Towards the Design of Efficient and Consistent Index Structure with Minimal Write Activities for Non-Volatile Memory

Edwin Hsing-Mean Sha, *Senior Member, IEEE*, Weiwen Jiang, *Student Member, IEEE*, Hailiang Dong, Zhulin Ma, Runyu Zhang, Xianzhang Chen, and Qingfeng Zhuge, *Member, IEEE*

Abstract—Index structures can significantly accelerate the data retrieval operations in data intensive systems, such as databases. Tree structures, such as B$^+$-tree alike, are commonly employed as index structures; however, we found that the tree structure may not be appropriate for Non-Volatile Memory (NVM) in terms of the requirements for high-performance and high-endurance. This paper studies what is the best index structure for NVM-based systems and how to design such index structures. The design of an NVM-friendly index structure faces a lot of challenges. *First*, in order to prolong the lifetime of NVM, the write activities on NVM should be minimized. To this end, the index structure should be as simple as possible. The index proposed in this paper is based on the simplest data structure, i.e., linked list. *Second*, the simple structure brings challenges to achieve high-performance data retrieval operations. To overcome this challenge, we design a novel technique by explicitly building up a contiguous virtual address space on the linked list, such that efficient search algorithms can be performed. *Third*, we need to carefully consider data consistency issues in NVM-based systems, because the order of memory writes may be changed and the data content in NVM may be inconsistent due to write-back effects of CPU cache. This paper devises a novel indexing scheme, called "**V**irtual **L**inear **A**ddressable **B**uckets" (VLAB). We implement VLAB in a storage engine and plug it into MySQL. Evaluations are conducted on an NVDIMM workstation using YCSB workloads and real-world traces. Results show that write activities of the state-of-the-art indexes are 6.98 times more than ours; meanwhile, VLAB achieves 2.53 times speedup.

Index Terms—Non-volatile memory, NVM-based in-memory systems, indexing scheme, high performance, data consistency

1 Introduction

Non-Volatile Memory (NVM), such as Phase Change Memory (PCM) [7], [29] and 3D Xpoint memory [1], has the potential to be directly placed on the memory bus. Hybrid main memory architectures that combine NVM and DRAM exhibit the promising advantages in the future computer systems [16], [26], [32], [38]. With such architectures, we can put "*NVM-based in-memory systems*", such as file systems or database systems, in NVM. By this way, we can achieve high performance and low power consumption over the traditional storages [10], [13], [20], [36], [37].

Indexes have significant effects on the performance for systems with intensive data accesses, such as databases. In the design of indexes, the speedup of query operations is one of the most important concerns, since almost all the data processing operations, such as insert, delete and update, involve query as their sub-operations. For NVM-based in-memory systems, two other concerns need to be addressed.

The first concern is to minimize the number of "write activities on NVM" (abbreviated as NVM writes) [9], [10], [24], since NVM writes not only incur long delay but also need to be controlled due to the limited endurance of NVM. The commonly used index structures for disk-based systems may not be appropriate for NVM-based in-memory systems. For instance, the tree structures, such as B$^+$-tree alike, are widely utilized in the traditional storage systems, but they will involve a lot of undesired writes on NVM-based systems. Specifically, NVM writes of those tree-based structures may come from three types of operations: *the reordering operation* when an element is inserted or deleted in a node; *the split operation* when a node is full; and *the operations for data consistency*, in which the log files are written on NVM. This paper devises a new index structure that can significantly reduce the NVM writes incurred by these operations.

The second concern is data consistency [4], [34], [35], [38]. To guarantee the data consistency of NVM-based in-memory systems, the cache should be taken into consideration.

- *E.H.-M. Sha, H. Dong, Z. Ma, R. Zhang, and X. Chen are with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: {edwinsha, leondong1993, mazhulin600, zry.cqu, xzchen109}@gmail.com.*
- *W. Jiang is with the College of Computer Science, Chongqing University, Chongqing 400044, China, and with the Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15261. E-mail: jiang.wwen@pitt.edu.*
- *Q. Zhuge is with the School of Computer Science and Software Engineering, East China Normal University, Shanghai Shi 200062, China. E-mail: qfzhuge@gmail.com.*

Manuscript received 30 Mar. 2017; revised 8 Sept. 2017; accepted 12 Sept. 2017. Date of publication 18 Sept. 2017; date of current version 16 Feb. 2018. (Corresponding author: Qingfeng Zhuge.)
Recommended for acceptance by R. F. DeMara.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2754381

Specifically, due to the cache line replacement and the write-back policy of CPU cache, the order of memory writes described in a program may be changed and the content in NVM may not be consistent. Therefore, data consistency techniques for disk-based systems, such as logging, cannot be directly employed in NVM-based in-memory systems. To make sure the memory writes in order and the content in NVM consistent, we need to explicitly control the flush of cache lines and set memory barriers.

The objective of this paper is to design an NVM-friendly indexing scheme that can minimize the number of NVM writes and guarantee data consistency, meanwhile achieving high overall performance. The design of such indexing scheme needs to address the following three challenging problems. (1) The index structures should be as simple as possible in order to reduce NVM writes caused by insert, delete and split operations. (2) The simplicity of the index structure may cause performance degradation. Hence, new techniques need to be developed to mitigate the possible performance degradation. (3) The operations related to data consistency may incur significant overhead in performance as well as NVM writes. Therefore, we need to minimize the number of those operations while maintaining the data consistency. This paper presents a novel design of indexing scheme, called "**V**irtual **L**inear **A**ddressable **B**uckets" (VLAB), which can achieve high performance and minimal write activities, while guaranteeing data consistency. The efforts and understandings presented in this paper might be useful for the design of any NVM-friendly data structures.

For *challenge 1*, through our studies, we employ the simplest structure (i.e., linked list) as the basic structure of VLAB. Specifically, the index is composed of a list of sorted buckets. Each bucket stores elements, and all elements in one bucket are larger than that in the previous bucket. Since the insert, delete, and split operations on a linked list are much easier than that on a tree, we can significantly reduce the number of NVM writes. However, in order to locate an element in a linked list, we have to sequentially search each bucket, which may drastically degrade system performance. Hence, we need new techniques to accelerate the search process in the linked list.

For *challenge 2*, we devise a new technique, called Dual-VA, that allows the list of buckets having a second contiguous Virtual Address (VA) space in addition to the default non-linear one. By this way, we can make the list of buckets "virtually linear addressable". Based on Dual-VA, to access the bucket with a given offset, we can find it through simple calculations instead of searching from the beginning. Therefore, given a key, we can perform binary search in the linked list structures. Additionally, we present a more efficient search algorithm, called "guided binary search". The presented algorithm can efficiently shrink the search range. In consequence, the number of comparisons can be significantly reduced.

For *challenge 3*, first, we utilize the "atomic write" provided by NVM chips to guarantee data consistency. Therefore, we can conduct in-place updates instead of writing extra data in NVM, such as log files. Second, we carefully design the data management operations (e.g., insertion, deletion, etc.) to minimize the number of cache-line flushes and memory barrier settings. As a result, the overhead in system performance can be mitigated while guaranteeing data consistency. In the meantime, the number of NVM writes can be significantly reduced.

Our main contributions are listed as follows.

- We conduct an in-depth investigation to understand what is the best index structure for NVM and how to design an NVM-friendly index structure.
- We develop a novel technique, called Dual-VA, to organize linked lists using contiguous virtual address spaces, on which we can conduct efficient searches.
- We propose an efficient and consistent indexing scheme (VLAB) based on Dual-VA list. VLAB can reduce a significant number of NVM writes while achieving high overall performance. In addition, the memory requirements of VLAB are much less than tree-based indexes.
- We implement VLAB in a storage engine and plug it into MySQL for evaluations. Comprehensive experiments are conducted using YCSB workloads [14] and real-world traces on an NVDIMM workstation.

Evaluation results show that VLAB outperforms the existing tree-based structures significantly in terms of timing performance and NVM writes. As for timing performance, VLAB achieves 2.53 times and 5.29 times speedup on average, compared with wB$^+$-tree [10] and NV-Tree [38], respectively. As for NVM writes, the numbers of write activities incurred by wB$^+$-tree and NV-Tree are 6.98 times and 47.53 times larger than VLAB. Results demonstrate the practicality and efficiency of our proposed index structure.

The rest of this paper is organized as follows. In Sections 2 and 3, we introduce the background and motivation of the index design for NVM-based in-memory systems. Then, we present the design framework in Section 4. The detailed implementation of our proposed indexing scheme is given in Section 5. In Section 6, we show the evaluation results. Section 7 concludes this paper.

2 Background

2.1 Non-Volatile Memory

Non-Volatile Memory has been widely discussed to complement or substitute DRAM in memory hierarchy [8], [12], [13], [15], [33], [34], [36]. When the NVM is incorporated into memory hierarchy, it is byte-addressable like DRAM. The distinct difference between NVM and DRAM is the non-volatility of NVM, i.e., the data written on NVM will be persistent even when the system powers off. This paper focuses on the hybrid main memory architectures that combine both NVM and DRAM. In such systems, the file systems and database systems can be put in NVM, called NVM-based in-memory systems.

