set the bucket size to 256B, DCPMM's internal data transfer size. There are a 16B header and 15 entry slots, each of which can contain either a 16B (key, value) or a 16B (key, pointer) entry. The header is composed of a 16-bit bitmap and 14 1-byte fingerprints. We follow previous studies [32, 33, 38, 52] to use fingerprints (i.e. a 1B hash code of a key) to accelerate the search in a bucket. Note that given the 256B bucket size, the layout is non-ideal in that we have to use only 14 fingerprints to support 15 entry slots. (The non-ideal layout will be discussed later in this section). The 16-bit bitmap is designed to handle two cases:

- If the least significant bit is 1, then all 15 entries in the bucket are (key, value) entries. The rest of the 15 bits indicate whether the corresponding slots are occupied or empty.
- If $k > 0$ least significant bits are 0 and the next bit is 1, then the first k slots in the bucket contain (key, pointer) entries for popular duplicate keys, and the rest $15 - k$ slots contain (key, value) entries. The $15 - k$ most significant bits in the bitmap record the occupied/empty status of the (key, value) slots.

Data-aware adaptation. Figure 3(b) illustrates the adaptation of the bucket structure as entries are inserted. Note that when an insert to a bucket fails, Pea Hash triggers the bucket adaptation.

In ①, insertion proceeds as normal before the bucket is full. Pea Hash supports the entry moving method [32] for faster insertion as will be discussed in Section 3.4. Here, the insert of (4,d) triggers the moving of entry (1,a), (2,b), and (3,c) to the second 64B line in the bucket so that the next insert finds an empty slot in the first line.

In ②, when an insert (1,f) sees a full bucket, Pea Hash performs a bucket compaction, which checks all entries including the pending entry to be inserted, and combines entries of duplicate keys. Here, key 1 and 2 are identified as duplicate keys. Then, for each duplicate key, Pea Hash allocates a value bucket to store all its values. It reorganizes the main bucket to contain (key, pointer) and (key, value) entries. Since the first cache line is mainly occupied by (key, pointer) entries, entry moving may not be effective and is disabled.

In ③, for the insert of (3,s), Pea Hash checks if there is a (key, pointer) entry for the key 3. Since such entry does not exist, it inserts (3,s) into the main bucket.

In ④, for the insert of (1,t), there exists a (key, pointer) entry for the key 1. Therefore, Pea Hash follows the pointer to insert the value t into the value buffer.

This process continues until the bucket is full. Then another bucket compaction operation is triggered. If no more duplicate keys are combined in the main bucket, Pea Hash considers that the segment is full under the current hashing strategy. Then it follows the adaptive hashing strategy to either transition to the next hashing strategy or perform a segment split.

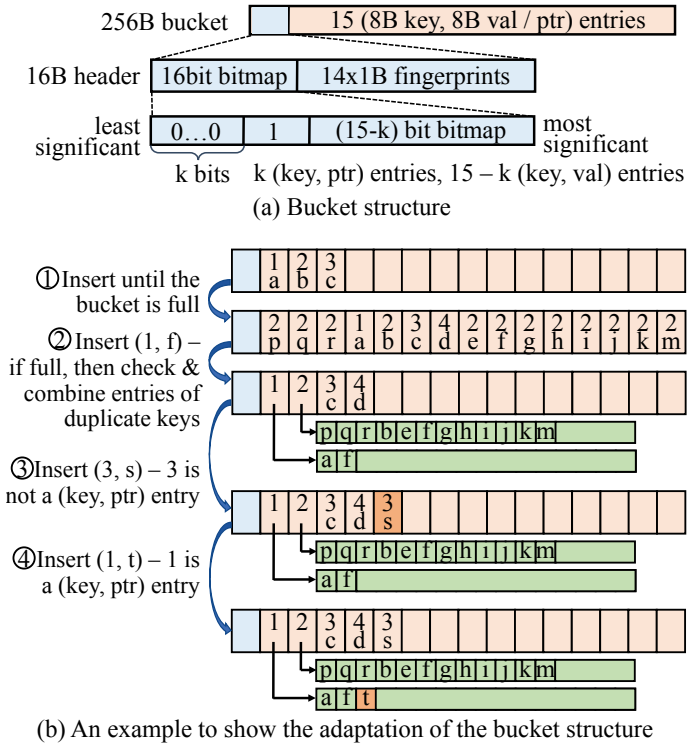


Fig. 3. Data-aware adaptive bucket.

We see that this adaptation process satisfies the two design goals. ① and ③ avoid pointer dereferences for unique keys or keys with only a few duplicates. For popular keys, ② and ④ store the (key, pointer)'s in front slots and the values in contiguous value buffers[3].

Managing value segments. We aim to place values of a popular duplicate key in a contiguous value buffer. To avoid frequent space allocation calls, we design 12 types of value segments as shown in Figure 1(a). A type-k (k=0, ..., 10) segment is divided into value buffers of $2^k \times 256$B each. That is, a type-0 segment consists of 2048 buffers of 256B each. A type-10 segment consists of 2 buffers of 256KB each. When a type-k value buffer is full and a new value is to be inserted, Pea Hash allocates and copies the values to a type-$(k+1)$ value buffer. The type-11 segment is special. It uses an 8B pointer to build a linked list of type-11 512KB segments in case the values of a very popular key require even larger space to store.

Coping with non-ideal layout. The hash bucket size is often determined by the underlying hardware characteristics, such as the 256B internal data transfer size in DCPMM in this paper. As a result, the bucket layout may not be ideal. In our case, there are 14 fingerprints and 15 entry slots in a bucket.

A naïve approach to getting around this problem is to simply ignore one slot and support only 14 slots. We do not take this approach because it wastes valuable space. Another approach is to compute 7-bit fingerprints rather than 1B fingerprints. However, this slows down fingerprint comparison because SIMD instructions can no longer be used.

[3]Note that during the bucket compaction operation, it is possible to leave keys with a small number of duplicates in the main bucket (e.g., key 1 in ②). However, we find that this may incur more frequent bucket compaction operations and have negative performance impact. Therefore, we employ the current simpler design.

