Towards the design of efficient hash-based indexing scheme for growing databases on non-volatile memory

Zhulin Ma, Edwin H.-M. Sha, Qingfeng Zhuge, Weiwen Jiang,
Runyu Zhang, Shouzhen Gu

PII: S0167-739X(18)33100-5
DOI: https://doi.org/10.1016/j.future.2019.07.035
Reference: FUTURE 5088

To appear in: *Future Generation Computer Systems*

Received date : 17 December 2018
Revised date : 18 March 2019
Accepted date : 16 July 2019

Highlights

- **In the design of Non-Volatile Memory (NVM) based indexes, the hash-based structure is one of the most promising candidates since it can take full advantages of byte-addressable property of NVM to perform query operations with constant time complexity.**
- **Rehash operations of hash-based indexes incur large number of NVM writes and range query operations cause drastic performance degradation.**
- **Utilizing a list of ordered nodes to store entries instead of a large one makes rehash operation affect two nodes at most when a node is full, which can reduce the overhead of rehash operations.**
- **An auxiliary structure based on the linked list which can efficiently locate nodes make range query operations manageable on hash-based strucutres.**

Towards the Design of Efficient Hash-based Indexing Scheme for Growing Databases on Non-Volatile Memory

Zhulin Ma[a], Edwin H.-M. Sha[b], Qingfeng Zhuge[b], Weiwen Jiang[c], Runyu Zhang[a], Shouzhen Gu[b]

[a]*College of Computer Science, Chongqing University of China*
[b]*School of Computer Science and Software Engineering, East China Normal University*
[c]*Department of Electrical and Computer Engineering, University of Pittsburgh*

Abstract

The index is a fundamental component in data intensive systems to accelerate data retrieval operations. In the design of Non-Volatile Memory (NVM) based indexes, the hash-based structure is one of the most promising candidates since it can take full advantages of byte-addressable property of NVM to perform query operations with constant time complexity. However, we found that the basic operation, "rehash operation", may incur a large number of write activities on NVM, which is harmful to the endurance of NVM, and will cause drastic performance degradation. Additionally, range query operations cannot be efficiently conducted on hash-based indexes. In this paper, we first investigate how to design an NVM-friendly hash-based structure with the considerations of endurance and performance issues. Then, we propose a novel indexing scheme called "Bucket Hash", which can significantly reduce the overhead caused by rehash operations and range query operations. We evaluate the proposed Bucket Hash using YCSB workloads. Compared with existing indexes, Bucket Hash achieves 40% reduction on average in the number of NVM writes, meanwhile gaining 30% improvement on timing performance.

Keywords: Hashing Data structures, Non-Volatile Memory, Index Schemes

1. Introduction

Hashing techniques provide an efficient way to implement indexes for in-memory systems. With the development of Non-Volatile Memories (NVMs) [1, 2, 3], how to efficiently design indexes in NVM-based in-memory system becomes ever critical. NVM has distinct properties over traditional memories (DRAM), such as non-volatility, large capacity, and high energy efficiency. In the design of hash-based indexes for NVM, it is important in order to minimize the number of "write activities on NVM" (NVM writes), because NVM writes not only incur long overhead but also have a destructive impact on the endurance of NVM. In this paper, we investigate how to design an efficient hash-based indexing scheme with minimum number of NVM writes.

Besides minimizing NVM writes, the overall performance is another important metric in the design of index structures. In hash-based indexes, there are two operations limiting the overall performance: rehash and range query operations. Rehash operations are used to reconstruct the index when a larger space is needed in order to store entries. Specifically, when the number of entries exceeds a threshold, "rehash" operation will be performed. It creates a new larger bucket and redistributes all entries into it. Hence, rehash operations not only incur lots of NVM writes, but also lead the system unresponsive in a specified period (called "latency") when the number of entries is large. Such long time is unacceptable to real-time applications. In this paper, our objective is to reduce the cost caused by rehash operations.

Range query operation is another bottleneck on performance in hash-based indexes. This operation is to retrieve all the entries in a given range, which is commonly required in big-data oriented applications, such as Spark [4] and CephFS [5]. However, range query operation is usually inefficiency in hash-based indexes, because entries are stored out-of-order and traversing the entire index is necessary to retrieve entries. This paper is intended to improve the performance of range query operations of hash-base structures.

In this paper, we propose a novel hash-based indexing scheme, called Bucket Hash. Bucket Hash utilizes a list of ordered nodes to store entries. In this approach, each rehash operation only affects two nodes at most when a node is full. Therefore, both latency and number of NVM writes can be reduced. In addition, for range query operations, we design an auxiliary structure based on the linked list to efficiently locate nodes. As a consequence, we can efficiently find entries in a small number of qualified nodes instead of searching the entire index.

Main contributions of this paper are listed as follows:

- We investigate the bottleneck of hash-based indexes and study how to design an efficient indexing scheme for NVM-based systems.

- We propose a novel index structure (Bucket Hash) to alleviate the overhead incurred by rehash operations.

- We propose a set of management methods based on Bucket Hash to make range operations of hash-based indexes manageable.
- We conduct comprehensive evaluations on a workstation using YCSB benchmarks to demonstrate the efficiency and effectiveness of the proposed Bucket Hash.

Experimental results show that Bucket Hash outperforms existing structures in term of timing performance and NVM writes. Specifically, Bucket Hash achieves 72%, 48% and 25% reduction on elapsed time, meanwhile achieving 61%, 45% and 31% reduction on the number of NVM writes compared with B^+-Tree, Wb^+-Tree and standard hash structure, respectively.

The rest of this paper is organized as follows. Section 2 provides the background and motivation. Section 3 and 4 present the design framework and implementation of Bucket Hash. Experimental results of Bucket Hash are shown in Section 5. Finally, Section 6 concludes this paper.

2. Background and Motivation

In this section, we first present the background of Non-volatile memory. Then we review two commonly used index structures. Finally, we describe the motivation for our work.

2.1. Background: NVM

Computer memory has been rapidly developed in recent years. A new category of memory, NVM, has attracted more and more attention in both academia and industry [6, 7]. NVM, such as PCM [1] and STT-RAM [2], (i) is byte addressable, (ii) has DRAM-like performance, and (iii) has a large cell density. A distinctive property between NVM and DRAM is the non-volatility of NVM, i.e., the data written on NVM will be persistent even when the system suffers from power failure. Hence, NVM has been regarded as a promising substitute of DRAM in memory hierarchy [8, 9, 9, 10, 11, 12].

However, NVM suffers from the limitation of write endurance, and asymmetric read/write overhead. As for the endurance of NVM, if the number of write activities in a memory cell exceeds the endurance limitation, the NVM chips are worn-out. As for asymmetric read/write, the NVM writes may be

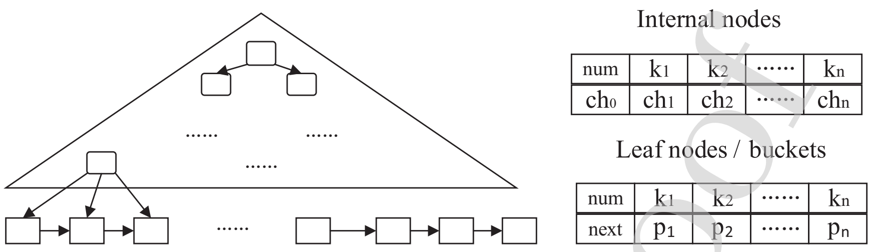

Figure 1: The structure of B$^+$-Tree

4x-20x slower than reads[13, 14], and NVM writes consume higher energy than NVM reads[15, 14].

In consequence, how to reduce the number of NVM writes is an important issue in terms of high-performance and high-endurance. In the existing work, several techniques have been proposed to reduce the number of NVM writes. From the low (circuit, hardware) level, [16] proposes Flip-N-Write technique to avoid redundant bit-writes; [15] presents the wear leveling mechanisms to evenly distribute writes on NVM; authors in [9] propose a technique to remove significant amount of redundant writes from the memory. From the high (application) level, authors in [17] present NV-tree to reduce consistency cost for NVM-based system; [18] proposes a new type of main-memory B$^+$-Tree to reduce extensive NVM writes and CPU cache flush operations. In this paper, we revisit the existing indexing structures and focus on the design of an NVM-friendly structure from the application level.

2.2. Background: existing index structures

There are two typical index structures (tree-based and hash-based structures) commonly employed in systems and we will describe them in the following texts. In this paper, we focus on the design of hash-based structures due to its fast retrieval operations.

Tree-based structures.