In the design of NVM-based in-memory systems, we have to minimize the number of writes on NVM due to two reasons. (1) NVM has asymmetric read and write latencies; in particular, the NVM writes may be 4X-20X slower than reads [4], [10]. (2) NVM suffers from the limitation of write endurance; i.e., once the number of writes of a memory cell exceeds the endurance limit, the NVM chip is considered worn-out [4], [33]. For NVM-related products, the most serious problem is how to guarantee the lifetime of NVM.

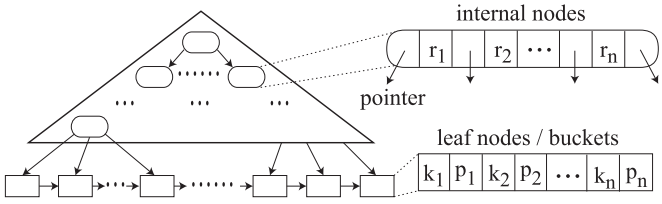


Fig. 1. An example of a B$^+$-tree.

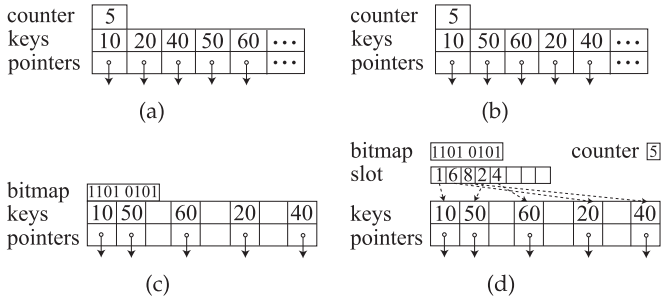


Fig. 2. Four node types: (a) Sorted node; (b) counter based unsorted node; (c) bitmap based unsorted node; (d) bitmap and slot based node.

In the existing research efforts, techniques in different levels are proposed to deal with the side effects caused by write activities on NVM. From the low (circuit, hardware or compiler) level, the early write termination technique [40] is proposed to avoid redundant bit-writes, and the wear leveling mechanisms [11], [27] are proposed to evenly distribute writes on NVM. From the high (application) level, researchers found that the traditional structures for disk-based systems are not suitable for NVM-based in-memory systems [9], [10], [34], [39]. In this paper, we focus on the design of an NVM-friendly structure from the application level to minimize NVM write activities.

2.2 Existing Index Structures

Tree structures, especially B$^+$-tree, are widely utilized as the basic structure of the existing indexes. In [17], [21], [25], [30], authors design flash-friendly B$^+$-tree to improve endurance of flash memory. In [18], [22], [28], authors present cache conscious tree structures for in-memory systems. Recently, more attention has been put into the design of indexes for NVM-based in-memory systems, such as CDDS B-tree [34], wB$^+$-tree [10], NV-tree [38] and different variants of B$^+$-tree [9]. We will discuss the traditional B$^+$-tree and its variants in the following of this section.

Fig. 1 illustrates the structure of B$^+$-tree, where each node stores pairs of keys and pointers, called entries. In the traditional B$^+$-tree, entries in a node are sorted by key in ascending order. Such nodes are called "sorted nodes", as shown in Fig. 2a. To maintain entries in order, it has to perform shift operations when an entry is inserted or deleted, which involves a large number of writes on NVM. Specifically, when an entry is inserted or deleted, half of entries in a node need to be shifted on average.

To reduce NVM writes caused by shift operations, authors in [9] replace the sorted nodes by unsorted nodes. There are two types of unsorted nodes: *counter based unsorted node* and *bitmap based unsorted node*, as shown in Figs. 2b and 2c, respectively. Compared with sorted nodes, when an entry is inserted in an unsorted node, we just need to write entries to an unused position and increase the counter by one or set one bit in bitmap. However, when locating an entry in unsorted nodes, we need to traverse the whole node, which degrades the query performance.

To reduce the number of NVM writes while achieving satisfactory performance, authors in [10] present a bitmap and slot array based node for B$^+$-tree. This kind of node is called *bmp+slot node*, as shown in Fig. 2d. It is composed of a bitmap, a counter and a slot array. The slot array indirectly records the sorted order of valid entries. For instance, the number stored in the kth slot indicates the position of the kth minimum keys. When locating elements in a *bmp+slot node*, binary search can be employed to improve query performance, but the heavy writes on slot array will also reduce the lifetime of NVM.

Note that this paper considers the indexes which support query operations as well as range query and sorting operations. The high-performance in conducting range query and sorting operations is commonly required in data intensive systems, such as relational databases. A special kind of index, hashing, cannot efficiently support these operations, which is not considered in this paper.

3 Motivation

3.1 Problems on Tree-Based Structures

Tree-based structures are vulnerable to be changed when performing insert or delete operations. The "structure change" indicates the reorganization of the internal nodes in a tree, as shown in Fig. 3. Considering B$^+$-tree, when we conduct 20 million insertions with randomly generated keys, there are more than 50,000 structure changes. These changes will introduce a large number of writes on NVM. In addition, it severely degrades index performance.

The structure changes are mainly caused by the split and merge operations. When a node is full, it has to be split into two nodes. Similarly, if the number of elements in a node is less than a threshold and the node cannot borrow an element from its sibling, then it has to be merged to its sibling.

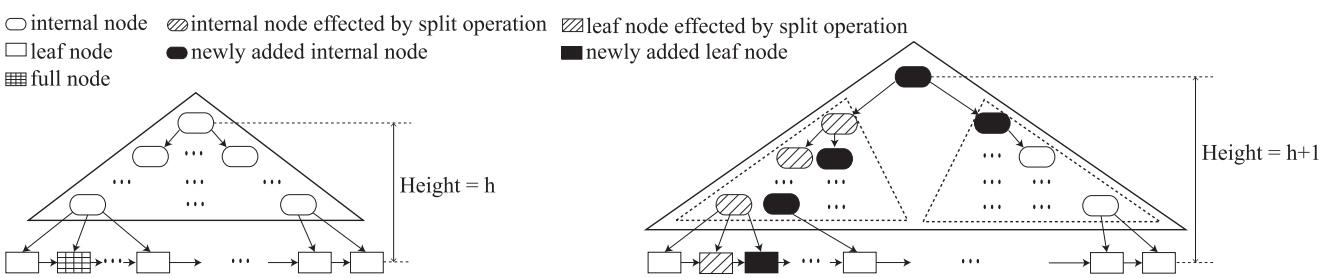


Fig. 3. The effects of the split operation in B$^+$-tree.

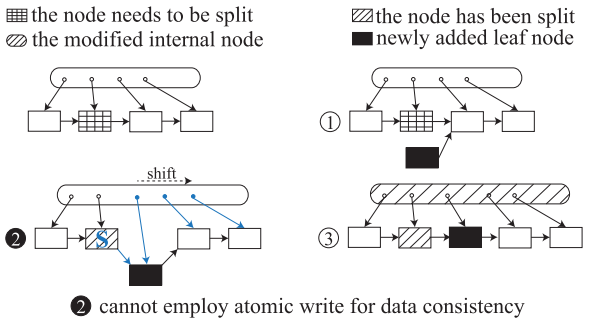


Fig. 4. Consistency issue on the split process in tree structures.

In the following, we are going to discuss the effects of one split operation on a leaf node.

As shown in Fig. 3, the effects caused by the split operation on a leaf node may be propagated to the root. This is because that tree-based indexes have to maintain the balance of the tree to achieve high performance in query. In this example, the split operation on a leaf node incurs NVM writes in $2h + 1$ nodes, where h is the height of the tree.

To reduce the NVM writes caused by the structure changes, we need a simple structure instead of the tree structure. The simplicity of the index structure, however, may cause the degradation in performance, in particular for query operations. Thus, we need new techniques to accelerate the query process in the simple index structure, such that the basic operations on the index can be performed faster than that on tree-based indexes.

3.2 Problems on Data Consistency

There are two relevant problems in data consistency. The *first* problem is how to choose proper consistency mechanisms for NVM-based in-memory systems. Different mechanisms cause different overhead on NVM writes and performance. The *second* problem is how to minimize the overhead caused by the extra instructions for data consistency. Specifically, to guarantee memory writes in order and the data content in NVM consistent, we need to manually insert instructions (i.e., `CLFLUSH` and `MFENCE`) in programs to control flushes of cache lines and set memory barriers. However, these instructions have high overhead.

3.2.1 Data Consistency Mechanisms

There are two traditional mechanisms for data consistency, i.e., logging and shadowing. The logging approach needs to record extra information (e.g., the addresses of the modified variables) in log files. While the shadowing approach needs to make a copy of the modified node. Both mechanisms may introduce a large number of extra writes on NVM to keep redundant data.

To avoid the redundant writes on NVM, the design of indexes should take full advantages of "atomic write" provided by NVM chips to conduct in-place updates. As stated in [12], [13], NVM chips can guarantee the completion of an 8-byte word write even if a power failure occurs, called "atomic write". Based on this property, data consistency can be guaranteed by conducting in-place updates. By this way, no extra NVM writes are involved.

Note that the atomic write cannot guarantee the completion of a write having more than 8 bytes. This forces the tree-based structures to use logging and shadowing approaches [10]. For example, in Fig. 4, states ①and ③are consistent states. In the transition from state ① to state ③, we need to modify three fields as shown in step ❷: the bitmap (or counter) in node "S", the sibling pointer in node "S", and the pointers in the parent node. The above example emphasizes the demands on the design of a novel simple structure.