To cope with the problem, we observe that slot 15 has the lowest probability to be occupied. Therefore, we store fingerprints for the first 14 slots and omit the fingerprint for slot 15. In most cases, the bitmap shows that slot 15 is empty, and it is filtered out. When it is occupied, Pea Hash specifically checks slot 15 in addition to the normal checks.

Supporting different key sizes. There are two possible methods. First, the bucket layout can be adjusted for different, pre-defined entry lengths. For example, for (4B key, 8B value), an entry is 12B large. A 256B bucket can hold 19 entries. In general, if the entry size is E, then the number of entries = $\lfloor \frac{256}{E+1.125} \rfloor$. Second, we can store (8B hash code of key, 8B pointer) in the bucket, where the pointer refers to the actual (key, value) entry stored outside the hash table. The first method is more suitable for small fixed sized keys, while the second can flexibly support large variable sized keys.

3.4 NVM-optimized Pea Hash Index

NVM main memory [1, 7, 13, 46] is capable of supporting significantly larger memory capacity than DRAM. However, compared to DRAM, NVM has lower read and write bandwidth, and persisting data from CPU cache to NVM (e.g., with `CLWB` and `SFENCE`) is significantly slower than normal writes [31, 49]. Therefore, our NVM-optimized design aims to reduce the number of NVM accesses, especially persist operations as much as possible.

The index operations in Pea Hash are similar to previous hashing indices (e.g., CCEH [37] and Dash [33]) that are based on extendible hashing. The main difference is how the two optimization techniques are supported. Therefore, in the following, we first review existing techniques for NVM-optimized index designs that we employ. Then, we focus on the NVM persistence design for the two proposed optimization techniques. After that, we discuss NVM space management and concurrency control. Finally, we describe how to recover the index from a crash.

Exploiting existing techniques for NVM performance. We learn from previous NVM-persistent index studies, including hashing indices [15, 33, 37, 52], B+-Trees [2, 11, 12, 32, 38, 50], and radix trees [28, 29, 34], and emphasize the following techniques:

- *Selective Data Persistence*: We only need to persist data that are critical to the consistency of the index. For example, B+-Tree performance can be improved by storing the non-leaf nodes in DRAM because the non-leaf nodes can be reconstructed from the leaf nodes [32, 38, 50]. In our case, we store the auxiliary structures in DRAM as shown in Figure 1. Specifically, the locks will be cleared upon recovery. The bitmap of allocated segments, FESid, and the number of stash segments can be reconstructed by examining the directory, the StashSid array, and the value bucket manager in NVM.
- *NVM-Atomic Writes*: Since write-ahead logging (WAL) incurs NVM write amplification because data are written both to the target location and to WAL, previous NVM-optimized indices exploit NVM atomic writes to avoid WAL [2, 12, 28, 29, 32–34, 37, 38, 50, 52]. We follow this principle to perform NVM atomic writes for all index operations except the bucket compaction operation (which will be described later in this subsection).
- *Entry Moving in a Bucket*: We employ entry moving for the hash buckets. This technique is first proposed in LB+-Tree [32] to reduce NVM persist operations for a 256B B+-Tree leaf node. An insert to a leaf node looks for the first empty slot to store the new entry, then updates the node metadata. In the good case, both the slot and the metadata are in the first 64B line of the node. Then one persist suffices. In the bad case, they are in different lines and require two persist operations. Without entry moving, the bad case is more often as the first line is soon filled. When encountering the bad case, the entry moving technique creates empty slots in the first line by moving as many entries from the first line to the line with the new entry. In this way, a later insert sees the good case, saving one persist for the insert.

Persistence for adaptive hashing strategy: strategy transition. A strategy transition does not need to move any existing index entries in a main segment. Typically, the only NVM write is to update the hashing strategy in the 8B directory element. This can be persisted with a single NVM atomic write.

Infrequently, segment ID/1024 may exceed the current capacity of `StashSid[]` as indicated by the counter in DRAM during a transition to 2-choice+stash. In such cases, Pea Hash allocates, initializes, and persists new stash segments. Then, it populates and persists the new stash segment IDs in `StashSid[]`.

Persistence for adaptive hashing strategy: segment split. When an insertion fails under 2-choice+stash, Pea Hash splits the main segment into four segments. It allocates, populates, and persists the new segments, then modifies the directory. There are two cases.

(1) Directory update: If there are $2^{G-L} \geq 4$ directory elements for the old segment, Pea Hash performs directory update using two steps. Step 1 updates all the relevant directory elements except the lowest element, then persists the writes. Step 2 performs an NVM atomic write to update the lowest element. During recovery, Pea Hash copies the lowest element across all 2^{G-L} elements. This discards the new segments if a crash occurs during directory update.

(2) Directory expand. If $2^{G-L} < 4$, Pea Hash first expands the directory, then performs directory update. As shown in Figure 1(b), there are two directory regions. Pea Hash computes and persists the new, expanded directory in the unused region. Then it performs an NVM atomic write for the associated 8B global depth. During recovery, Pea Hash decides which directory is the latest by comparing the two global depths and choosing the larger.

Note that the directory region capacity is computed based on a predefined maximum hash table size. In case this capacity is exceeded, we can allocate a directory in the NVM segment space and record its starting address in the 8B global depth, using bit 63 to determine if the 8B contains a global depth or an address.

Persistence for data-aware adaptive buckets: bucket compaction. Since bucket compaction is performed in place, we employ a small WAL in the value segment manager to guarantee crash consistency. Note that Pea Hash supports all other index operations with NVM atomic writes without logging.

NVM space management. We design a lightweight NVM space manager for Pea Hash. As shown in Figure 1, it organizes the main NVM space in 512KB segments to reduce NVM space allocation overhead. All segments are aligned at 256B boundary.

We use a segment bitmap and FESid in DRAM to accelerate segment allocation. A bit is set in the bitmap if the corresponding segment is used. FESid roughly indicates the first empty segment in the bitmap. It is updated after allocation/deallocation without atomic operations, which is good enough for performance improvement purpose. To find an empty segment, the NVM space manager scans the segment bitmap starting from FESid. This reduces the scan cost compared to a full scan of the bitmap. Once a 0 bit is detected, the manager allocates the segment by atomically setting the bit in the bitmap. If the scan fails, it tries the full bitmap scan. Both the segment bitmap and FESid are rebuilt during recovery.