Tree-based structures, especially B$^+$-Tree, are widely used as the basic structure of existing indexes in file systems and database systems. The basic structure of B$^+$-Tree is shown in Figure 1. It is a hierarchical structure which consists of several nodes. Each node contains pairs of keys and pointers, called "entries". Entries in a node are required to be sorted by the keys; in this way, it can utilize binary search when locating an entry in a node.

While B$^+$-Tree is commonly used in many systems, however, when it is directly employed in NVM-based systems, it incurs large number of NVM

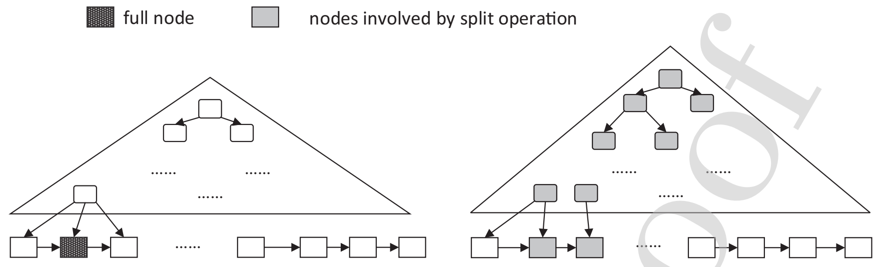

Figure 2: Effects of the split operation in B$^+$-Tree

writes due to two aspects. First, maintaining entries in order incurs large quantity of NVM writes. Because shift operations will be conducted to make entries in order when an entry is inserted or deleted. Specifically, when we insert or delete an entry, half of entries in a node will be shifted on average.

Second, maintaining structure balanced causes numerous NVM writes. In order to achieve high performance on querying, it needs to keep structure balanced. Specifically, when a node is full, it will be split into two nodes. Similarly, when the number of entries in a node is less than a threshold and the node cannot borrow an entry from its sibling, it will be merged with its sibling. In general, split or merge operations on a node will be propagated to its parent. In the worst case, it will be propagated to the root. As shown in Figure 2, the split operation on the leaf node involves $2h + 1$ nodes in total, where h is the height of the tree.

There is a lot of work which devoted to optimizing B$^+$-Tree structure in different aspects. In [19, 20, 21], authors proposed CPU cache optimized B$^+$-Tree solutions to improve the overall performance. In [22, 23], flash-friendly B$^+$-Tree is proposed to improve the endurance of flash memory. Recently, more attention has been put into the design of indexes for NVM-based systems. For instance, Venkataraman et al. [24] proposed the CDDS B-Tree, a persistent and concurrent B-Tree which relies on versions to achieve consistency. Chen et al. [12] proposed that using unsorted nodes with bitmaps to decrease the number of writes to SCM and they extended their work by proposing the write-atomic B$^+$-Tree (Wb$^+$-Tree) [18]. Yang et al. [25] proposed the NV-tree, a consistent and cache-optimized B$^+$-Tree variant with reduced CPU cacheline flush. Oukid et al. [26] proposed a hybrid SCM-DRAM persistent and concurrent B$^+$-Tree, which can achieve similar performance to DRAM-based counterparts.

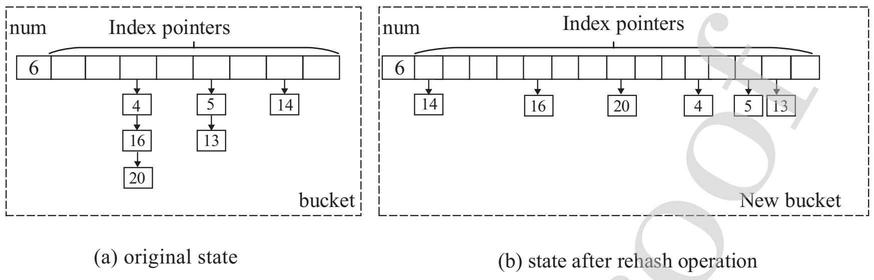

Figure 3: Example of rehash operations

Hash-based structures.

Hash-based structures are commonly employed for the applications that have high-performance querying requirements. Figure 3 (a) illustrates the structure of standard hash. Entries in standard hash are stored in a large continuous memory space, called "bucket" in this paper. The position of each entry is calculated by hash functions. Hence, there may be multiple entries that stored in one position, which we call a "hash collision" occurs. Several techniques are commonly accepted to deal with hash collisions, such as Chained hashing [27], Linear probing [28], cuckoo hashing [29]. Figure 3 (a) represents Chained hashing technique when hash collisions occur. With the increase number of hash collisions, the performance degradation is more significant. To cope with this, the idea is to control the number of entries kept in the index. Generally, there is a counter to record the number of entries in a bucket. When the counter exceeds a threshold, rehash operations will be executed. Rehash operation is to create a new larger bucket and redistribute all entries into it.

Figure 3 illustrates the state of the standard hash structure before and after rehash operations. In this example, after the key whose value is 20 is inserted, the counter is up to six, which is the threshold as shown in Figure 3 (a). Hence, rehash operations will be conducted. Specifically, a larger bucket which can hold fifteen entries is created. And for each entry, its new position will be recalculated. For instance, the position of entry 4 is 2 before the rehash operation and the newly position becomes to 10 after the recalculation. Hence, the rehash operation incurs high overhead both in timing performance and NVM writes and we will discuss in the next section.

Many work on hash-based techniques has been proposed. In [30], linear hashing with partial expansion is presented, which has high query performance and storage utilisation. Yang et al. [31] present a SSD-optimized lin-

Table 1: Comparisons on two index structures in terms of query operations (seconds).

#operations	B^+-Tree	standard hash
10M	9.67s	3.58s
20M	19.18s	7.07s
30M	38.65s	14.46s

ear hashing index to reduce small random writes to SSDs which are caused by index operations. Recently, many research efforts have been put into investigate how to make hashing schemes suitable for NVM-based systems. For instance, Pagh et al. [29] propose cuckoo hashing, a practical, advanced hashing scheme with high memory efficiency and $\mathcal{O}(1)$ expected time for both insertion and retrieval operations. Fan et al.[32] propose optimistic cuckoo hashing, a concurrent and cache-friendly hashing scheme with high memory efficiency. Breslow et al.[33] present the Horton table, which can reduce the expected cost of lookups. Zuo et al.[34] propose a new write-friendly hashing scheme to solve hash collisions for NVM systems. Xia et al.[35] propose a new hybrid index combining hash structure and B^+-Tree for DRAM-NVM memory system. However, all the hash-based structures mentioned above suffer from the high overhead of range query operations. In this paper, our objective is to design a hash-based structure which can improve the performance of range query operations.

2.3. Motivation

In this subsection, we compare the timing performance and the number of NVM writes of two above indexes by conducting different number of query and insert operations, respectively. Table 1 and 2 illustrate the experimental results that standard hash is roughly three times faster than B^+-Tree, meanwhile reducing 90% of NVM writes compared with B^+-Tree. In this paper, we aim to design an indexing scheme which is NVM-friendly and high-performance. Hence, we are focused on hash-based structures. However, we find that there are two basic operations limited the overall performance of hash-based structures.

Standard hash structure leads to poor performance and a large number of NVM writes when performing rehash operations. *As for performance*, since the bucket size of standard hash is usually very large to hold all entries in one bucket, there will be a long latency when we perform rehash operations. For

Table 2: Comparisons on two index structures in terms of insert operations (Bytes).

#operations	B^+-Tree	standard hash
10M	2.20×10^9	1.51×10^8
20M	4.39×10^9	2.02×10^8
30M	8.77×10^9	4.06×10^8

instance, the latency is up to 10.9 seconds when the node size is 1GB. The long latency may be unacceptable for real-time applications. To reduce the latency, we want to use multiple small buckets other than one large bucket.

As for write activities on NVM, all entries moving from the original bucket to a new bucket will incur large number of NVM writes, which considerably reduces the endurance of NVM. For instance, if there are 1G entries in a bucket and the size of each entry is 8 Bytes, then a rehash operation will cause 8GB NVM writes. To prolong the lifetime of NVM, the design of hash-based indexes should reduce the movements of entries in rehash operations.

Besides rehash operation, range query operation is another bottleneck of standard hash structure. This is because all entries are scattered in one bucket, and the position of each entry is calculated through hash functions. Thus, we cannot take any advantages of efficient algorithms but traverse all entries to check whether the key falls in the given range.

In this paper, our objective is to design a novel indexing scheme based on hash-structure, which has less overhead (i.e., the latency and number of NVM writes) of rehash operations than existing hash-based structures, and meanwhile, reduces the number of comparisons in conducting range query to improve performance.