TABLE 1
Seven Different Index Structures

ID	Type	Internal nodes	Leaf nodes	Consistency
A	B$^+$-tree	Fig. 2a	Fig. 2a	Shadowing
B	B$^+$-tree	Fig. 2a	Fig. 2a	Logging
C	B$^+$-tree [9]	Fig. 2a	Fig. 2b	Logging
D	B$^+$-tree [9]	Fig. 2a	Fig. 2c	Logging
E	wB$^+$-tree [10]	Fig. 2d	Fig. 2d	Atomic Write + Logging
F	B$^+$-tree [9]	Fig. 2c	Fig. 2c	Atomic Write + Logging
G	VLAB	None	Fig. 2c	Atomic Write

3.2.2 `CLFLUSH` and `MFENCE`

In order to guarantee the data consistency in NVM-based in-memory systems, there are two considerations need to be addressed. *First*, the content in NVM may not be the most current. That is to say, when a program executes a memory write operation, CPU directly writes data to cache rather than NVM. *Second*, the order of memory writes in a program may be changed due to the cache line replacement and the write-back policies of CPU cache.

For the *first* consideration, we need "flush" operation to flush out data from cache to NVM. For the *second* consideration, we need "fence" operation that issues a memory barrier to guarantee the memory operations after the barrier cannot be executed until those before the barrier have completed. Processors commonly provide these two kinds of instructions. For instance, x86 processors provide `CLFLUSH` for flush operation and `MFENCE` for fence operation [19]. Since `MFENCE` only guarantees the order of the execution of instructions, but cannot guarantee the order of write-back to the memory, we need to use the combination of `CLFLUSH` and `MFENCE` to guarantee data consistency.

The use of `CLFLUSH` and `MFENCE` can degrade system performance [10], [38]. The numbers of these instructions are proportioned to the number of NVM writes, which emphasizes the importance to reduce the writes on NVM.

3.3 Motivational Example

In this section, we compare 7 kinds of indexes in the number of NVM writes and query performance. The property of each index is shown in Table 1. Indexes A and B are traditional B$^+$-tree, using different data consistency mechanisms. Indexes C and D are proposed in [9], in which all internal nodes are *sorted nodes* while leaf nodes are *counter based unsorted nodes* and *bitmap based unsorted nodes*, respectively. Index E uses the *bmp+slot node*, which is proposed in [10]. To further reduce the number of NVM writes, index F is proposed in [9], in which both internal nodes and leaf nodes in are *bitmap based unsorted nodes*. And index G is proposed by this paper.

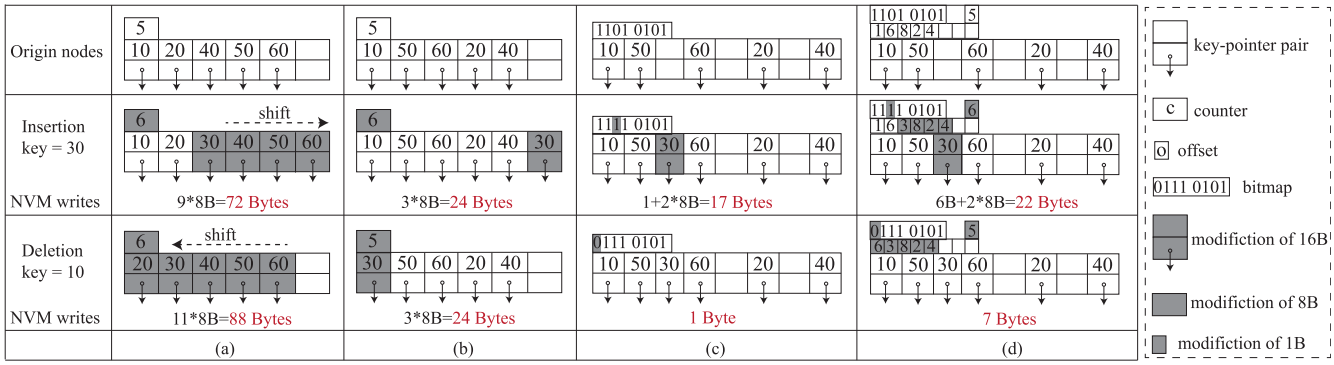


Fig. 5. NVM writes (bytes) caused by insertion and deletion on four types of leaf nodes: (a) Sorted node; (b) counter based unsorted node; (c) bitmap based unsorted node; (d) bitmap and slot based node.

NVM writes in the operations of indexes can be divided into three categories: (1) writes within leaf nodes; (2) writes caused by structure changes; (3) writes for consistency.

First, the leaf node needs to be modified when an entry is inserted or deleted. The sorted node has the maximum number of writes, since we need to perform shift operations to keep all entries in order. As shown in Fig. 5a, when we insert 30 to the node, it incurs shifts on three key-pointer pairs, one insertion of a key-pointer pair, and one modification of counter. The NVM is byte-addressable, and we utilize byte as the basic unit to calculate the NVM writes. The counter, key, and pointer are 8-byte types. Hence, the total number of writes is $(4 \times 2 + 1) \times 8 = 72$ bytes. On the contrary, the *bitmap based unsorted leaf node* has the minimum number of writes. For insertion, we only need to write an entry to the leaf node and modify a bit in bitmap. For deletion, only a bit in bitmap needs to be modified. As shown in Fig. 5c, one insertion and one deletion incur 17 bytes and 1 byte modifications, respectively.

Second, split or merge operations may involve a lot of NVM writes in the process of structure changes. As stated in Section 3.1, in tree-based structures, the effects of split operations at a leaf node may be propagated along internal nodes to the root node, which incurs a large number of NVM writes. As shown in Fig. 6a, roughly 50 percent of NVM writes are caused by structure changes for indexes E and F.

Third, the data consistency implemented by shadowing or logging may incur a large number of NVM writes. Shadowing mechanism needs a copy of the node and logging approach needs to write log file, both of which cause extra writes in NVM. As shown in Fig. 6a, for indexes A, B, C and D, the NVM writes are mainly caused by the maintenance of data consistency.

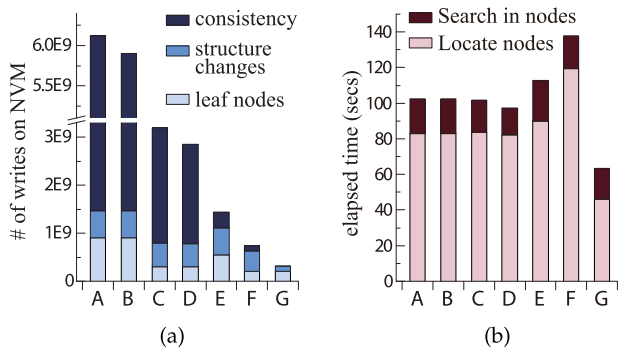


Fig. 6. Comparisons on the number of NVM writes and the elapsed time of query operations.

We record all categories of NVM writes for indexes by performing a workload with 30 million of insertions and 20 million of deletions. We report the results in Fig. 6a. Results demonstrate that if we replace the sorted leaf nodes (indexes A, B) by unsorted nodes (indexes C, D), 2 times of reductions on NVM writes can be achieved. If all nodes are implemented by unsorted nodes (index F), it can achieve 7 times of reductions on NVM writes, compared with indexes A and B. Our proposed index can further reduce the number of NVM writes by simplifying split operations and data consistency. Specifically, index G (ours) achieves 15 times of reductions on NVM writes over indexes A and B.

The query operation heavily affects the index overall performance, since it is the base of all other operations. The latency of query operations can be divided into two parts: (1) the latency of locating leaf nodes, and (2) the latency of searching the given keys in leaf nodes. We perform query operations on a workload with 80 million entries, and record the above two kinds of elapsed times.

From the results in Fig. 6b, it clearly shows that the latency of locating leaf nodes takes the major proportion of the overall elapsed time, compared with the latency of searching in leaf nodes. There are two reasons: (1) the number of entries stored in a leaf node is small; (2) the search in leaf nodes can take full advantage of caches, while the process of locating leaf nodes cannot. The above observations motive us to achieve better overall performance by reducing the elapsed time in locating leaf nodes. The basic idea is to make a contiguous virtual address space, based on which we can use calculations in locating leaf nodes, rather than access memories to retrieve the stored pointers. We will present the detailed techniques in Section 4.2.

In this paper, our objectives are listed as follows.

- *Design a novel and simple structure (not tree-based structure) that has smaller number of NVM writes than any other existing index schemes.*
- *Design a index scheme that uses atomic writes to guarantee data consistency and has the minimum overhead incurred by flush and fence operations.*
- *The performance of our indexing scheme should be better than the existing B^+-tree based index schemes.*

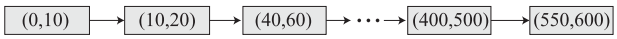


Fig. 7. The basic structure of VLAB: Sorted linked list.