The other two NVM space management related structures are the StashSid array and the value bucket manager. The former records all the allocated stash segment IDs. The latter is composed of a list of allocated value segment IDs, the headers of the free lists for type-k (k=0, ..., 11) value segments, and a small log per thread to support bucket compaction.

Concurrency control. We follow Dash [33] to use optimistic locks for concurrency control. We place the concurrency control structures in DRAM to reduce NVM writes, as shown in Figure 1.

The lock table in DRAM consists of locks to protect the segments and locks to protect the buckets in the segments. For each segment, there is one segment lock entry and b bucket lock entries (b=8 in our experiments). A lock entry takes 4 bytes. Since the segment size is 512KB, the space overhead of the lock table is $\frac{4(b+1)}{512KB} = 0.007\%$.

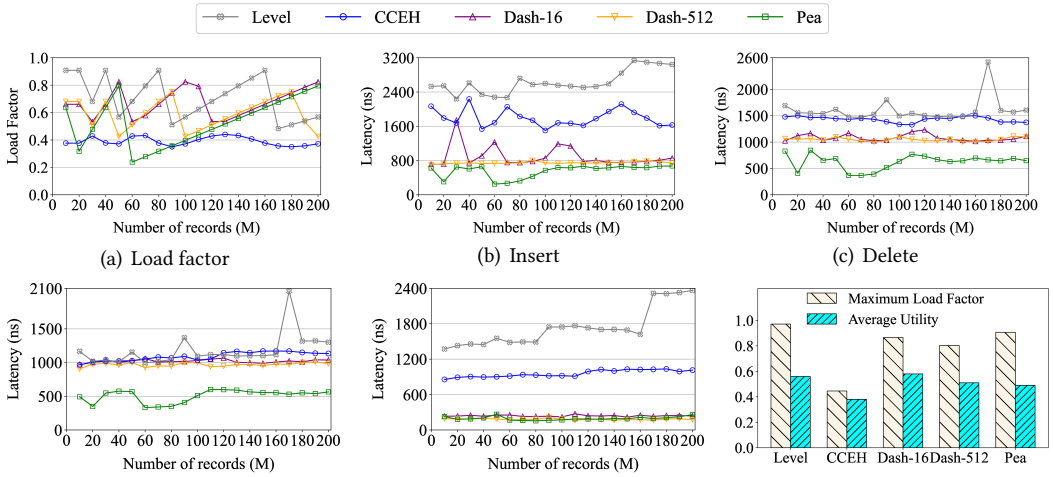
To protect concurrent accesses to a bucket, we map the bucket ID in the segment to one of the b bucket lock entries using the modulo operation. A 32-bit lock entry is composed of 8-bit thread ID, 8-bit reference count, and 16-bit version. A non-zero reference count tells that the bucket is locked. The thread ID is used to support the case where two buckets mapping to the same lock entry are to be visited in two-choice hashing. For an insert/delete, we lock a bucket by atomically setting the thread ID, increasing the reference count, and increasing the version. For a search, we ensure that the bucket is unlocked, and we compare the before and after versions to make sure that the bucket is not changed during the search. If the bucket is locked or if the before and after versions are inconsistent, then we will retry the search from the beginning. Since $\#threads \ll b \times \#segment$ (e.g., $\#segment$=8192 in our experiments), the probability of false sharing (i.e., two buckets associated with the same lock entry are accessed by two concurrent threads) is low.

We use segment locks to protect concurrent accesses to segments. A segment lock entry contains a lock bit and a version. For a search/insert/delete, we first take the fast code path assuming that the operation does not modify the segment metadata. We compare the before and after versions for such normal operations. If we find that an insert fails and it is about to trigger segment metadata updates (e.g., strategy switch and segment split), we retry the insert from the beginning. Note that at this moment, we have not yet modified any buckets or segment metadata. For the retry, we take the slow code path to acquire the corresponding segment lock by atomically setting the lock bit and increasing the version to protect the segment metadata updates.

The directory lock is used for directory expand operations. It contains a lock bit, a version bit, and a bit to identify which of the two directories is currently in use. Similar to the segment locks, normal directory reads only check the version without obtaining the lock. Only the slow code path takes the directory lock. Interestingly, we can view the directory, segment, and bucket structure as a multi-level structure, and employ the classical lock coupling technique in the slow code path.

Recovery. During recovery, Pea Hash selects the latest directory by comparing the global depths. Then, it scans the directory, the StashSid array, and the value bucket manager to populate the in-DRAM auxiliary structures. The scan of the directory is stridden [37]. Suppose the current element is i (i=0 at the beginning). If element i's local depth is L, then then next 2^{G-L} elements are associated with the same main segment. The scan copies element i to element i+1, ..., i+2^{G-L}-1 to deal with crash during directory update. Then it sets i=i+2^{G-L} for the next stride. The largest segment ID with 2-choice+stash determines the number of the stash segments. If the logs in the value bucket manager are not empty, they record states for bucket compactions. Pea Hash recovers the buckets by undoing any changes. This cost is small because the number of concurrent bucket compaction is bounded by the number of threads. Overall, Pea Hash can recover instantly in a few ms (cf. Section 4).

Intel Optane DCPMM vs. NVM-generic features. The designs in this section are generally pertinent for any NVM implementation. For Intel Optane DCPMM in our experimental machine, we set the bucket size to 256B. This can be adjusted to optimize for different internal data transfer sizes in other future NVM implementations.



(a) Load factor (b) Insert (c) Delete

(d) Positive Search (e) Negative Search (f) Memory utilization for 200M records

Fig. 4. Comparison with state-of-the-art NVM-optimized hashing indices (single thread).

4 EVALUATION

We describe the experimental setup in Section 4.1. Then, we evaluate the overall performance of NVM-optimized and DRAM-based Pea Hash in Section 4.2, and study the benefits of the two proposed optimization techniques in Section 4.3.