3. Design Framework

3.1. Design principles

Based on above observations, there are three principles in the design of our Bucket Hash indexing scheme.

• **Reduction in NVM writes of rehash operations.** In order to reduce the number of NVM writes caused by rehash operations, we make the movements of entries as few as possible. Instead of rehashing all entries, we move half of the entries on average to the new bucket and keep the rest entries in the original bucket. In this way, we can reduce about half of movements compared with standard hash structure. The number of NVM writes can also be dramatically reduced.

• **Reduction in latency of rehash operations.** To reduce the latency of rehash operations, we want that the operations work on small buckets instead of large ones. Small buckets hold fewer entries, which can reduce the number of entries need to be rehashed. In this way, rehash operations can be quickly finished and the latency can be degraded.

• **Make range query operations manageable.** To make range query operations manageable, we organize small buckets which are created in rehash operations as a sorted linked list. All keys of entries in a bucket are larger than those in the previous buckets. In this way, we can significantly reduce the number of entries need to be checked in range query operations. Thus, the performance can be dramatically improved.

3.2. Basic structure

Based on the design principles, the basic structure of proposed Bucket Hash is a linked list. Specifically, all entries are stored in a linked list of buckets. Bucket Hash has the following three properties: (1) only half of entries on average will be moved in each rehash operation; (2) each rehash operation works on a small bucket (instead of a large one); and (3) entries in a small number of buckets (instead of all the entries) need to be compared when conducting range query operations.

The basic structure of Bucket Hash is shown in Figure 4, in which bucket is the basic element. Each bucket has 5 fields, including (1) a "counter" to record how many entries in the bucket, (2) a "llcp" to record the longest length of common prefixes of keys stored in the bucket and this field is related to split operations, (3,4) two pointers "prev" and "next" that point to the previous bucket and next bucket respectively, (5) a set of pointers that point to entries. In (5), the position of each pointer is calculated through hash functions. When hash collisions occur, chained hashing is employed to address collisions. Specifically, we use a linked list, called 'collision list", to organize the entries which have hash collisions.

There are two advantages of the proposed structure: (1) for rehash operations, we only need to move half of entries on average in a bucket to a newly created one, which can not only reduce the NVM writes but also accelerate rehash operations; (2) for range query operations (e.g., find keys ranged from *min* to *max*), we only need to search entries between the first bucket covering the *min* key and the last bucket covering the *max* key. We can ensure that

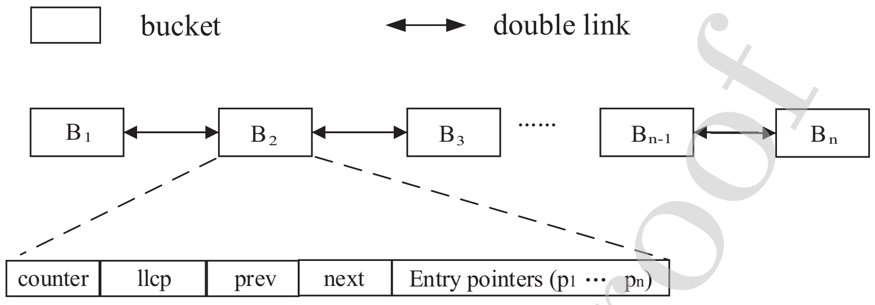

Figure 4: Fundamental structure

all entries in the given range fall in the buckets between these two buckets in the linked list.

However, linked list may degrade performance when locating buckets. More specifically, in each query, it has to sequentially search buckets from the head. In the next subsection, we will present an auxiliary structure to accelerate query process.

3.3. Improvement in performance

To make query operations more efficient on a linked list, we present an auxiliary structure called "multi-level lookup tables" to accelerate the process of locating of buckets.

Figure 5 (b) illustrates the auxiliary structure. It is a hierarchical structure which consists of multiple lookup tables. Each table has $2^8 = 256$ pointers, and each of them points to a table or a bucket in the next level.

The auxiliary structure is built according to the keys. For each key, it is partitioned into 8 segments (8-bit integers), as shown in Figure 5 (a). We use the notation $k^{[i]}$ to represent the i^{th} segment of a certain key k. The integer of the i^{th} segment will be used as the offset in the i^{th} level of the auxiliary structure for getting the pointer that points to the next level. If the pointer points to another lookup table, we repeat the process until we find a bucket. For instance, we can get the keys whose first 8-bit is 0x00 by accessing the first pointer in L_1 lookup table.

In the multi-level lookup table, for pointers in l^{th} level, some of them may point to other lookup tables and some may point to a bucket. Because the keys with the same prefix (the longest common bits) will be stored in the same bucket and if the number of keys with the same prefix exceeds the bucket size, we need to split the bucket into two buckets according to the

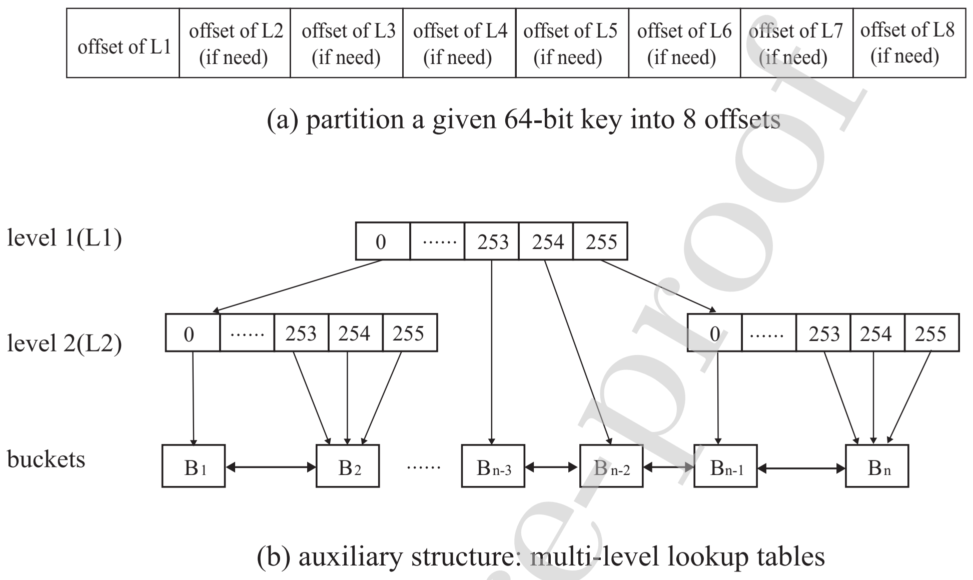

Figure 5: Partition of a given key and the structure of multi-level lookup table

next bit of the common prefix. At this point, the length of new common prefix is one more than that of previous one. If the length of new common prefix uses a new segment, we need to create a new lookup table in $(l+1)^{th}$ level. Otherwise, there is the bucket in $(l+1)^{th}$ level. In this way, we can use the i^{th} segment as the offset in the i^{th} level for searching.

As shown in Figure 5 (b), the first 8 bits of B_1 and B_2 are both 0x00. Hence, the pointer in offset 0 in L1 lookup table points to another table in L2. In this table, the pointers whose offset is within 0 to 127 point to B_1, this is because all the keys whose second 8-bit is 0x00 to 0x7f can be stored in the bucket. Similarly, since the keys whose second 8-bit is within 0x80 to 0xff can be stored in B_2, the corresponding pointers point to B_2.

Note that the auxiliary structure is different from the radix tree[7] in two aspects. First, the auxiliary structure of Bucket Hash can be stored on DRAM, while the entire structure of radix tree must be stored on NVM. Because entries of Bucket Hash are maintained in a linked list on NVM and the auxiliary structure can be reconstructed according to the linked list. On the contrary, the entire structure of radix tree must be stored on NVM to guarantee the consistency. Hence, radix tree will incur larger number of NVM writes than Bucket Hash. Second, the auxiliary structure of Bucket Hash requires much less memory than the radix tree. Because the height of radix tree is 8 levels when the size of key is 64 bits, with a chunk of 8 bits in length, it creates a large number of nodes. While in Bucket Hash the height of each bucket does not have to be the same and is always shorter than that of radix tree (usually 2 or 3 levels). Therefore, Bucket Hash has higher spatial utilization than radix tree.