4 Design Framework

4.1 Reduction in NVM Writes

4.1.1 The Design of Simple Structure to Reduce NVM Writes Caused by Structure Changes

The basic structure of the proposed index is the simplest data structure: *"sorted linked list"*. The element in the linked list, called "bucket", is similar with the leaf node in tree-based structures. Each bucket in the list contains entries whose keys fall within a specific range (min, max). And all keys in a bucket are larger than that in the previous bucket. Note that due to split operations, the size of ranges in different buckets may be different. In Fig. 7, we use a rectangle with notation (min, max) to represent a bucket.

Obviously, the simple linked list can reduce the number of NVM writes caused by structure changes. When performing the split operation on a bucket, it first moves half of entries to a new bucket, and then inserts the new bucket into the linked list. Therefore, the split operation only affects 2 buckets in the linked list. Compared with the effect of $2h + 1$ nodes (see Section 3.1) in tree structures, our index can dramatically reduce overhead caused by structure changes.

4.1.2 The Design of Buckets in the Linked List

The NVM writes within the buckets are mainly caused by the basic operations: insertion and deletion. Before presenting our design, we first compare the NVM writes caused by one insert operation and one delete operation within different kinds of nodes without splitting. Fig. 5 illustrates detailed comparison. Figs. 5a and 5d reflect that *sorted node* and *bitmap and slot based node* incur a lot of NVM writes to keep entries in order. On the contrary, the *bitmap based unsorted node*, in Fig. 5c, can minimize the NVM writes for both insertion and deletion.

The bucket in our indexing scheme is based on the *bitmap based unsorted node* (i.e., Fig. 2c). Specifically, the structure of a bucket is shown in Fig. 8. In a bucket, there are 6 fields, including the minimum ("min") and maximum ("max") boundaries of keys, pointers ("vnxt", "nxt") to point to the next bucket, a bitmap and entries. Each bucket contains entries whose keys fall within the range (min, max). Buckets are linked as a sorted list by "nxt" fields, and the "vnxt" field is used for consistency. Note that to minimize the number of NVM writes, entries stored in a bucket are out-of-order. And we use the sequential search to locate an entry in a bucket. Since the maximum number of entries in a bucket is small, the searches in buckets will not degrade the overall performance, as shown in Fig. 6b.

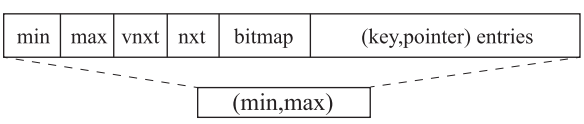


Fig. 8. The basic unit in VLAB: Bucket.

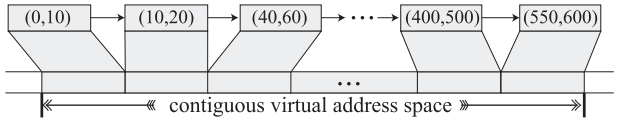


Fig. 9. The structure of Dual-VA list.

4.1.3 Challenges of the Proposed Indexing Scheme

Based on the above discussions, the proposed indexing scheme has the following properties: (1) the basic structure of the proposed index is a *"sorted linked list"*; (2) the structure of leaf nodes is based on the *"bitmap based unsorted node"*. Benefiting from these properties, the NVM writes caused by data management and data consistency operations can be minimized. We will present efficient data management operations based on atomic write in Section 5.

However, the simplicity of structure in the proposed indexing scheme brings a big challenge, i.e., how to make the performance of query operations on the linked list better than that on the B$^+$-tree? In B$^+$-tree, the time complexity of query operation is logarithmic, i.e., $O(lgn)$. As for linked list, however, it has to sequentially search nodes from the head, namely that its time complexity is linear, i.e., $O(n)$. Thus, we need to improve the structure to make the time complexity of the query operation to be logarithmic. Furthermore, we expect that the performance of our index can be better than that of B$^+$-tree, which is more challenging.

4.2 Improvement in Performance

To achieve better query performance than tree based search, the basic idea is to find buckets using calculation rather than pointers. We will propose an efficient search algorithm, called guided binary search, which is based on the binary search. The guided binary search can greatly reduce the number of comparisons in locating a bucket. As a result, it can dramatically speedup the query operations. However, the guided binary search cannot be directly conducted on a linked list. This paper presents new techniques to conduct guided binary search on our proposed index.

4.2.1 Use Contiguous Virtual Address Space

To introduce binary search on a sorted linked list, the basic idea is to build a contiguous address space for it. Note that since the operations on the index (or linked list) use virtual addresses, we only need to guarantee the virtual addresses of sorted buckets are contiguous, and there is no need to keep their physical addresses in order. Our method is to assign another virtual address for each bucket in the list. And the newly assigned virtual addresses form a contiguous virtual address space. The sorted linked list with a contiguous virtual address space is called *"Dual-VA list"*. Fig. 9 illustrates the structure of Dual-VA list.

Once we insert a bucket into Dual-VA list, we need to rebuild the contiguous virtual address space. However, the rebuilding process may cause degradations on performance. Thus, Dual-VA list should be "stable", so that the times of rebuilding can be reduced.

4.2.2 An Auxiliary Structure

In this paper, we introduce an auxiliary structure, called *"Multi-braids"*, to make Dual-VA list stable. The structure of

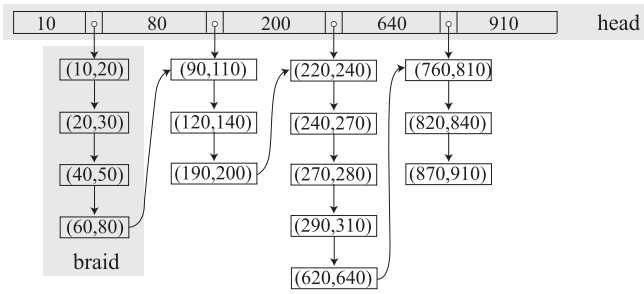


Fig. 10. The structure of Multi-braids.

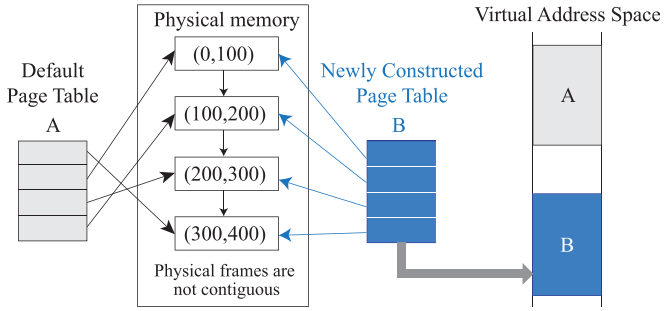


Fig. 11. Implementation of Dual-VA list: Two virtual addresses for non-linear frames in physical memory.

Multi-braids is shown in Fig. 10. It is composed of a head and multiple braids. The pointer in the head points to a list of sorted buckets (called braid). The Multi-braids structure is used for buffering newly added buckets. Specifically, when a bucket needs to be inserted in the index, we insert it to the Multi-braids rather than Dual-VA list. Only when the Multi-braids is full, we combine Dual-VA list and Multi-braids (see Section 5.2).

The detailed implementation of "Dual-VA list" and "Multi-braids" will be introduced in Section 5. Experimental results show that the elapsed time of query operations on the proposed index is much less than that on B^+-tree.

4.2.3 Guided Binary Search

The acceleration on the query process can significantly improve the index performance. Because the query operation is the base of insert, delete, and update operations. For instance, when we perform insert operations, the first step is to locate the bucket whose range covers the given key.

In this paper, we propose an efficient search algorithm, called guided binary search. Benefiting from the contiguous virtual address space provided by Dual-VA list, we make two design principles as follows. *First*, the location of buckets is based on *calculation* rather than retrieve the stored pointers. Specifically, by calculating the offset of a bucket, we can directly access it. By this way, the number of memory accesses can be dramatically reduced. *Second*, the calculation should be "smart". The traditional binary search can shrink the search range by half after each comparison. Based on the known information, we would like to determine a much narrower search range after the first comparison, such that the number of comparisons can be greatly reduced.

In the guided binary search, we first estimate the position of the bucket with high probability to contain the given key. Then, we determine the initial range based on the "min" and "max" fields of the estimated bucket. For instance, if position $B1$ is estimated, assuming that the given key is less than the "min" in $B1$, then the initial range is determined as $[B1 - \frac{K}{2^\alpha}, B1]$, otherwise, $[B1, B1 + \frac{K}{2^\alpha}]$. In the above calculation, K is the number of buckets in Dual-VA list and α is an adjustable coefficient. If the given key falls in the initial range, we can shrink the search range from K to $\frac{K}{2^\alpha}$. Namely, we only need to conduct α comparisons in the shirked range by using binary search.

By conducting experiments under YCSB workloads [14], we observed that it has more than 90 percent probability that queried key can fall in the initial range by setting α to 8. That is to say, we only need to conduct less than 10 comparisons to locate a bucket. On the contrary, the tree-based structures need more than 20 comparisons for each query operation.