4.1 Experimental Setup

Machine configuration. We run experiments on a server equipped with two Intel Xeon Gold 5218 CPUs (2.3GHz, 16 cores/32threads, 22MB L3 cache), 2×192GB DRAM, and 2×768GB Optane DCPMM in App Direct mode. The server runs Linux 4.15 and PMDK 1.7. All the code is compiled using GCC 7.5 with all optimization enabled. Experiments are run on a single CPU socket with its associated NVM and DRAM to avoid NUMA effects.

Solutions to compare. We compare NVM-optimized Pea Hash with three NVM-optimized hashing indices: 1) CCEH [37], 2) Level Hash [52], and 3) Dash [33]. We obtain Dash from its github repository (https://github.com/baotonglu/dash). Since the original CCEH and Level Hash implementations are based on DRAM emulation [37, 52], we use the versions found in the Dash repository, which are ported to run on DCPMM using PMDK. Pea Hash uses 512KB segments and 256B buckets. We use the parameters of CCEH, Level Hash, and Dash in their original papers [33, 37, 52] for fairness. Specifically, Dash (denoted *Dash-16*) uses 16KB segments and 256B buckets. Each segment of Dash has two stash buckets. CCEH uses 16KB segments and 64B (one cache line) buckets. Level Hash uses 128B buckets. To explore the effect of various segment sizes, we also evaluate a variant of Dash with 512KB segments (denoted *Dash-512*). We choose Dash because it outperforms CCEH and Level Hash.

We compare the DRAM-based Pea Hash with 1) Level Hash, which has shown better in-DRAM performance than previous hash tables (e.g., BCH [19]), 2) CLHT [14], a CPU cache optimized hash table, 3) CCEH, 4) Dash-16, and 5) Dash-512. We obtain the DRAM-based Level Hash (https://github.com/Pfzuo/Level-Hashing) and CLHT (https://github.com/LPD-EPFL/CLHT) from github. Both CLHT and Level Hash use cache-line-sized 64B buckets, following their default source code for fairness. CLHT resizes to 4x its original size when on average there are 1.5 buckets in its linked lists. The resulting maximum load factor is 55.4%. For 3)–5), we remove the NVM features of CCEH and Dash.

Data sets. Both keys and values are 8B integers. For unique key experiments, we follow previous work [33] to generate uniformly distributed keys. For duplicate key experiments, we use both synthetic data sets following the zipfian distribution, and real-world graph data sets obtained from SNAP [45], as shown in Table 2.

Table 2. Graph data sets used in duplicate key experiments.

Name	Type	Nodes	Edges
wiki-Talk	Directed	2394385	5021410
cit-Patents	Directed	3774768	16518948
soc-LiveJournal	Directed	4847571	68993773
com-Orkut	Undirected	3072441	117185083

4.2 Overall Performance

NVM-optimized Pea Hash: single-threaded performance. We initialize the hash tables to be roughly 5MB large, then insert 10M index entries so that the hash tables naturally grow into a steady state before running the actual experiments. (We use "M" as an abbreviation for million for concise presentation). Then, we insert 190M random entries into the hashing indices. After inserting every 10M entries, we measure the average latency of 100 random point operations: 1) insert a new entry; 2) delete an existing entry; 3) positive search for existing entries; and 4) negative search for nonexistent entries.

Figure 4(a) reports the instantaneous load factors as the index sizes grow. The curves show sawtooth shapes. The load factors grow as the index sizes increase until resizing. After resizing, the load factors drop drastically. We see that Pea Hash attains similar maximum load factors compared to Level and Dash, while CCEH has poor memory utilization.

Figure 4(b)–(e) show the average latencies of the four index operations as the index sizes grow. There are interesting correlations between the latency curves and the load factor curves. When the load factors are low, the hash tables often have good performance (which is quite clear at 20M and 60M points for Pea Hash). This verifies the conflict between memory utilization and performance. Pea Hash exploits adaptive hashing strategy to mitigate the problem. We see that Pea Hash outperforms the other hash tables for all four operations in most cases. Compared to Dash-16/Dash-512/CCEH/Level, Pea Hash achieves 1.1–4.9x/1.1–2.9x/2.3–7.6x/3.4–9.1x speedups for insert, respectively. Similarly, Pea Hash accelerates deletion by 1.2–3.2x/1.3–2.9x/ 1.7–3.9x/1.8–4.0x, and positive search by 1.7–3.2x/1.6–2.8x/1.8–3.2x/1.8–3.9x. For negative search, Pea Hash achieves 3.4–6.0x/5.8–12.0x speedups compared to CCEH/Level. Compared to Dash-16/Dash-512, Pea Hash has similar negative search latencies.

Comparing the four types of operations, we see that insertion and deletion often take longer than search because they perform NVM writes and persists. Negative search in Pea Hash is faster than positive search because negative search often stops after checking the bucket header and hence read fewer cache lines.

Memory utilization. Figure 4(f) plots the maximum load factor and average utility of the hashing indices. As defined in Section 2.2.2, average utility captures the dynamic behavior of the hash tables. From the figure, we see that the adaptive hashing strategy in Pea Hash achieves similar maximum load factor and average utility compared to Level and Dash.

CCEH's memory utilization is significantly lower. CCEH employs linear probing with bounded probe distance, while the other three hash tables all employ some form of 2-choice hashing, which provides higher memory utilization.

Figure 4(f) mainly considers NVM space utilization. As for DRAM footprints, we find that all the persistent hash tables barely use DRAM. We consider the DRAM space overhead for a Pea

Table 3. Resizing operations during 200M inserts.