3.4. Theoretic Analysis

In this subsection, we provide theoretic analyses to show that the height of our multi-level lookup tables is usually small. Before the analysis, we define some useful notations. We use L to represent the number of levels. Let M be the size of total number of entries and B be the bucket size. The maximum loading factor of buckets is f, which represents the ratio of the maximum number of entries set in buckets to the capacity of buckets. For instance, if buckets can hold 1M entries at most, there are at most 0.7M entries are stored in the buckets, we say f is 0.7.

The height of multi-level lookup tables depends on the distribution of data. We consider that $k\%$ of entries are gathered in an interval with length $j\% \times 2^{64}$ in our analyses.

Based on the conditions, we know that there are $1-k\%$ of entries gathered in an interval with length $(1-j\%) \times 2^{64}$. In order to calculate the height of the auxiliary structure, we should first calculate the heights, denoted by L_1 and L_2, caused by $k\%$ and $1-k\%$ of entries respectively. Then we get the maximum of L_1 and L_2 as the value of L.

Since the number of buckets which can hold the $k\%$ and $1-k\%$ of entries is $\frac{M\times k\%}{B*f}$ and $\frac{M\times(1-k\%)}{B*f}$, for $k\%$ of entries, there are $j\% * 256$ cells used to get pointers in each level and (1-j%)*256 cells are corresponding to the rest of entries. Hence, we can get L_1 and L_2 as follows:

$$\frac{\log\frac{M\times k\%}{B\times f}}{\log(256\times j\%)} \leq L_1 < \frac{\log\frac{M\times k\%}{B*f}}{\log(256\times j\%)} + 1$$

and

$$\frac{\log\frac{M\times(1-k\%)}{B\times f}}{\log(256\times(1-j\%))} \leq L_2 < \frac{\log\frac{M\times(1-k\%)}{B*f}}{\log(256\times(1-j\%))} + 1$$

Then, $L = max(L_1, L_2)$.

Note that if $k = j = 100\%$, it means that data are uniformly distributed on the interval $[0,2^{64})$.

Table 3: The height of auxiliary structure with different parameters.(Bucket Size is 1MB and load factor is 0.7)

M	k%	j%	L
40GB	100%	100%	2
40GB	80%	20%	3
5TB	100%	100%	3
5TB	80%	20%	4

Based on the above analysis, we calculate the height of multi-level lookup table using different parameters. Results in Table 3 shows that the height of the auxiliary structure is 3 or 4 even when we insert 5TB entries. Therefore, query operations can be efficiently performed.

4. Operations in Bucket Hash

In Bucket Hash, we implement five operations: query, insert, delete, split and range query. When a bucket is full, we need to move part entries to a new bucket and insert the new bucket to linked list, called "split operation". In this subsection, we will introduce the implementation of these operations.

4.1. Query operation

The query operation is the base of all other operations. As mentioned before, there are two parts in Bucket Hash: buckets and auxiliary structure. When query operations are conducted, we first use auxiliary structure to locate the bucket where the entry is stored. Then we use hash functions to find the position of the key in the bucket.

Algorithm 1 shows the major steps of query operations. In line 2 to 4, we utilize auxiliary structure to locate the specific bucket for a given key k. First, we get the *flag* stored in the cell with offset $k^{[1]}$ in the root (the internal node with level 1). If the *flag* is equal to 0, it means the pointer in the cell points to another lookup table. We repeat the process using $k^{[i]}$ for the lookup table with level i until the *flag* is 1, which means the pointer points to a bucket. After locating the bucket, we use hash functions to calculate the position of k and check whether the key exists.

Figure 6 illustrates the detailed process of query operations. First, it utilizes the first 8 bits as the offset in the lookup table with level 1 (steps

Algorithm 1 Implementation of query operations

```
1: let $i \leftarrow 1$
2: repeat
3:     get the flag in the cell with offset $k^{[i]}$ in the internal node with level $i$.
4:     i ← i+1
5: until flag is equal to 1
6: calculate the position where the given key is stored through hash function
7: search the collision list in the position to return whether the given key is
   exist.
```

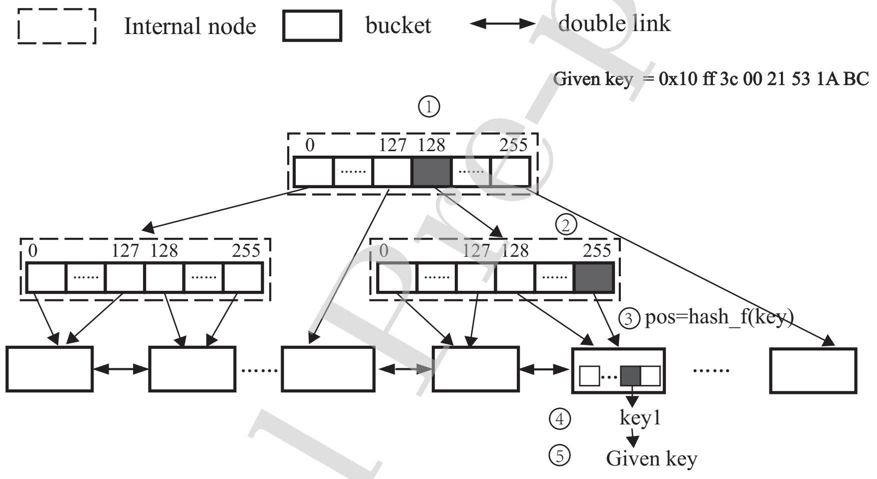

Figure 6: Illustration of query operations to find a given entry

①). Since the pointer at the offset points to another lookup table, it uses the second 8 bits as the offset in the lookup table with level 2 (steps ②). After these two steps, we locate the bucket where the given key is stored. Then, in step ③, we use hash functions to calculate the position of the given key. Next, we sequentially search the collision list at the position until we find the given key (steps ④ and ⑤). Note that since the collision keys are stored in order, we can stop searching if the current key is larger than the given key.

4.2. Insert operation

When a new entry is inserted into our index, it follows three steps: *i)* locate the target bucket in which the new entry will be inserted. *ii)* calculate the position where the new entry should be stored. *iii)* write the new entry

Algorithm 2 Implementation of insert operations

```
1: Locate the bucket $B$ through auxiliary structure to hold the new entry
2: Calculate the position $p$ in B through hash function
3: write the new entry in $p$ and increase the counter of $B$
```

to the position and increase the counter. The pseudo-code of insertion is shown in Algorithm 2.

Note that there are two explanations for the above implementation. *First*, we can always find a bucket to hold new entries through auxiliary structure. *Second*, if the bucket is full, we need to split the bucket. Detailed implementation will be given later in this section.

4.3. Delete operation

In delete operations, it follows three steps: *i)* locate the target bucket where the given entry is stored. *ii)* calculate the position where the entry is stored in the target bucket. *iii)* sequentially find the given entry in the position. *iv)* modify two pointers of entries adjacent to the entry. Algorithm 3 shows the major steps of delete operations.

Algorithm 3 Implementation of delete operations

```
1: Locate the bucket $B$ through auxiliary structure;
2: Calculate the position $p$ in $B$ through hash function;
3: Find the given entry $E$ in position $p$;
4: Modify the pointers of entries adjacent to $E$;
```

4.4. Split operation

The split operation is conducted in two steps: (1) moving entries from the splitting bucket to a newly created bucket; (2) inserting the new bucket to double linked list and modify the information of the auxiliary structure. Algorithm 4 shows the major steps of the split operation.

In the first step(line 1-9), we create a new bucket and move a part of entries from the splitting bucket to the new bucket. Specifically, we only move entries whose bit at the *llcp* (longest length of common prefixes) position is t (0 or 1). In order to reduce NVM writes, we select t according to the number of entries needing to be moved. However, since the traverse of all entries is time consuming, we employ the sampling technique to accelerate the selection process(line 2). After moving entries to the new bucket, we update split information both in original and new buckets(line 9).

Algorithm 4 Implementation of split operations

```
 1: Create a new bucket, $N$;
 2: Using sampling technique to select a number from 0 and 1, denoted by
    $t$;
 3: Get the split information $llcp$ in the original bucket;
 4: for each key of entries in the original bucket do
 5:    if the bit in $llcp$ position is equal to $t$ then
 6:       Move the entry to the new bucket
 7:    end if
 8: end for
 9: Set the split information in original bucket and new bucket to be $llcp$
    $+1$;
10: if llcp$\neq$1 and (llcp-1)%8 == 0 then
11:    create a new lookup table, L
12:    set the pointers of L to point the original bucket or new bucket accord-
       ing the bit at llcp bit
13:    make the pointer which points to the original bucket point to L
14: else
15:    for each pointer which points to the original bucket do
16:       Get the offset of the pointer
17:       if the bit of the offset in the $llcp$ position is $t$  then
18:          Make the pointer in the cell point to the new bucket
19:       end if
20:    end for
21: end if
22: if the number of entries of the new bucket is 0 then
23:    repeat the above process
24: end if
```

In the second step(line 10-22), we first check whether we need to create a new lookup table(line 10). If $llcp$ is not 1 and $(llcp - 1) \bmod 8$ is 0, we will create a new loolup table and insert it to the auxiliary structure(line 13). Otherwise, we update pointers which point to original bucket to the new bucket only if the bit at $llcp$ position of the pointer offset is t. Hence, only half of pointers which point to the original bucket will be affected.