5 Implementation

5.1 Implementation of Dual-VA List

Dual-VA list is the core component in VLAB. It facilitates us to conduct efficient binary search on a sorted list. The basic idea of Dual-VA list is to create a new segment of contiguous virtual address space for the sorted list, in addition to the default non-linear one. As shown in Fig. 11, we will allocate a new segment of virtual address space, i.e., segment B. Virtual addresses in this new segment will be translated to the physical frames (buckets) by our newly constructed page table. Note that the physical addresses of the list of buckets are not contiguous but their virtual addresses in the new segment are contiguous. We design a system call, `build_dualva_list`, to perform the above procedure. Note that the function of `build_dualva_list` is different from the existing Linux standard system call `mmap`. Specifically, `build_dualva_list` builds a mapping between a set of "existing" physical frames and a segment of virtual address space, while `mmap` builds a mapping between a set of "new" physical frames and a segment of virtual address space. Though we use `do_mmap_pgoff`, a subroutine of `mmap`, to set up a segment of virtual address space, we cannot call `mmap` directly to meet our requirement.

In `build_dualva_list` system call, the inputs include the head pointer of the linked list and the number of buckets. By invoking `build_dualva_list`, a linear mapping between the new segment of virtual address space and the list of buckets will be built. Finally, it returns the starting address of the new segment of virtual address space. The system call contains two steps.

For *step one*, we allocate a new segment of virtual address space and associate it with the corresponding page table. We first invoke kernel function `do_mmap_pgoff` to set up a new segment of virtual address space. Then, we invoke kernel functions, such as `pud_alloc`, to associate virtual addresses with page table entries. Note that the size of the new segment equals the total size of buckets, which can be computed by the number of buckets (the second input).

For *step two*, we build the linear mapping between the page table entries and the list of buckets. Specifically, we traverse the given linked list from its head one by one. For the kth bucket in the linked list, we insert its physical address to the kth entry in the newly constructed page table,

TABLE 2
Comparisons of Different Indexes in N_w, N_{clf}, and N_{mf}

ID: Type	Insertion without split	Insertion with split	Delete without merge
B: *tradB*$^+$	N_w = 24 M + 72 N_{clf} = 0.375 M + 3, N_{mf} = 2	N_w = L*(36N + 144) + 24M + 72 N_{clf} = L*(9N/16 + 6.675) + 0.375M + 3, N_{mf} = 2 + 2L	N_w = 24M + 24 N_{clf} = 0.375M + 2.25, N_{mf} = 2
C: *cntB*$^+$	N_w = 72 N_{clf} = 4, N_{mf} = 2	N_w = L*(24N + 144) + 72 N_{clf} = L*(0.375N + 7.5) + 4, N_{mf} = 2 + 2L	N_w = 72 N_{clf} = 4, N_{mf} = 2
D: *bmpB*$^+$	N_w = 65 N_{clf} = 4, N_{mf} = 2	N_w = L*(99N/4 + 113) + 65 N_{clf} = L*(99N/256 + 6.25) + 4, N_{mf} = 2 + 2L	N_w = 17 N_{clf} = 3, N_{mf} = 2
E: *wB*$^+$	N_w = M + 34 N_{clf} = M/64 + 97/32, N_{mf} = 3	N_w = L*(10N + 78) + M + 34 N_{clf} = L*(19N/128 + 233/128) + M/64 + 109/32, N_{mf} = 3 + 2L	N_w = M + 16 N_{clf} = M/64 + 2, N_{mf} = 3
G: Ours	N_w = 17, N_{clf} = 2, N_{mf} = 2	N_w = 88 + 129N/8, N_{clf} = 5 + N/4, N_{mf} = 5	N_w = 1, N_{clf} = 1, N_{mf} = 1

N: total number of entries in a node; M: number of valid entries in a node; L: number of split nodes (≥ 1).

which involves kernel functions `mk_pte` and `set_pte`. By traversing the whole linked list, the page table is filled up.

After the above two steps, we successfully build a new segment of contiguous virtual address space for the list of buckets. Since the given list of buckets is sorted in order, we can conduct the efficient binary search based on the newly created contiguous virtual address space.

5.2 Operations in VLAB

In VLAB, we implement five operations: query, insert, delete, split and combine. When a bucket is full, we need to split it. To make Dual-VA list stable, the newly created bucket is inserted into Multi-braids. When the Multi-braids is full, we need to combine the Multi-braids and Dual-VA list, called "combine operation".

In this section, we will introduce the implementation of the above five operations. In addition, we will analyze the number of NVM writes (N_w), `CLFLUSH` (N_{clf}) and `MFENCE` (N_{mf}) in insert, delete, and split operations. Table 2 gives the comparison results of different indexes on N_w, N_{clf}, and N_{mf}. The formulas of *tradB*$^+$, *cntB*$^+$, *bmpB*$^+$ and *wB*$^+$ can be obtained by analyzing each operation in detail. For instance, in *cntB*$^+$, each insertion will write an entry (16 bytes) and modify a counter (8 bytes). And it employs redo-log, which records the address together with the new value. Hence, the total writes on NVM (N_w) is $(16 + 8) \times 3 = 72$ bytes. For VLAB, we will explain the formulas later in this section. From the table, we can see that for all metrics, VLAB achieves smaller values than other indexes, which shows the superiority of the proposed index in the theoretic way.

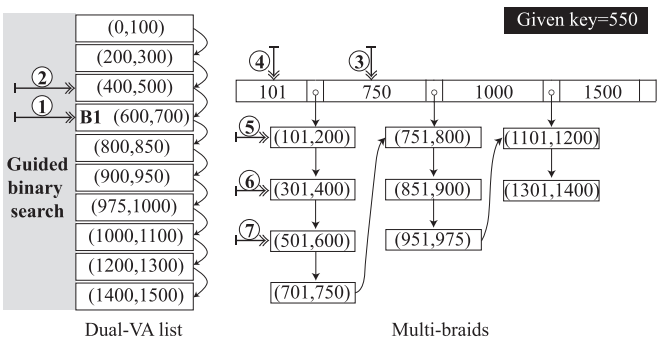


Fig. 12. Illustrations on searching entries with key 550.

5.2.1 Query Operation

The query operation is the base of all other operations. As stated, there are two structures in VLAB: Dual-VA list and Multi-braids. When query operation is conducted, we first find the given key in Dual-VA list. If we failed to find the entry, then we try to find it in multi-braids. Please note that since the ranges in Dual-VA list and Multi-braids have no intersections, query operations on Dual-VA list and Multi-braids can be conducted in parallel.

The detailed process of querying is illustrated in Fig. 12. This figure demonstrates the process to locate the bucket whose range includes the given key. First, it performs guided binary search on the Dual-VA list using the contiguous virtual address (steps ①and ②). After these two steps, we know that the bucket is not in Dual-VA list. Then, in steps ③and ④, we perform binary search on the head to multi-braid to locate the braid that may contains the bucket. Next, we sequentially search buckets in the located braid (steps ⑤to ⑦). Note that since the buckets in a braid is in order, we can stop searching if the minimum key in the bucket is larger than the given key. In this example, we located the bucket that may contains the given key after step ⑦. Finally, we get the target entry in the located bucket.

5.2.2 Insert Operation

When we insert a new entry into an index, it can be conducted in three steps: i) locate an unused position for the new entry; ii) write the new entry to the located position; iii) modify the corresponding bit in bitmap. The detailed description is given as follows.

Implementation of an insert operation
1: Locate the bucket $\mathcal{B}$ in either Dual-VA list or Multi-braids, where $min_{\mathcal{B}} < key < max_{\mathcal{B}}$.
2: Get an unused entry $\mathcal{E}$ in $\mathcal{B}$, and write (k, p) to $\mathcal{E}$.
3: `CLFLUSH`($\mathcal{E}$); `MFENCE`().
4: Modify the corresponding bit of $\mathcal{E}$ in bitmap to 1.
5: `CLFLUSH`($\mathcal{B}.bitmap$); `MFENCE`().

Note that there are three considerations on the above implementation. *First*, if no buckets can satisfy the condition in line 1, then we will create a new bucket in Multi-braids. *Second*, the `CLFLUSH` and `MFENCE` in lines 3 and 5 are

performed to guarantee that the entry is persistent in NVM before the modification of bitmap. *Third*, if the bucket $\mathcal{B}$ is full after the insertion, we need to perform the split operation. Detailed implementation of the split operation will be given later in this section.

Now, we analyze the values of N_w, N_{clf}, and N_{mf}, respectively. Since each insertion will write a key, a pointer to an unused entry and modify 1 bit in bitmap, it writes 17 bytes on NVM. Note that the NVM is byte-addressable, and thus we use byte as the basic unit to calculate the NVM writes. As for data consistency, we need two pairs of `CLFLUSH` and `MFENCE` to keep the order of writes. Therefore, for an insertion operation without split, we have $N_w = 17$, $N_{clf} = 2$ and $N_{mf} = 2$, as shown in Table 2.

5.2.3 Delete Operation

In the delete operation, we only need to modify one bit in the bitmap. The deletion of entry (k, p) can be conducted as follows.

Implementation of a delete operation
1: Locate the bucket $\mathcal{B}$ in either Dual-VA list or Multi-braids, where $min_{\mathcal{B}} < key < max_{\mathcal{B}}$.
2: Find the entry $\mathcal{E}$ whose key is equal to k.
3: Modify the corresponding bit of $\mathcal{E}$ in bitmap to 0.
4: `CLFLUSH`($\mathcal{B}.bitmap$); `MFENCE`().

Based on the above description, we have $N_w = 1$, $N_{clf} = 1$ and $N_{mf} = 1$, as shown in Table 2.