	Level	CCEH	Dash-16	Dash-512	Pea
Count	**11**	525328	262301	16367	2728
Min Lat	85.8ms	**44.6us**	46.1us	1.78ms	2.76ms
50% Lat	2.79s	**62.7us**	65.3us	1.99ms	4.97ms
90% Lat	44.5s	**80.9us**	88.4us	4.96ms	5.15ms
99% Lat	89.4s	148us	**101us**	5.22ms	5.37ms
Max Lat	89.4s	25.1ms	10.6ms	7.12ms	**6.17ms**
Total time	178s	45.2s	24.0s	43.4s	**12.4s**

Hash table containing 200 million 16B entries. The hash table takes at least 3.2GB of NVM to store the index entries. After inserting 200 million entries, there are 8192 segments. As described in Section 3.4, there is a segment lock and b=8 bucket locks per segment. Each lock entry takes 4 bytes. Hence, the size of the lock table is $8192 \times (8 + 1) \times 4$ = 295KB. The rest of the auxiliary data structure in Figure 1 takes less than 10KB. Hence, DRAM space/NVM space $\leq \frac{295KB+10KB}{3.2GB} = 0.009\%$. The DRAM space overhead is negligible.

Performance vs. memory utilization. Figure 4(b)–(e) show that compared to state-of-the-art NVM-optimized hash tables, Pea Hash improves performance. This is because adaptive hashing strategies allow the use of simpler and more efficient hashing strategies for different load factors. Figure 4(f) shows that Pea Hash maintains good average utility. Taking Figure 4 as a whole picture, we conclude that Pea Hash has achieved the goal of optimizing performance while maintaining good memory utilization.

Resizing latency. Table 3 shows the statistics about resizing operations (which incur rehashing, segment splits, and/or directory expansion) during the single-threaded insert experiments in Figure 4(b). First, CCEH, Dash, and Pea employ extendible hashing to amortize the resizing overhead across multiple segments. Compared to Level, we see that they perform much higher numbers of resizing operations, but the latency of individual resizing is much lower. Second, segment sizes have significant impact on resizing operations. Compared to Dash-16 with 16KB segments, Dash-512 and Pea with 512KB segments see smaller count but higher cost of resizing operations. Third, Pea Hash achieves the smallest total resizing time because of the optimized segment split and directory expansion implementation. Its resizing latency is 2.76–6.17ms, which is good even for interactive queries.

Scalability. Figure 5 compares the NVM-optimized hashing indices while increasing the number of threads from 1 to 16. We initialize the hash tables to be roughly 5MB large, then insert 10M index entries before running the actual experiments. For Figure 5(a)-(d), we perform four experiments in the following order (a) 190M inserts; (b) 190M positive searches; (c) 190M negative searches; and (d) 190M deletes. For Figure 5(e)-(f), we perform 190M mixed operations. The figures report operation throughput. The higher the better. According to Figure 4(a), after inserting 10M+190M index entries, all hash tables except Dash-512 are under relatively high load factors (compared with their maximum load factors). Specifically, search and deletion of Pea Hash are tested under the most sophisticated strategy, i.e. 2-choice with stash.

We see that Pea Hash scales significantly better than the other hashing indices. Using 16 threads, Pea Hash improves insertion, deletion, and positive search throughput by 1.13x–13.8x compared with Level, CCEH, Dash-16, and Dash-512. For positive search with 16 threads, Dash supports ~41 Mops/s. Pea Hash improves this to ~48 Mops/s. While 48/41=1.17 looks small, we believe this is quite significant improvement. For negative search with 16 threads, Pea Hash outperforms Level and CCEH by a factor of 3.6x and 5.1x, respectively. Compared to Dash-16 and Dash-512, Pea Hash

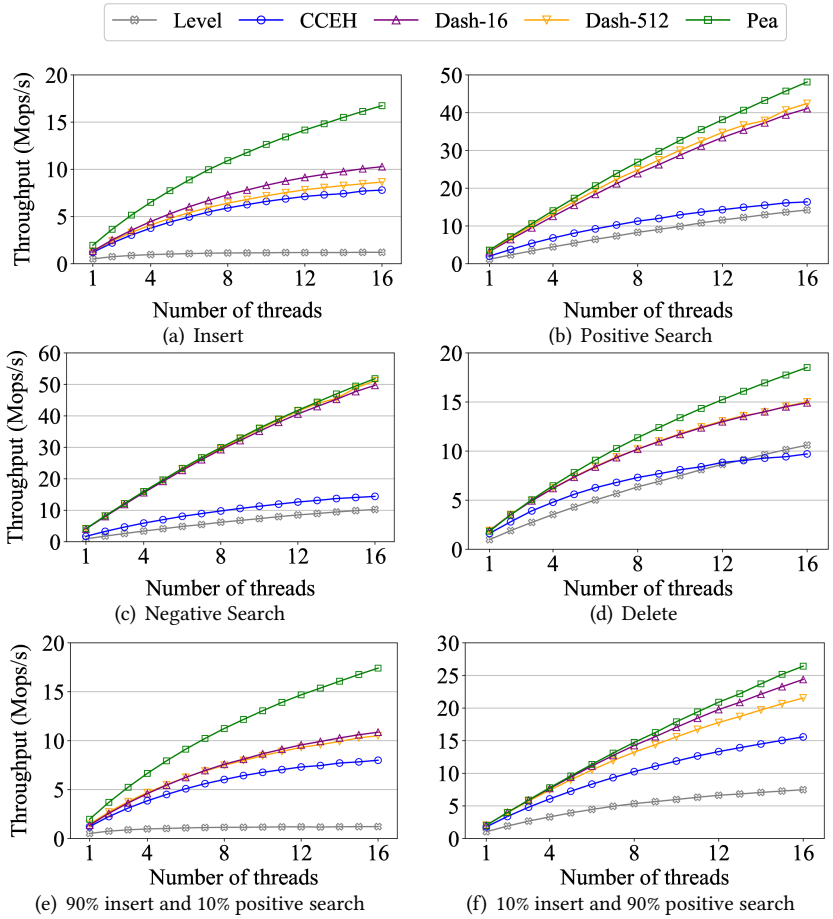


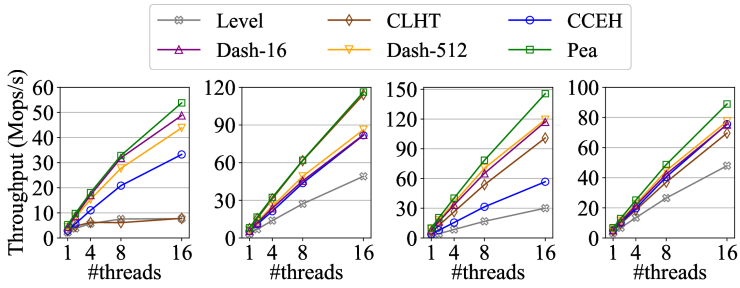
Fig. 5. Comparison with state-of-the-art NVM-optimized hashing indices varying the number of threads.