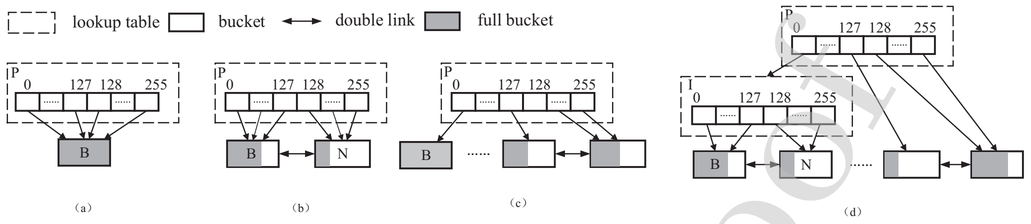

Figure 7: Examples of split operation

Note that, if after these steps, the bucket is still full (i.e., we cannot move any entry to the new bucket), we will increase the *llcp* and repeat these steps.

Split example

Figure 7 illustrates an example of the split operation. In Figure 7 (a), there is only one bucket and all pointers in the lookup table point to the bucket in initial state. In this case, all entries are stored in the bucket.

When a bucket is full, we need to split the bucket. As shown in Figure 7 (b), the new bucket N is created when bucket B is full and move part of entries to it according to the first bit of keys. In this way, we can guarantee that the first bit of keys in a bucket is the same and we use the notion of longest length of common prefixes (*llcp*) to record this information. Assume that we store the keys whose first bit is 0 in B and the rest are kept in N. For the keys whose $key^{[1]}$ is within 128 to 255, we locate the new bucket N through the lookup table. Hence, we need to make the pointers in the cells with offset within 128 to 255 point to N.

With the splitting of buckets, the *llcp* of buckets may be up to 8, indicating that only one pointer points to the bucket, such as bucket B. As shown in Figure 7 (c), since the bucket B is pointed by the cell pointer with offset 0, the first 8 bits of keys in B are 0x00. In this case, when B is full, we conduct the split operation according to the ninth bit. Note that, a lookup table only has 256 pointers corresponding to 8 bits. In order to locate a key using the ninth bit, we need to create a new lookup table I between B and its parent as shown in Figure 7 (d). Similarly, if we want to locate a key using the seventeenth bit (the *llcp* of the split bucket is 17), we repeat the process of creating a new internal node. Generally speaking, if the *llcp* in a bucket is divisible by 8 and the bucket needs to be split, we will create an internal node between the bucket and its parent.

For the auxiliary structure with multiple levels of lookup tables, figure 7 (d) demonstrates the locating process in multi-level lookup tables. When we

Algorithm 5 Implementation of range query operations

```
1: Locate the bucket $S$ by querying s;
2: Locate the bucket $E$ by querying e;
3: Find entries in $S$ and $E$ which are belongs to range (s,e) and copy them
   to result buffer;
4: Copy the entries in each bucket between $S$ and $E$;
5: Return the result buffer;
```

need to locate buckets, we first access the pointer at P with offset $k^{[1]}$. If the pointer points to the internal node I, we further access the pointer with offset $k^{[2]}$ in I to locate bucket B or B_1.

4.5. Range query operation

Buckets in Bucket Hash are organized as a sorted double linked list. The sorted structure can make it easy to conduct range query operations. For a range (s,e), we only need to locate the two buckets (S and E) respectively according to the range boundary. All entries in the buckets between the S and E fall in the given range. Hence we do not need to check these buckets. In this manner, we can guarantee that at most two buckets (S and E) need to be checked in each range query operation. Therefore, the performance of range query operations can be significantly improved.

Given a range(s,e), range query operations can be conducted in 3 steps. First, we locate the two buckets namely S and E by querying s and e respectively. Second, we check entries whether belong to the range in the two buckets and copy those that fall in the range to result buffer. Third, we directly copy entries to result buffer in each bucket between S and E.

Algorithm 5 shows the pseudo-code for the range query operation.

4.6. Discussion on advantages of bucket hash

Bucket Hash can achieve smaller number of NVM writes than other indexes due to two reasons. First, Bucket Hash employs the simplest inked list together with an auxiliary structure Lookup Table, which can dramatically reduce the NVM writes in split operations. Second, Bucket Hash only moves part entries when conducting rehash operations compared with standard hash, which reduces the heavy overhead incurred by moving entries from one bucket to another.

Meanwhile, Bucket Hash has better timing performance than existing indexes. First, Bucket Hash utilize calculations to locate buckets compared with tree-based structures, which is faster than memory accesses technique. Second, Bucket Hash is much faster than standard hash in terms of range query operations. Because Bucket Hash maintains buckets as a sorted linked list and design an auxiliary structure to accelerate the process of locating buckets. Hence, Bucket hash has good overall performance.

5. Evaluation

In this section, we conduct our experiments using Yahoo! Cloud Serving Benchmark (YCSB)[36]. We utilize six workloads, including three workloads to simulate the life cycle of databases, and three pre-defined workloads in YCSB. All the workloads are uniform distributed and the key size is 8 Byte. Properties of these six workloads are given as follows:

- GrowPhase: 90 %/10 % ratio of insert/delete operations, simulating the build-up phase of databases.
- AdultPhase: the number of insert, delete, query and update operations is the same, simulating the active phase of databases.
- ElderPhase: 60 %/40 % ratio of delete/query operations, simulating the obsolete databases.
- Workload-A: predefined workloads in YCSB, containing 50 %/50 % query/update operations.
- Workload-B: predefined workloads in YCSB, containing 95 % /5 % query/update operations.
- Workload-D: predefined workloads in YCSB, containing 95 %/5 % query/insert operations.

In all of above workloads, the query operation also includes several range query operations.

With above workloads, we implement our Bucket Hash, two tree-based structure ("B^+-Tree"," Wb^+-Tree") and a hash-based structure ("Standard Hash") with C program. Wb^+-Tree [18] using unsorted nodes with bitmaps to decrease the number of writes to SCM and meanwhile employ slot arrays to maintain the order of elements in nodes, which is an efficient and NVM-friendly index structures. The node size of two tree-based indexes is set as 4K with the commonly used and the node size of hash-based indexes is set as 1M according to the actual application senior. Then we evaluate the performance of these four structures in terms of timing performance, NVM

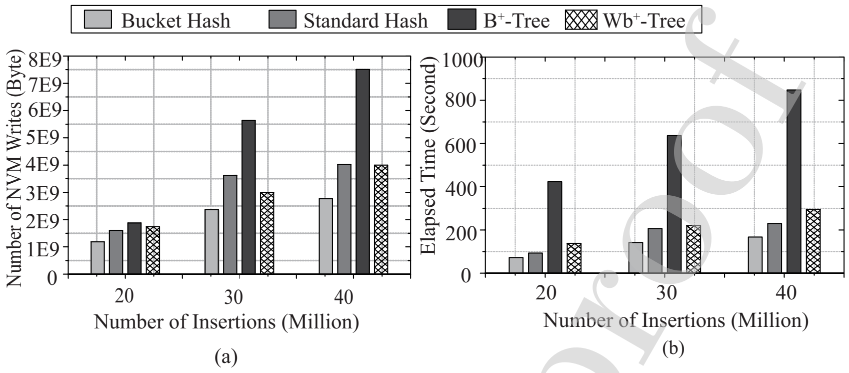

Figure 8: comparison of the insert performance on different structures

writes and memory consumption. We first describe the experimental setup and then discuss the performance with different types of NVMs.

We conduct all experiments on a Linux server (Kernel version 3.13.0-24) equipped with two Intel Xeon E5-2640 v4 processors. Each processor has ten cores running at 2.4 GHz and the size of the shared last level cache is 20MB. The memory size of the server is 256GB.