5.2.4 Split Operation

When a bucket is full, as shown in Fig. 13, we need to split it into two buckets, which involves two operations. *First*, we need to select a splitter that partitions the entries into two sets. The set of entries with larger keys needs to be moved to a new bucket. *Second*, the new bucket will be inserted into the Multi-braids.

To efficiently select a splitter, we use the idea of random sample sort [5]. First, we set a $stride$ and select $252/stride$ keys from the node, as shown in Fig. 13. Then, we choose the median of the selected keys as the splitter. By this way, roughly half of entries are partitioned into each set.

The algorithm of split operations is given as follows.

Implementation of a split operation
1: Write the last bit to 1.
2: `CLFLUSH`(); `MFENCE`().
3: Select a splitter.
4: Move entries whose keys are larger than splitter to the new bucket, write bitmap in the new bucket.
5: `CLFLUSH`(); `MFENCE`().
6: Link the new bucket to the Multi-braids.
7: `CLFLUSH`(); `MFENCE`().
8: Modify bitmap and "max" field in the original bucket.
9: `CLFLUSH`(); `MFENCE`().
10: Write the last bit to 0.
11: `CLFLUSH`(); `MFENCE`().

Note that we will utilize the last bit in bitmap as a flag to guarantee the data consistency of the split operation.

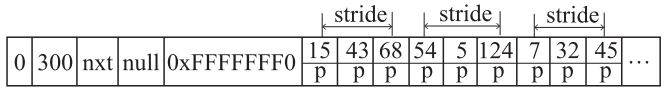


Fig. 13. The illustration of a full node.

According to the above algorithm, there are three states in split operation: (1) the created bucket is not linked to the Multi-braids, (2) the created bucket is in the Multi-braids, but the contents in the original bucket are invalid (bitmap has not been modified), and (3) split operation is successfully conducted.

When the failure occurs during the split operation, we can identify the state of index according to two evidences: (1) whether or not the last bit of bitmap is 1; (2) whether or not the created bucket can be found in the Multi-braids. Once the state is identified, we can recover the index to the correct state (i.e., state 3) by simply executing the corresponding split routine.

5.2.5 Combine Operation

The Multi-braids is an auxiliary structure to buffer buckets for Dual-VA list. When multi-braids is full, namely that its head cannot hold new braids, we need to combine it with Dual-VA list. As shown in Fig. 10, Multi-braids can be accessed as a sorted list. Hence, the combine operation is to merge two sorted lists.

In the combine operation, we use the $vnxt$ field in each bucket to guarantee data consistency. The combine operation can be conducted in 3 steps. *First*, we use $vnxt$ field to link all nodes in order. *Second*, we copy $vnxt$ to nxt for each node. *Third*, we write the $vnxt$ of each node to $null$. The procedure of combine operation is illustrated in Fig. 14, where there are 4 nodes in Dual-VA list (DL), and two nodes in the Multi-braids (MB).

Fig. 14 shows 4 states of the combine operation. When the failure occurs before state 2 shown in Fig. 14b, contents in the nxt fields are not changed. In other words, we can get the original sequence of two lists. Hence, in the recovery procedure, we can conduct the *First* step of combine operation to get into the state 2. If a failure occurs after state 2, the sequence of combined Dual-VA list can be obtained by $vnxt$ fields. In this case, we can perform *Second* step and *Third* step to complete the combine operation.

5.3 Recovery

By applying the approaches presented in Section 5.2, all buckets can be recovered to consistent state after crashes. One remaining question is how to correctly load indexes into new processes after a normal shutdown or a crash.

In VLAB, all buckets (in both Dual-VA list and Multi-braids) are stored in NVM. As with NV-Tree [39], we store the head pointers of two lists of buckets to a reserved position in NVM. In the recovery, we first load these two sorted linked lists in to the process. Then, we combine these two sorted lists. Note that the linked list in Multi-braids is small, indicating that the overhead of combining two lists is small. Finally, we build a contiguous virtual address for the combined list by invoking our implemented `build_dualva_list` system call.

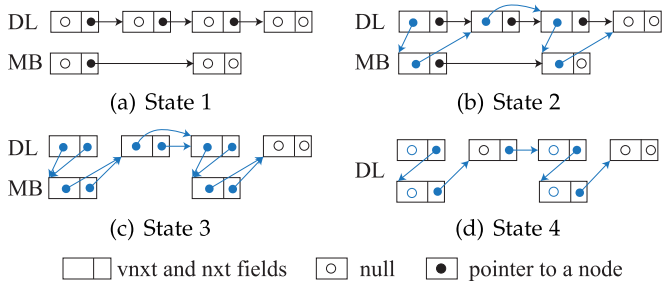


Fig. 14. Four different states in the combine operation.

5.4 Discussion on the Advantages of VLAB

VLAB can achieve smaller number of NVM writes than other indexes due to two reasons. First, VLAB employs the simplest linked list together with an auxiliary structure Multi-braids, which can dramatically reduce the NVM writes in the split processes. Second, we carefully devise each operation to use atomic write for data consistency, which avoids the heavy overhead incurred by the traditional logging or shadowing techniques.

Meanwhile, VLAB has better timing performance than other indexes, since VLAB can take full advantages of contiguous virtual address space to utilize calculations instead of memory accesses to locate buckets. By this way, we resolve the challenge that linked list requires the sequential search to locate buckets. Benefiting from the simple structures, performance overhead on data consistency is also reduced (i.e., having less `CLFLUSH` and `MFENCE` operations).

Furthermore, the memory requirements of VLAB are less than other indexes. This is because that VLAB employs the page table, instead of multiple layers of internal nodes, to organize buckets. Internal nodes in trees store key-pointer pairs to locate nodes in the next layer, while page tables in VLAB only keep pointers, eliminating the overhead of keys.

6 Experimental Results

In this section, we conduct experiments using Yahoo! Cloud Serving Benchmark (YCSB [14]). We utilize 6 workloads, including 3 workloads generated by YCSB to simulate the life cycle of a database, and 3 predefined workloads in YCSB. Properties of these workloads are given as follows:

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

Keys in all workloads are chosen from a Zipfian distribution. Finally, in Sections 6.6, 6.7, and 6.8, we use a real-world trace to evaluate the proposed techniques.

Before presenting evaluation results, we first define some useful notations. Let the number of NVM writes incurred by tree structures T be N_T, and that incurred by VLAB be N_V. When $\frac{N_T}{N_V} = \omega$, we use the notation "ωX" to indicate the degree of write reductions. For example, the numbers of NVM writes caused by traditional B$^+$-tree and VLAB are 2.64×10^{10} and 6.81×10^8, then VLAB achieves $\frac{2.64\times10^{10}}{6.81\times10^8} = 38.73$X reductions, as shown in Table 5. The speedup achieved by VLAB is defined in the similar way. For instance, in performing query operation, *VLAB* takes 26.40 seconds while traditional B$^+$-tree takes 52.52 seconds. We call that *VLAB* achieves $\frac{52.52}{26.40} = 1.99$X speedup.

As will shown in the experiments, compared with two state-of-the-art indexing schemes, *wB$^+$-tree* and *NV-Tree*, our proposed *VLAB* can achieve 6.98X and 47.53X reductions on average in the number of NVM writes, respectively. In the meantime, *VLAB* gains 2.53X and 5.29X speedup, respectively. All in all, *VLAB* has higher performance than existing indexes; meanwhile, it achieves significant reductions on NVM writes which is able to prolong the lifetime of NVM.

In the following of this section, we will show the experimental results in detail. First, we will introduce the implementation efforts and experimental setup. Then, we will report the results on conducting basic operations, including query, insert, delete and update. Finally, we will show the results on conducting mixed workloads generated by YCSB.

6.1 Implementation Efforts

We implement the Virtual Linear Addressable Buckets index in Linux based systems. We add two sets of system calls in Linux kernel, as shown in Table 3. The first set of system calls is implemented for the construction and destruction of Dual-VA list. Function `build_dualva_list` is designed to assign a contiguous virtual address for a list of pages (given by discrete virtual addresses in user program). And function `destroy_dualva_list` is to detach the mapping between contiguous virtual addresses and pages.

The second set of system calls is implemented for memory management. In the experiments, we use NVDIMM as a persistent storage device. In order to directly allocate and deallocate memory space from NVDIMM, we implement two system calls, `nvm_alloc` and `nvm_free`, according to [31]. In the buckets of *VLAB*, pointers are the offsets to the start address of the mapped memory space. Therefore, buckets can be located using the offset even if the mapping is changed after the reboot. For all index structures, the data will be stored in the NVDIMM.

Based on the added system calls, we implement a storage engine[1] for *VLAB* in MySQL [3]. The engine for *VLAB* is implemented according to MEMORY engine (a built-in engine for in-memory databases); in particular, we rewrite the interfaces related to indexing. In the implementation, our *VLAB* index is adaptive to variable sized keys. Based on our designed engine in MySQL, YCSB and real-world traces are used to conduct end-to-end performance evaluations.

In VLAB, there are three parameters need to be determined: (1) the expected search range in guided binary search, i.e., 2^α; (2) the size of the head in Multi-braids; and (3) the maximum length L of each braid.