Table 4. Recovery time of NVM-optimized hashing indices.

Index entries		Level	CCEH	Dash-16	Dash-512	Pea
100M	total	35.0ms	140.4ms	34.7ms	34.9ms	2.8**ms**
	logic	< 1ms	106.8ms	< 1ms	< 1ms	< 1**ms**
200M	total	35.3ms	253.8ms	35.4ms	35.4ms	2.7**ms**
	logic	< 1ms	217.7ms	< 1ms	< 1ms	< 1**ms**
300M	total	36.0ms	302.8ms	36.3ms	38.0ms	3.0**ms**
	logic	< 1ms	268.2ms	< 1ms	< 1ms	< 1**ms**
400M	total	39.0ms	480.6ms	35.2ms	38.9ms	2.9**ms**
	logic	< 1ms	435.2ms	< 1ms	< 1ms	< 1**ms**

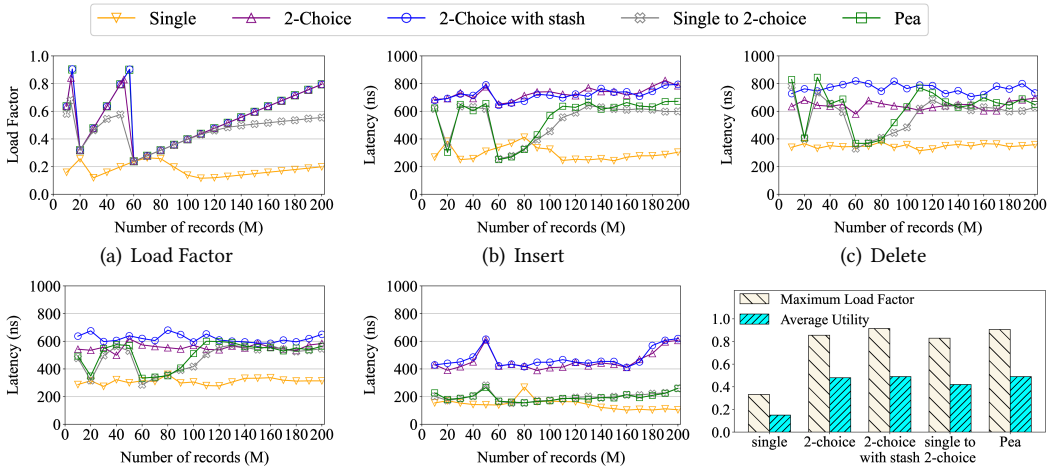
is similar or slightly better. For the mixed workloads, Pea Hash achieves the best performance among all hashing indices. In summary, Pea Hash scales well for all kinds of workloads.

Recovery time. We allocate 30GB NVM memory for each NVM-optimized index. After loading 100M, 200M, 300M, or 400M 16-byte entries, we kill the process. Then we rerun the code to measure the time until the system starts handling requests. The recovery is performed by a single thread.



(a) Insert (b) Positive search (c) Negative search (d) Delete

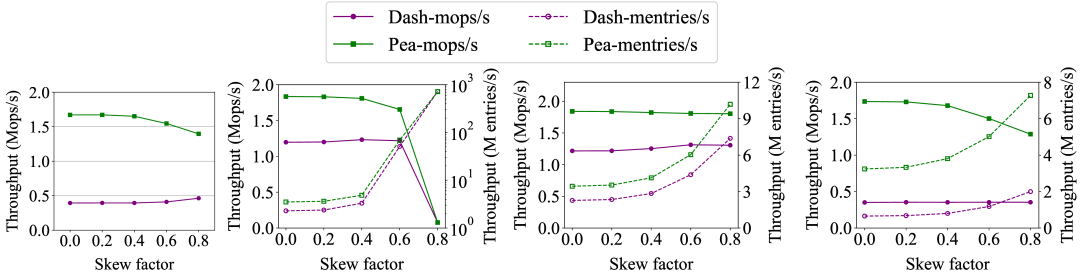
Fig. 6. Comparison with DRAM-based hashing indices.



(a) Load Factor (b) Insert (c) Delete

(d) Positive Search (e) Negative Search (f) Memory utilization for 200M records

Fig. 7. Comparison of different hashing strategies (single thread).



(a) Insert skew data (b) Skew search on skew data (c) Unique search on skew data (d) Unique delete on skew data

Fig. 8. Performance varying the skew factor of key's zipf distribution (single thread).

Table 4 reports both the total recovery time and the time to recover the hash table structures (denoted *logic*). The difference of the two is spent in initializing/recovering NVM allocation structures.

As shown in Table 4, the recovery time of Pea Hash is at least one order of magnitude lower compared to the other designs. Pea Hash recovers in about 3ms in total, which is fast enough for most practical use scenarios. We find that our lightweight NVM space manager recovers faster than the epoch-based NVM allocator used in Dash. The allocator of Dash spends about 35ms recovering

its epoch manager and garbage list, which dominates the recovery time of Dash and Level. In contrast, the NVM space manager of Pea Hash recovers in about 2ms.

Moreover, the logic recovery procedure of Pea Hash is also lightweight. It reads the directory and the metadata in NVM to reconstruct the auxiliary structure in DRAM. Only bucket compactions perform logging, and all other normal operations perform NVM atomic writes. Hence, the only data to recover are the buckets being compacted. Recovering these buckets is fast because the number of concurrent bucket compactions is bounded by the number of threads. Compared to the logic recovery time of Pea Hash, Level and Dash, CCEH spends over 100ms recovering its hash table. It suffers from scanning the segment metadata for checking whether the crash occurs during a segment split. The scan visits the segment metadata stored along with the allocated buckets, incurring a large number of random NVM accesses.