We use DRAM to emulate NVM similar to prior work [37]. In our experiments, we focus on emulating one type of NVM, STT-RAM . We set the read latency as 60 ns. Because the read latency of STT-RAM is similar with DRAM whose read latency is 60 ns [6]. The write latency is set as 110 ns, because the write latency of STT-RAM is about 50 ns slower than read latency [18]. We explicitly add extra write latency for each memory write and do not add extra rea latency for memory read. For other types of NVM, such as PCM, which have longer write latency, Bucket Hash can achieve better performance than STT-RAM. Because Bucket Hash has the least number of NVM writes than other index structures.

5.1. Insertion performance

We first compare the insertion performance of our Bucket Hash and its competitors. We use YCSB to generate a workload *InsertLoad*, which contains 50M insert operations. We initialize the index with 10M data in *InsertLoad* first , and then compare the insert performance with three different numbers of insertions: 20M, 30M, 40M.

5.1.1. Number of NVM writes

Figure 8 (a) reports the number of NVM writes of Bucket Hash and its competitors. The x-axis and y-axis represent the number of insertions and NVM writes, respectively. There are several observations in the results.

B$^+$-Tree incurs more NVM writes than two hash-based structures. One reason is that all nodes of B$^+$-Tree must be sorted in order. Thus, we need to frequently shift entries when performing insert or delete operations, which incurs lots of NVM writes. Another reason is B$^+$-Tree needs to maintain balance of structures, which can cause large quantities of NVM writes.

Standard Hash has less NVM writes than B$^+$-Tree. This is because standard hash has a simpler structure than B$^+$-Tree and it does not need to maintain balance of structures. Besides, entries in a bucket are unsorted in standard hash. However, rehash operations in standard hash account for a large proportion of NVM writes which makes Standard hash incur more NVM writes than Wb$^+$-Tree. In this experiment, the proportion of NVM writes caused by rehash operations are 50.2%, 66.8% and 60.1% when the number of insertions is 20M, 30M and 40M respectively.

Wb$^+$-Tree has less NVM writes than B$^+$-Tree and Standard Hash, since it does not need to maintain the order of entries in nodes compared with B$^+$-Tree and it does not execute rehash operations like Standard Hash.

Bucket Hash which is proposed in this paper, can reduce NVM writes by 53% ,30% and 25% on average compared with B$^+$-Tree, standard hash and Wb$^+$-Tree, respectively. The improvement is achieved due to two reasons. *First*, the auxiliary structure is stored in DRAM, which does not incur NVM writes. *Second*, it only moves part of entries instead of all entries when we perform rehash operations. Therefore, Bucket Hash can achieve significant reductions on NVM writes. For NVM-based products, reduction on NVM writes means the prolonged lifetime. Thus, Bucket Hash is practical for NVM-based systems.

5.1.2. Timing performance

Figure 8 (b) shows the elapsed time of four indexes when we perform different number of insertions. In this figure, the x-axis and y-axis represent the number of insertions and elapsed time, respectively. Results illustrate that Bucket Hash has better performance than its competitors.

Tree-based structures including **B$^+$-Tree** and **Wb$^+$-Tree** have longer elapsed time than two hash-based structures. This is because they use tree-based search to locate leaf nodes. Specifically, it uses binary search in each

Table 4: Maximum latency of rehash operations (seconds).

#operations	Standard Hash	Bucket Hash
20M	44.289	0.0027
30M	88.473	0.0036
40M	88.485	0.0039

level to locate a pointer points to the next level and must search down the path from root to a leaf node. With the increasing of data, there are more leaf nodes created in the index. In consequence, there are more binary search operations performed due to the increment of height. Besides, B+-Tree also uses binary search to find the position of entries in leaf nodes, which is much slower than hash-based methods.

Standard Hash has better performance than B$^+$-Tree and Wb$^+$-Tree. It does not need to locate buckets and uses calculation to find a position of entry, which can significantly improve the performance. However, compared with Bucket Hash, standard hash rehashes all entries, which seriously degrade the performance of rehash operations. The maximum latency of rehash operations depends on the bucket size. In our experiment, the bucket size is doubled for each rehash operations. Therefore, the latency for standard hash for 30M is 2 times larger than that of 20M due to the different bucket sizes. In the same way, the latency of 30M and 30M are similar. Table 4 shows that the latency of rehash operations on standard hash is up to 88 seconds, which is roughly 22000 times of Bucket Hash.

Bucket Hash has stable performance of rehash operations because the bucket size of Bucket Hash is fixed, . Moreover, since standard hash incurs more NVM writes than Bucket Hash, it needs more time to write entries into the index due to the asymmetrical performance of read and write. For overall performance, Bucket Hash achieves 80% , 42% and 28% reduction on elapsed time than B$^+$-Tree, Wb$^+$-Tree and standard hash, respectively.

5.2. Range query performance

In this subsection, we compare the performance of range query operations on four indexing schemes. We use the notion of "*valid entry*" to represent the entry falls in the given range and "*range query factor*" to represent the percentage of the number of *valid entries* in each range query operation. We use YCSB to generate three workloads with different *range query factor*, 1%,

Table 5: Elapsed time of range query operations (seconds).

factor	Wb^+-Tree	B^+-Tree	Stan. Hash	Bucket Hash
1%	0.092	0.117	78.392	0.492
3%	0.253	0.297	78.562	1.342
5%	0.462	0.512	78.354	2.324

3% and 5%. We first load 128M data in these workloads, and then perform 10 range query operations. Results in Table 5 show that, although Bucket Hash is not as good as B$^+$-Tree and Wb$^+$-Tree, it can achieve significant improvement over standard hash (Stan. Hash in Table 4).

The elapsed time of standard hash is similar for different workloads and it is much longer than B$^+$-Tree, Wb$^+$-Tree and Bucket Hash. This is because standard hash must traverse all entries to check whether they fall in the given range for each range query operation. Hence, timing performance of three workloads is almost the same. Furthermore, the large number of entries needs to be checked incurs poor performance of range query operations. In this experiment, it takes 78 seconds on average to get the results, which may be intolerable to users.

The buckets/nodes are organized as a sorted linked list both in Bucket Hash and tree-based structures. When we perform range query operations, the buckets/nodes (S, E) that hold the two range boundaries can be easily located. All entries between the two buckets/nodes fall in the given range. In other words, entries not in these buckets must be out of the range. Thus, Bucket Hash and tree-based structures can dramatically reduce the number of entries need to be checked. As shown in Table 5, Wb$^+$-Tree, B$^+$-Tree and Bucket Hash achieve 443, 268 and 68 times reduction on elapsed time on average compared with standard hash.

Since entries in B$^+$-Tree are compactly stored in leaf nodes, we only need to access the first k (number of entries) cells for each node. While entries in Bucket Hash are incompact due to calculation of hash functions, we need to access all cells in a bucket. Therefore, the elapsed time of Bucket Hash is 5.22 and 4 times over Wb$^+$-Tree and B$^+$-Tree.

5.3. Mixed workloads

In this subsection, we using six different YCSB workloads, including *GrowPhase, AdultPhase, ElderPhase, Workload-A, Workload-B, and Workload-*

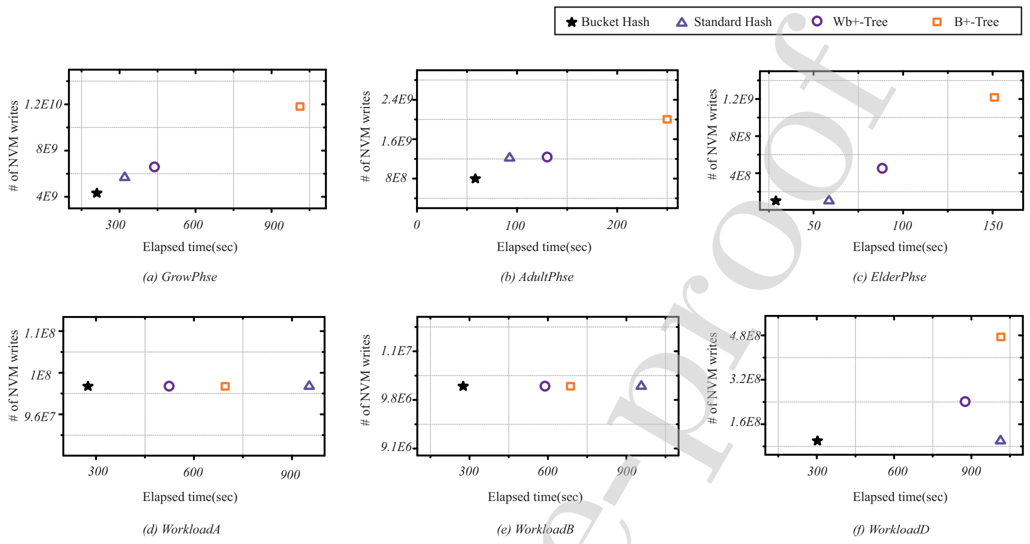

Figure 9: Comparisons on the number of NVM writes and elapsed time for different indexing schemes for mixed workloads: (a) GrowPhase; (b) AdultPhase; (c) ElderPhase; (d) Workload-A; (e) Workload-B; (f) Workload-D

D (see the beginning of this section) to evaluate four indexing schemes as shown in Figure 9. The x-axis represents the elapsed time of running stage, and the y-axis represents the number of NVM writes. It clearly demonstrates that Bucket Hash can achieve better results in terms of NVM writes and timing performance.