We first discuss α. Assume there are B buckets, it requires $log_2 B$ comparisons for the typical binary search. If

1. Storage engine as a plugin can be customized in MySQL.

TABLE 3
Two Sets of System Calls to Support VLAB

Name	Parameter	Return value	Description
`build_dualva_list`	addr_ori_list, size	addr_cont_va	Build Dual-VA list from the original list
`destroy_dualva_list`	addr_cont_va, size	void	Destroy the built Dual-VA list
`nvm_alloc`	size	pg_addr	Allocate pages in NVM_ZONE
`nvm_free`	pg_addr	void	Free the allocated pages in NVM_ZONE

we expect to reduce the number of comparisons to $\frac{log_2 B}{K}$, then we just need to set $\alpha = \frac{log_2 B}{K}$. In the experiments, we would like to reduce half of comparisons compared with the typical binary search, i.e., setting $K = 2$. For a workload with 1 million of buckets. We have $\alpha = \frac{log_2(1M)}{2} = 10$.

As for Multi-braids, when the length of a braid reaches a threshold L, it needs to be split into two braids, resulting the shift of contents in the head. To improve performance on shifting, we allocate a page (4 KB) for the head. The threshold L will affect the search efficiency in worst cases. Specifically, it needs $\frac{L}{2}$ comparisons to locate a bucket. Whereas a tree needs $log_2 B$ comparisons. To guarantee performance, we set $L \leq 2 \times log_2 B$. Limited by the head size, the Multi-braids will be combined to Dual-VA list when it is "full". Since we would like to reduce the frequency of invoking combine operation, we set $L = 2 \times log_2 B$. A Multi-braids is full when it contains $L \times 50\% \times N$ buckets, where 50 percent is the load rate of braids in worst-case situation and N represents the maximum number of braids.

6.2 Experimental Setup

We compare *VLAB* with 5 existing indexes, including "Traditional B$^+$-Tree" (*tradB$^+$*), "Bitmap-based B$^+$-Tree" (*bmpB$^+$*) [9], "Counter-based B$^+$-Tree" (*cntB$^+$*) [9], and two state-of-the-art structures "wB$^+$-tree" (*wB$^+$*) [10] and *NV-Tree* [38], [39]. The node (bucket) size of all indexes is set as 4K, and we will use different entry (key) sizes in our experiments. In the comparison, we consider three metrics: (1) the running performance, represented by elapsed time, (2) the number of NVM writes, recorded in the run time, (3) the number of extra instructions for data consistency, including `CLFLUSH` and `MFENCE`.

All experiments run on a workstation with Intel(R) Xeon (R) E5-2630 at 2.40 GHz and 128 GB memory. The workstation also contains 8 GB NVDIMM, which has similar read/write latency as DRAM. In the comparison, we implement all index structures in MySQL engines, and we use YCSB (0.12.0) for performance evaluation. Note that the experiments are conducted on a NVDIMM workstation instead of simulation. Specifically, all data of indexes are stored in NVDIMM using our added system calls.

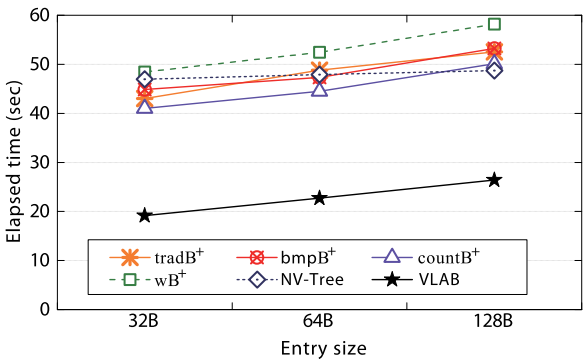


Fig. 15. Query performance comparison with workload *QueryLoad*.

6.3 Query Performance

Query is the basic operation of an index, which is the base of other operations. Therefore, the speed of the query operation will significantly affect the index performance. We compare the query performance with three different entry sizes: 32 B, 64 B and 128 B. Different entry sizes indicate that the number of entries in a node is different. Fig. 15 reports the comparison results. The workload *QueryLoad* is generated by YCSB, which first loads 50 million entries and then perform 50 million query operations. Keys in these operations follow the Zipfian distribution.

From the results in Fig. 15, we find that the elapsed time increases with the entry size. Since the size of nodes keeps unchanged, the increase of entry size will lead to more leaf nodes. In consequence, the height of trees may be increased, resulting in more comparisons in locating leaf nodes. In this experiment, the heights of trees (except NV-Tree) are 4, 5, and 6 corresponding to entry sizes of 32 B, 64 B, and 128 B, respectively. The growing trend of NV-Tree is slower than others, since the heights of NV-Tree are 5 and 6 for entry sizes of 32 B and 64 B which are larger than other trees, while its height remains 6 when entry size is 128 B. This is because each parent of leaf nodes in NV-Tree only keeps one child after rebuilding [38], resulting in larger height if there exist rebuild operations. For VLAB, the number of comparisons in the search is also increased with the number of nodes, resulting in the similar trend as trees.

Results show that *VLAB* achieves roughly 2X speedup compared with other indexes for all sized entries. The detailed figures compared with competitors are concluded in Table 5. From the above results, we observed that *wB$^+$* takes the largest elapsed time to perform queries. This is because it uses an auxiliary structure to store the order of entries, which brings performance overhead.

The performance improvement achieved by *VLAB* is attributed to the reduction on the number of comparisons in locating buckets. As shown in Table 4, by employing the proposed guided binary search, *VLAB* takes 7 to 10 comparisons on average to locate the final bucket in each query operation. On the contrary, the tree-based structures perform more than 20 comparisons. Namely, compared with other indexes, our *VLAB* can achieve 64.52, 64.45, and 59.69 percent reductions on the number of comparisons for 32 B, 64 B, and 128 B entry sizes, respectively.

TABLE 4
Number of Comparisons in Locating the Final Node (Bucket) when Performing Workload *QueryLoad*

Entry size	Tree structures		VLAB		
	Total	Average	**Total**	**Average**	**Reduction**
32 B	887,396,714	22.18	**314,878,413**	**7.87**	**64.52%**
64 B	971,732,960	24.29	**345,460,179**	**8.64**	**64.45%**
128 B	1,060,161,537	26.50	**427,330,265**	**10.68**	**59.69%**

Note that the proposed search algorithm is based on binary search, which can not be applied for all structures. Because it requires to retrieve information by randomly accessing computed nodes (buckets). In *VLAB*, the Dual-VA list provides the contiguous virtual addresses for buckets. Given an offset, it only takes $O(1)$ time complexity to access the bucket. In contrast, leaf nodes of other indexes are organized in linked list, which takes $O(n)$ time complexity to access the bucket with a given offset. In consequence, binary search cannot be efficiently performs on tree-based structures. The above results demonstrate the advantages of organizing nodes in contiguous virtual address spaces.

6.4 Insertion, Deletion and Update

Insertion, deletion and update are three basic operations in indexes. These operations will incur NVM writes. This section evaluates *VLAB* by performing these basic operations. We use YCSB to generate test workloads *InsertLoad*, *DeleteLoad* and *UpdateLoad*. In these workloads, it first loads 50 M data, and then perform 50 M corresponding operations (e.g., 50 M insertions for *InsertLoad*). In these operations, keys are chosen from a Zipfian distribution.

6.4.1 Number of NVM Writes

Fig. 16 reports the number of NVM writes of *VLAB* and its competitors for basic operations. In these figures, the x-axis represents three different entry sizes, and y-axis represents the number of NVM writes. Note that we use logarithmic scale on y-axis for ease of presentation. Results clearly illustrate that *VLAB* can dramatically reduce the number of NVM writes over competitors.

tradB$^+$ incurs large quantities of NVM writes in both insertion and deletion. One reason is that all nodes in *tradB*$^+$ must be sorted in order, which incurs a lot of writes in moving entries. Another reason is that the logging for consistency doubles the NVM writes. As shown in Table 5, *VLAB* achieves 38.73X, 689.42X, and 3X reductions for workloads *InsertLoad*, *DeleteLoad* and *UpdateLoad*, respectively.

bmpB$^+$ [9] and *cntB*$^+$ [9] achieve similar results on the number of NVM writes. Both structures utilize the unsorted leaf nodes. Hence, the NVM writes can be reduced, compared with the *tradB*$^+$. However, the number of NVM writes is still larger than *VLAB*. There are three reasons: (1) it contains extra internal nodes in NVM, (2) the split and merge operations incur a lot of writes on the internal nodes; (3) the logging method for consistency incurs extra writes. As shown in Table 5, compared with *cntB*$^+$, the reductions on NVM writes achieved by *VLAB* are 7.55X, 75.66X and 3X for insert, delete, and update operations. Compared with *bmpB*$^+$, these figures are 9.65X, 106.58X and 3X.

wB$^+$ [10]. It is not surprise that *wB*$^+$ has less NVM writes than other tree-based indexes, because it employs atomic write for data consistency and it aims to minimize the NVM writes. However, results show that, for insertions and deletions, the number NVM writes of *wB*$^+$ is 2.68X and 35.01X

TABLE 5
Times of Reductions on the Number of NVM Writes and Elapsed Time Achieved by *VLAB* for Basic Operations Using 128 B Entry Size