DRAM-based Pea Hash. Figure 6 reports the throughput of DRAM-based hashing indices varying the number of threads. The workload is the same as Figure 5(a)-(d). For insertion, deletion, and negative search, Pea Hash outperforms the other hash indices by 1.1x–4.9x with a single thread, and 1.1x–7.0x with 16 threads. For positive search, Pea Hash achieves 2.3x/1.02x/1.4x/1.4x/1.3x speedups compared with Level/CLHT/CCEH/Dash-16/Dash-512. As for memory utilization, CLHT can store at most three 16-byte entries in every 64B bucket. The actual maximum load factor in the experiments is 55.4%, and average utility is 29.8%. In comparison, Pea Hash achieves good performance without compromising the memory utilization.

4.3 Benefits of Individual Techniques

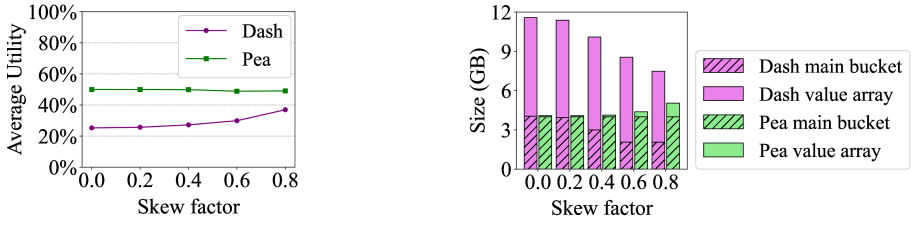
Adaptive hashing strategy. Figure 7 compares 1) the three pure hashing strategies, i.e., single hashing, 2-choice hashing, and 2-choice with stash, in the sequence of Pea Hash's adaptive hashing strategy, 2) an adaptive strategy consisting of single and then 2-choice, and 3) Pea Hash's adaptive strategy. All implementations are based on the NVM-optimized Pea Hash using the same configuration of bucket and segment sizes. We repeat the same experiments as in Figure 4.

Figure 7(a) reports the load factors as the hash tables grow, and Figure 7(f) shows the maximum load factor and average utility of five schemes. We see that single hashing has the lowest memory utilization. Single to 2-choice improves the average utility of single hashing by over 2x. The adaptive transition effectively improves memory utilization. Interestingly, single to 2-choice's memory utilization is slightly worse than pure 2-choice. This is because at the strategy transition time, the buckets resulting from the single-hashing are less balanced than those in pure 2-choice. Adding a third strategy as in Pea Hash leads to better memory utilization. The stash buckets help tolerate imbalanced and overflowed buckets, and avoid premature rehashing.

Figure 7(b)-(e) compare the operation latency for the five hashing strategies as the hash tables grow. Among the three pure hashing strategies, single hashing has the lowest latency, which is around half of the latencies of pure 2-choice and 2-choice-with-stash. The latency of adaptive strategies follow the fluctuation of load factors. For insertion, deletion, and positive search, the performance of adaptive strategies is similar to the respective pure strategies currently in use. As for negative search, the performance of pure 2-choice and 2-choice-with-stash are poor, since they have to probe 2 and 4 candidate buckets, respectively.

Overall, we see that the three pure strategies have different trade-offs between memory utilization and performance. The two adaptive strategies, especially Pea Hash, effectively reduce the operation latency without sacrificing high memory utilization.

Data-aware adaptive buckets: duplicate key support. We test the data-aware adaptive bucket optimization for duplicate keys on NVM. As Dash performs better than CCEH and Level hash, we compare Pea Hash with Dash for this set of experiments. We use default Dash configuration (i.e.



(a) Average utility of hash tables (b) Memory used after insertion

Fig. 9. Memory utilization for zipf distributed keys.

Dash-16) and modify it to support duplicate keys by storing (key, pointer) entries in the buckets, where the pointer points to a buffer of values for the same key. The initial buffer has a capacity of 4 values. It doubles its size when there is no room for an incoming insertion of the same key. We test the operations of inserting a key-value pair, and searching/deleting all values of a given key. A search copies all values of a given key from NVM to a value array in DRAM. Obtaining all values of a given key is frequently used in secondary indices and in hash-based query processing algorithms in database systems.

After inserting 10M entries in a 5MB hash table, we insert 190M zipfian distributed key-value pairs. As shown in Figure 8, the X-axis shows the skew factor of the Zipfian distribution varying from 0.0 to 0.8. The Y-axes show throughput in the number of operations per second (Mops/s) and the number of entries retrieved or deleted per second (M entries/s). The two are different since a search/delete returns/removes all the entries of a given key. Figure 8(a) shows the throughput of inserting 190M skew entries. Figure 8(b) shows the throughput of search on the inserted data. The distribution of search keys is the same as the insert keys. Figure 8(c) shows the performance of searching unique keys on inserted skew data. The X-axis refers to the skew factor of inserted keys. Figure 8(d) reports the performance of deleting unique keys on inserted skew data. Both search and deletion keys are shuffled randomly to avoid any influence of the spatial locality because of the key order.

For insertion and deletion, Pea Hash outperforms Dash by a factor of 3.0–4.9x. Pea Hash places as many pairs as possible in main buckets, thereby reducing the overhead of NVM allocation. Skew search tends to emphasize the search for popular duplicate keys, while unique search treats both frequent and infrequent keys similarly. Search throughput of Pea Hash is 1.4–1.5x higher than that of Dash in all search experiments except skew search with 0.8 skew factor. The improvement is mainly because a large fraction of search in Pea Hash visits only the main bucket without pointer dereference. In the case of skew search with 0.8 skew factor, this advantage weakens as a small number of popular keys tend to be searched, and temporal locality dominates the throughput. This is why skew search of both hash tables show similar results.