As for the insertion intensive workload (*GrowPhase*), Bucket Hash has much less NVM writes compared with standard hash and tree-based structures. Meanwhile, the performance of Bucket Hash is better than others. The results are in accordance with the analysis presented in Section 5.2.

As for the deletion intensive workload (*ElderPhase*), B$^+$-Tree incurs much more NVM writes than other structures and has the worst timing performance. Because B$^+$-Tree not only maintains structure balanced but also keeps entries in order in a node. Since bitmap and slot array are needed in nodes which makes entries stored out-of-order, Wb$^+$-Tree incurs less NVM writes than B$^+$-Tree. However, both of tree structures need to maintain the balance of tree, which incurs much more NVM writes than hash-based structures. In terms of two hash-based structures, thet have the same number of NVM writes. Because both Bucket Hash and standard hash need to modify two pointers to delete an entry. As for timing performance, Bucket Hash is better than standard hash, because the workload contains range query operations which can be efficiently performed in Bucket Hash.

As for $WorkloadA$ and *WorkloadB*,Bucket Hash achieves the best performance of its competitors as shown in Figure 9 (d) and (e). These two workloads contain both contain query and update operations, which makes all the index structures have the same number of NVM wirtes. Therefore, the performance depends on the query and range query operations. Wb$^+$-Tree has higher performance than B$^+$-Tree because its slot array can take full advantage of CPU cache. However, as for Standard Hash, it has worse timing performance than two tree-based structures. This is because performing range query operations incurs long latency on Standard Hash. As for Bucket Hash, it uses hash functions to locate buckets instead of tree-based search. Hence, Bucket Hash achieves the best overall performance than others.

As for $WorkloadD$, there are insert operations in this workload, thus tree-based structures incur large number of NVM writes due to maintaining tree structures than two hash-based structures. Bucket Hash has less NVM writes than standard hash since it rehashes half of entries on average in each rehash operation. In term of timing performance, due to range query operations mentioned before, standard hash has similar performance of tree structures and Bucket Hash achieves best performance among these structures.

As for workload *AdultPhase*, it contains 25 percent/ 25 percent/ 25 percent/ 25 percent ration of query/update/delete/insert operations. Results in Figure 9 (b) illustrate that B$^+$-Tree has the largest number of NVM writes and takes more elapsed time. while Bucket Hash has the best overall performance on this workload due to the least number of NVM writes and fast range query operations.

Table 6 summaries the experiments on mixed workloads. In the table, columns "WRT" represent the times of the number of NVM writes achieved by Bucket Hash. And columns "Speedup" represent the times of improvement on timing performance achieved by Bucket Hash. For instance, for workload $GrowPhase$ under column "v.s. B$^+$-Tree", the number under column "WRT" is 2.95X, which indicates that B$^+$-Tree is 2.95 times the number of NVM writes of Bucket Hash. The number under column "Speedup" is 4.14X, indicating that B$^+$-Tree is 4.14 times the elapsed time of Bucket Hash. Note that Bucket Hash has the same number of NVM writes on several workloads compared with standard hash. This is because the operations which incur NVM write in these workloads are delete or update operations. These operations cause the same steps both in Bucket Hash and standard

Table 6: Times of the number of NVM writes and elapsed time achieved by Bucket Hash for mixed workloads

Workloads	v.s. B$^+$-Tree		v.s. Wb$^+$-Tree		v.s. Standard hash	
	WRT	Speedup	WRT	Speedup	WRT	Speedup
GrowPhase	2.95X	4.14X	1.57X	2.05X	1.42X	1.31X
AdultPahse	2.52X	4.29X	1.67X	2.31X	1.53X	1.59X
ElderPhase	12.1X	5.22X	5.79X	3.41X	1.00X	2.02X
WorkloadA	1.00X	2.45X	1.00X	2.02X	1.00X	3.47X
WorkloadB	1.00X	2.48X	1.00X	2.18X	1.00X	3.48X
WorkloadD	4.74X	3.35X	1.92X	2.97X	1.00X	3.35X
Average	4.05X	3.655X	2.16X	2.49X	1.16X	2.54X

hash. Hence, there are several 1.00X under column "WRT" when we compare Bucket Hash with Standard hash based on these workloads. According to the results, we can conclude that Bucket Hash can achieve dramatically reduction on NVM writes and improvement on timing performance.

5.4. Memory usage

In this subsection, we compare memory consumption using three workloads: *InsertLoad*, *GrowPhase* which are mentioned before and *DeleteLoad* which contains 40M insert operations and 20M delete operations.

In an index, except for the essential memory used to store entries, there is some "extra memory" used to store structures. *First*, all the three indexes need to store buckets/leaf nodes which entry pointers are stored in. *Second*, for B$^+$-Tree and Bucket Hash, extra memory is also used to store internal nodes/lookup tables. Table 7 reports the memory consumption of four indexing schemes when we perform different test workloads. We set entry size to 16B in these experiments. Results clearly demonstrate that Bucket Hash and standard hash have similar memory requirements, which is much less than tree-based structures.

As for tree-based structures, Wb$^+$-Tree has more memory requirements than B$^+$-Tree. This is because Wb$^+$-Tree needs to maintain a bitmap and a slot array in each node which uses extra memory to store. Table 7 shows that Wb$^+$-Tree needs 70MB extra memory more than B$^+$-Tree.

Although both Bucket Hash and tree-based structures need auxiliary

Table 7: Comparisons on four indexes in the memory usage (MB).

workloads	Wb^+-Tree	B^+-Tree	Stan. Hash	Bucket Hash
InsertLoad	2356.39	2226.39	1122.35	1123.44
DeleteLoad	2650.01	2567.01	1397.77	1398.84
GrowPhase	1730.78	1690.78	1067.70	1068.76

structure to locate buckets/leaf nodes, the size of internal nodes in B$^+$-Tree is far larger than the lookup table in Bucket Hash. Specifically, there are only 256 cells in each lookup table in Bucket Hash and each cell only contains a 8-byte pointer and a 1-byte flag. As for B$^+$-Tree, there are lots of keys and pointers in each internal node, which need to hold more information than Bucket Hash. Hence, results in Table 7 show that the total memory usage of B$^+$-Tree and Wb$^+$-Tree is roughly 1.8 and 2 times than Bucket Hash.

Standard hash reduces the memory used to store lookup tables compared with Bucket Hash. However, there are fewer lookup tables (about 256) in Bucket Hash and each lookup table has a very small size as mentioned above. Results in Table 7 report that the memory usage of Bucket Hash is just 2M more than that in standard Hash when we insert scores of mega bytes.

5.5. Impact of auxiliary structure on timing performance

The auxiliary structure of Bucket Hash has two advantages. *First*, auxiliary structure can accelerate query process. Because we need to organize the buckets as a linked list in order to reduce the number of NVM writes. However, conducting query operations on a linked list incurs high overhead which reduces the performance of indexes. Hence, we design an auxiliary structure to accelerate query operations. *Second*, with the auxiliary structure and sorted linked list, we can efficiently perform range query operations on Bucket Hash. In general, we should first locate buckets through the auxiliary structure and then perform the operation. As analysed in Section 4.4, there are only several levels in the auxiliary structure, which incurs little latency on timing performance. In order to prove the analysis, we test the overhead of the auxiliary structure in this subsection.

We compare the overhead of multi-lookup table on three workloads: *InsertLoad*, *Range query* and *DeleteLoad*. Table 8 summaries the experimental results. In the table, column “AS time” represents the time spent on the auxiliary structure, column “Total time” represents the whole elapsed time

Table 8: The impact of auxiliary structure on timing performance (seconds).

workloads	AS time	Total time	ratio
Insert	23	252	0.09
Delete	18	243	0.07
Range query	17	208	0.08

to finish these workloads and column "ratio" represents the percentage of "AS time" in "Total time". Results in the table illustrate that the time spent on the auxiliary structure is only a fraction of the total time (less than 1%), which does not incur long overhead on timing performance.

6. Conclusion

Hash-based structures are widely adopted in real-world applications. This paper presents an NVM-friendly hash-based indexing scheme, called "Bucket Hash", for NVM-based in-memory databases. Specifically, to reduce the latency of rehash operations, we use multiple small buckets instead of larger ones. In order to minimize the number of NVM writes, Bucket Hash rehashes roughly half of entries to a new bucket during rehash operations and maintain these buckets as a sorted linked list. Since the linked list suffers from inefficiency in searching, this paper introduced an auxiliary structure to improve the performance of query operations. Furthermore, combine the auxiliary structure and the sorted linked list, our proposed indexing scheme can make range query operations manageable. Experimental results show that our proposed indexing scheme can significantly reduce the number of NVM writes, compared with B$^+$-Tree, Wb$^+$-Tree and standard hash. In the meantime, we can achieve better performance in all benchmarks.

Acknowledgements

This work is partially supported by NSFC 61472052, National 863 Program [GrantNo.2015AA015304], China Scholarship Council [No. 201706050116].

References

[1] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, et al.,

Phase change memory technology, Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena 28 (2010) 223–262.

[2] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, et al., Spin-transfer torque magnetic random access memory (stt-mram), ACM Journal on Emerging Technologies in Computing Systems (JETC) 9 (2013) 13.

[3] J. J. Yang, R. S. Williams, Memristive devices in computing system: Promises and challenges, ACM Journal on Emerging Technologies in Computing Systems (JETC) 9 (2013) 11.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Usenix Conference on Hot Topics in Cloud Computing, 2010, pp. 10–10.

[5] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, Ceph: a scalable, high-performance distributed file system, in: Symposium on Operating Systems Design and Implementation, 2006, pp. 307–320.

[6] H. Volos, A. J. Tack, M. M. Swift, Mnemosyne: lightweight persistent memory, in: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011, 2011, pp. 91–104. URL: `http://doi.acm.org/10.1145/1950365.1950379`. doi:10.1145/1950365.1950379.

[7] S. K. Lee, K. H. Lim, H. Song, B. Nam, S. H. Noh, 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, 2017, pp. 257–270.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, D. Coetzee, Better i/o through byte-addressable, persistent memory, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 133–146.

[9] P. Zhou, B. Zhao, J. Yang, Y. Zhang, A durable and energy efficient main memory using phase change memory technology, in: ACM

SIGARCH computer architecture news, volume 37, ACM, 2009, pp. 14–23.

[10] B. C. Lee, E. Ipek, O. Mutlu, D. Burger, Architecting phase change memory as a scalable dram alternative, in: ACM SIGARCH Computer Architecture News, volume 37, ACM, 2009, pp. 2–13.

[11] M. K. Qureshi, V. Srinivasan, J. A. Rivers, Scalable high performance main memory system using phase-change memory technology, ACM SIGARCH Computer Architecture News 37 (2009) 24–33.

[12] S. Chen, P. B. Gibbons, S. Nath, 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, 2011, pp. 21–31.

[13] J. Arulraj, A. Pavlo, S. R. Dulloor, Let's talk about storage & recovery methods for non-volatile memory database systems, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 707–722.

[14] J. Yue, Y. Zhu, Accelerating write by exploiting pcm asymmetries, in: High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, IEEE, 2013, pp. 282–293.

[15] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, B. Abali, Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2009, pp. 14–23.

[16] S. Cho, H. Lee, Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance, in: Ieee/acm International Symposium on Microarchitecture, 2010, pp. 347–357.

[17] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, B. He, Nv-tree: Reducing consistency cost for nvm-based single level systems., in: FAST, volume 15, 2015, pp. 167–181.

[18] S. Chen, Q. Jin, Persistent B + -trees in non-volatile main memory, VLDB Endowment, 2015.

[19] J. Rao, K. A. Ross, Making b+-trees cache conscious in main memory, in: ACM SIGMOD Record, volume 29, ACM, 2000, pp. 475–486.

[20] S. Chen, P. B. Gibbons, T. C. Mowry, Improving index performance through prefetching, volume 30, ACM, 2001.

[21] R. A. Hankins, J. M. Patel, Effect of node size on the performance of cache-conscious b+-trees, in: ACM SIGMETRICS Performance Evaluation Review, volume 31, ACM, 2003, pp. 283–294.

[22] H.-W. Fang, M.-Y. Yeh, P.-L. Suei, T.-W. Kuo, An adaptive endurance-aware-tree for flash memory storage systems, IEEE Transactions on Computers 63 (2014) 2661–2673.

[23] W.-H. Kim, B. Nam, D. Park, Y. Won, Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split., in: FAST, 2014, pp. 273–285.

[24] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al., Consistent and durable data structures for non-volatile byte-addressable memory., in: FAST, volume 11, 2011, pp. 61–75.

[25] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, S. Swanson, Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories, ACM Sigplan Notices 46 (2011) 105–118.

[26] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, W. Lehner, Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 371–386.

[27] J. L. Carter, M. N. Wegman, Universal classes of hash functions (extended abstract), in: ACM Symposium on Theory of Computing, 1977, pp. 106–112.

[28] B. Pittel, Linear probing: The probable largest search time grows logarithmically with the number of records, Journal of Algorithms 8 (1987) 236–249.

[29] R. Pagh, F. F. Rodler, Cuckoo hashing, Journal of Algorithms 51 (2004) 122–144.

[30] P.-. Larson, Linear hashing with partial expansions, in: International Conference on Very Large Data Bases, 1980, pp. 224–232.

[31] C. Yang, P. Jin, L. Yue, D. Zhang, Self-adaptive linear hashing for solid state drives, in: IEEE International Conference on Data Engineering, 2016, pp. 433–444.

[32] B. Fan, D. G. Andersen, M. Kaminsky, Memc3: compact and concurrent memcache with dumber caching and smarter hashing, in: Usenix Conference on Networked Systems Design and Implementation, 2013, pp. 371–384.

[33] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, D. M. Tullsen, Horton tables: Fast hash tables for in-memory data-intensive computing., in: USENIX Annual Technical Conference, 2016, pp. 281–294.

[34] P. Zuo, Y. Hua, A write-friendly hashing scheme for non-volatile memory systems, in: Proc. MSST, 2017.

[35] F. Xia, D. Jiang, J. Xiong, N. Sun, Hikv: A hybrid index key-value store for dram-nvm memory systems, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17), 2017, pp. 349–362.

[36] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with ycsb, in: ACM Symposium on Cloud Computing, 2010, pp. 143–154.

[37] J. Ou, J. Shu, Y. Lu, A high performance file system for non-volatile main memory, in: Eleventh European Conference on Computer Systems, 2016, p. 12.

Author Biography

Zhulin Ma, received the B.S. degrees from network engineering, Chongqing University, China in 2016. She is currently working towards the Ph.D. degree under the supervision of Dr. Edwin H.-M Sha in the Department of Computer Science at Chongqing University. Her research interests include database design, non-volatile memories and optimization algorithms.

Edwin H.-M. Sha received Ph.D. degree from the Department of Computer Science, Princeton University, USA in 1992. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at University of Notre Dame, USA. Since 2000, he has been a tenured full professor at the University of Texas at Dallas. He has published more than 389 research papers in refereed conferences and journals. His work has been cited over 3300 times. He received Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award, NSFC Overseas Distinguished Young Scholar Award, etc.

Qingfeng Zhuge received her Ph.D. from the Department of Computer Science at the University of Texas at Dallas in 2003. She obtained her B.S. and M.S. degrees in Electronics Engineering from Fudan University, Shanghai, China. She is currently a professor at East China Normal University, Shanghai, China. She received Best Ph.d. Dissertation Award in 2003. She has published more than 90 research articles in premier journals and conferences. Her research interests include parallel architectures, embedded systems, supply-chain management, real-time systems, optimization algorithms, compilers, and scheduling.

Runyu Zhang is currently working towards the Ph.D degree. His current research interests include non-volatile memory, parallel architectures and optimization algorithms.

Shouzhen Gu received the B.S. degree in computer science from Chongqing University, Chongqing, China, in 2010, where she in currently pursing the Ph.D. degree with the College of Computer Science. Her current research interests include multiprocessor parallel embedded systems and nonvolatile memory architecture optimization.

Author Photo

Zhulin Ma

Edwin H.-m. Sha

Qinfeng Zhuge

Runyu Zhang

Shouzhen Gu