We evaluate memory utilization by calculating the average utility of each hash table during the insertion experiments varying skew factor from 0.0 to 0.8. As shown in Figure 9(a), the average utility of Pea Hash is around 50% and is quite stable for different skew factors, while the average utility of Dash increases from 25% to 37%. Figure 9(b) break downs the memory used after the insertion process. Each key in Dash occupies a slot in a main bucket and a value array, resulting in higher NVM consumption. The memory utilization of Pea Hash is consistent regardless of the skew factor, since Pea Hash balances the number of duplicate keys in each main bucket. The data-aware adaptive bucket design saves redundant pointers for unique keys or keys with only a few duplicates, bringing benefit to both performance and memory utility.

Figure 10 shows the performance on the four real-world graph data sets. We perform four experiments. 1) **SI** (insert skew data): insert all (src-node, dest-node) entries into an empty 5MB

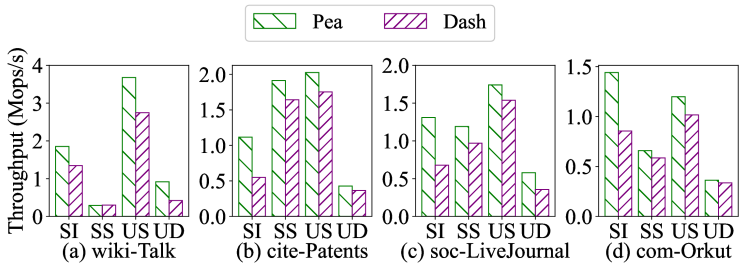


Fig. 10. Performance for graph datasets.

hash table; 2) **SS** (skew search on skew data): search, where the distribution of search keys is the same as the inserted keys; 3) **US** (unique search on skew data): search for unique keys; and 4) **UD** (unique delete of skew data): delete unique keys. Overall, Compared with Dash, Pea Hash achieves 1.8x/1.1x/1.2x/1.5x speedups on average for skew insert / skew search / unique search / unique delete, respectively. In several cases, we see very significant speedups, e.g., 2x for skew insert on cite-Patents, 1.9x for skew insert on liveJournal.

5 CONCLUSION

In this paper, we propose Pea Hash, a novel performant extendible adaptive hashing index. We identify the conflict between performance and memory utilization, and propose adaptive hashing strategy to address the problem. We design data-aware adaptive buckets to efficiently support duplicate keys. Experiments on real Intel Optane DCPMM show that both NVM-optimized Pea Hash and DRAM-based Pea Hash index achieve significantly better performance than prior state-of-the-art hashing indices, while maintaining desirable memory utilization. In conclusion, we believe that Pea Hash is a promising hashing index solution.

ACKNOWLEDGMENTS

This work is partially supported by Natural Science Foundation of China (62172390). Shimin Chen is the corresponding author.

REFERENCES

[1] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 9, 2 (2013), 1–35.

[2] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.

[3] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1999. Balanced Allocations. *SIAM J. Comput.* 29, 1 (1999), 180–200.

[4] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (2013), 85–96.

[5] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (2021), 1544–1556.

[6] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 281–294.

[7] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. 2010. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena* 28, 2 (2010), 223–262.

[8] J. L. Carlson and S. Sanfilippo. 2013. *Redis in action.* Manning.

[9] Larry Carter and Mark N. Wegman. 1979. Universal Classes of Hash Functions. *J. Comput. Syst. Sci.* 18, 2 (1979), 143–154.

[10] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood Hashing (Preliminary Report). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. 281–288.

[11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 21–31.

[12] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.

[13] Rob Crooke and Mark Durcan. 2015. A revolutionary breakthrough in memory technology. *Intel 3D XPoint launch keynote* (2015).

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 631–644.

[15] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2015. Revisiting Hash Table Design for Phase Change Memory. *ACM SIGOPS Oper. Syst. Rev.* 49, 2 (2015), 18–26.

[16] Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.* 380, 1-2 (2007), 47–68.

[17] A. I. Dumey. 1956. Indexing for Rapid Random-Access Memory Systems. *Computers and Automation* 5, 12 (1956), 6–9.

[18] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. 1979. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)* 4, 3 (1979), 315–344.

[19] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, Nick Feamster and Jeffrey C. Mogul (Eds.). USENIX Association, 371–384.

[20] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. *Database systems - the complete book (2. ed.)*. Pearson Education.

[21] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

[22] Guy Harrison. 2015. *Next generation databases: NoSQL, newSQL, and big data.* Apress.

[23] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5218)*, Gadi Taubenfeld (Ed.). Springer, 350–364.

[24] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1049–1063.

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[26] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (2009), 1543–1561.

[27] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

[28] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*. 257–270.

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

[30] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 294–303.

[31] Jihang Liu and Shimin Chen. 2020. Initial experience with 3D XPoint main memory. *Distributed Parallel Databases* 38, 4 (2020), 865–880.

[32] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[33] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.

[34] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. 1–16.
[35] W. D. Maurer and Ted G. Lewis. 1975. Hash Table Methods. *ACM Comput. Surv.* 7, 1 (1975), 5–19.
[36] Memcached. 2018. Memcached. https://memcached.org/.
[37] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.
[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. 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 Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 371–386.
[39] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2161)*, Friedhelm Meyer auf der Heide (Ed.). Springer, 121–133.
[40] Rina Panigrahy. 2005. Efficient hashing with lookups in two memory accesses. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*. SIAM, 830–839.
[41] W. Wesley Peterson. 1957. Addressing for Random-Access Storage. *IBM J. Res. Dev.* 1, 2 (1957), 130–146.
[42] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems (3. ed.)*. McGraw-Hill.
[43] Redis. 2018. Redis. https://redis.io/.
[44] Pramod J. Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional.
[45] Stanford. 2022. Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/.
[46] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *nature* 453, 7191 (2008), 80–83.
[47] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. 2017. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 553–565.
[48] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2014. Characterizing Facebook's Memcached Workload. *IEEE Internet Computing* 18, 2 (2014), 41–49.
[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. 169–182.
[50] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. 167–181.
[51] Pengfei Zuo and Yu Hua. 2018. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Trans. Parallel Distributed Syst.* 29, 5 (2018), 985–998.
[52] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.

Received July 2022; revised October 2022; accepted November 2022