Workloads	v.s. *tradB*$^+$		v.s. *bmpB*$^+$		v.s. *cntB*$^+$		v.s. *wB*$^+$		v.s. *NV-Tree*	
	WRT Red.	Speedup	WRT Red.	Speedup	WRT Red.	Speedup	WRT Red.	Speedup	WRT Red.	Speedup
QueryLoad	-	1.99X	-	2.02X	-	1.90X	-	2.20X	-	1.85X
InsertLoad	38.73X	5.96X	7.55X	3.05X	9.65X	2.45X	2.68X	1.89X	3.98X	4.49X
DeleteLoad	689.42X	5.78X	75.66X	3.86X	106.58X	3.16X	35.01X	3.33X	56.33X	7.92X
UpdateLoad	3.00X	1.70X	3.00X	1.88X	3.00X	1.54X	1.00X	1.85X	95.15X	29.80X

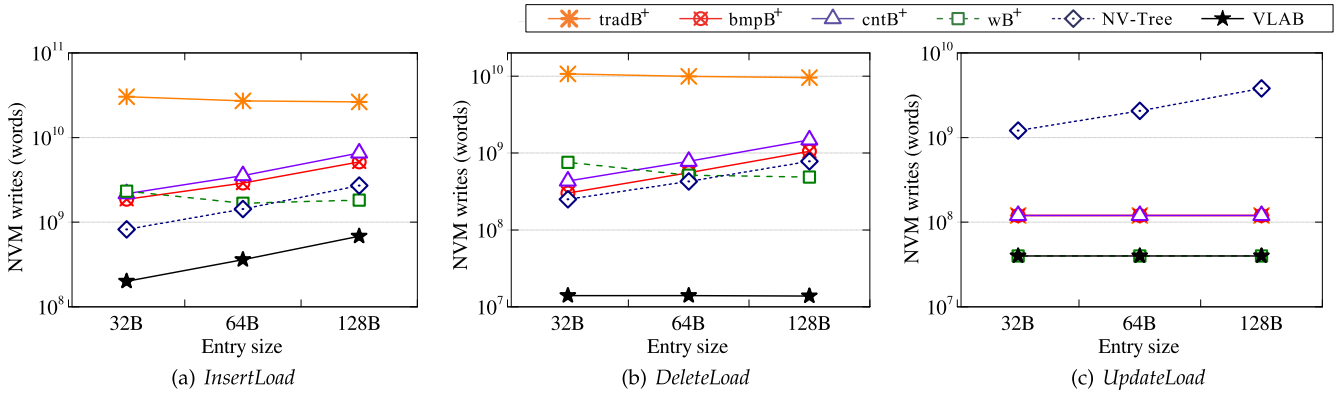


Fig. 16. Comparisons on the number of NVM writes (words) for different indexing schemes by conducting insertion, deletion, and update operations with different entry sizes: (a) Insertions; (b) deletions; (c) updates.

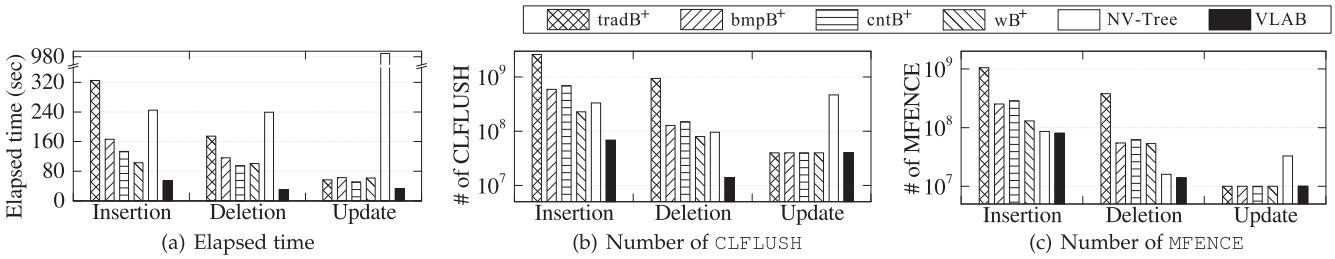


Fig. 17. Comparisons on the performance and overhead on data consistency in performing insertion, deletion and update with 128 B entry size: (a) Performance results; (b) CLFLUSH; (c) MFENCE.

more than *VLAB*. Like *cntB*$^+$ and *bmpB*$^+$, split operations in *wB*$^+$ may cause a large number of NVM writes. As for update operation, since both *VLAB* and *wB*$^+$ employ atomic write to update the pointer in an entry, they have the same number of NVM writes.

NV-Tree [38], [39] requires extra information in entries for data consistency, which incurs a lot of NVM writes. Specifically, for the update operation, it continually performs a delete operation and an insert operation, which insert two new entries into the leaf node. Therefore, it has more NVM writes than others. Compared with *NV-Tree*, our *VLAB* achieves 3.98X, 7.92X, and 29.80X reductions on NVM writes for insert, delete, and update operations, respectively.

As for our proposed *VLAB*, *first*, it only stores a linked list in NVM, which can dramatically reduce NVM writes caused by the changes on structures. *Second*, it only maintains a small number of metadata (i.e., bitmap) in each bucket, and the entries in a bucket are unsorted, both of which eliminate the extra writes on NVM. *Third*, it takes full advantages of atomic write to avoid the use of logging and shadowing for data consistency. Therefore, *VLAB* achieves significant reductions on NVM writes. For NVM-related products, reductions in NVM writes indicate the prolonged lifetime. Hence, *VLAB* is practical for NVM-based systems.

From the above experimental results, we observed that the trends of the results for different entry size are almost the same. For ease of presentation, we only show the results of 128B entry size in the following experiments.

6.4.2 Performance Comparison

Fig. 17a reports the performance comparison for insert, delete, and update operations when the entry size is 128 B. From the results, it is clear to see that *VLAB* performs much better than the competitors for each operation.

As for *insert and delete operations*, *tradB*$^+$ use logging approach for data consistency, which involves a lot of extra load/store operations. Hence, the elapsed time for *tradB*$^+$ is larger than other indexes; in particular, it is 5.96X and 5.78X larger than *VLAB* for insertion and deletion, respectively. In addition, *VLAB* performs better than other four indexes. Specifically, as for insertion, *VLAB* achieves 3.05X, 2.45X, 1.89X and 4.49X speedup, compared with *bmpB*$^+$, *cntB*$^+$, *wB*$^+$ and *NV-Tree*, respectively. As for deletion, these figures are 3.86X, 3.16X, 3.33X and 7.92X. The significant reductions on elapsed time obtained by *VLAB* are mainly because of the following two reasons: (1) the search process of *VLAB* is faster than others; (2) the performance degradation caused by guaranteeing data consistency for *VLAB* is smaller than others. We will show the overhead for data consistency later.

As for *update operations*, *bmpB*$^+$, *cntB*$^+$, *wB*$^+$, and *VLAB* only modify the value in an entry. Therefore, the elapsed time is dominated by the query performance. From the results, we can see that *VLAB* can achieve roughly 1.8X speedup. On the contrary, the update operation in *NV-Tree* is composed of one insert operation and one delete operation. Therefore, it performs worst in all competitors; in particular, *VLAB* is 29.80X faster than *NV-Tree*.

6.4.3 Data Consistency Overhead

Figs. 17b and 17c compare the data consistency overhead. Note that we use logarithmic scale on y-axis to represent the number of flush and fence instructions. Results show that *VLAB* achieves the minimum number of CLFLUSH and MFENCE compared with its competitors.

As for CLFLUSH, *VLAB* reduces total CPU flushes by 38.18X-281.17X compared to *tradB*$^+$, 8.74X-35.53X compared to *bmpB*$^+$, 10.21X-39.99X compared to *cntB*$^+$, 3.35X-15.51X compared to *wB*$^+$, and 3.85X-15.15X compared to *NV-Tree*.

As for MFENCE, the reductions on total fence operations obtained by *VLAB* are 24.54X compared to *tradB*$^+$, 3.15X compared to *bmpB*$^+$, 3.58X compared to *cntB*$^+$, 1.63X compared to *wB*$^+$, and 6.7 percent to *NV-Tree*.

Table 7 demonstrates the performance degradation caused by data consistency when performing insert operations. Results reflect that the extra CLFLUSH and MFENCE instructions can heavily degrade the system performance. For *VLAB*, without extra CLFLUSH and MFENCE instructions,the elapsed time can reduced to 36.59 from 54.53. In addition, we observe that the performance degradation on *VLAB* is much less than that on others. It verifies that our proposed consistency mechanism performs better than others.

6.5 Mixed Workloads

Fig. 18 reports the performance of six indexing schemes using six different YCSB workloads, including *GrowPhase*, *AdultPhase*, *ElderPhase*, *Workload-A*, *Workload-B*, and *Workload-D* (see the beginning of this section). The x-axis represents the elapsed time of running stage, and the y-axis using logarithmic scale represents the number of writes on NVM. Results in this figure clearly demonstrate that our proposed *VLAB* can achieve the best running performance while minimizing the number of writes on NVM.

As for the insertion intensive workloads (*GrowPhase*, *Workload-D*) and the deletion intensive workload (*ElderPhase*), *VLAB* has much less NVM writes over competitors.

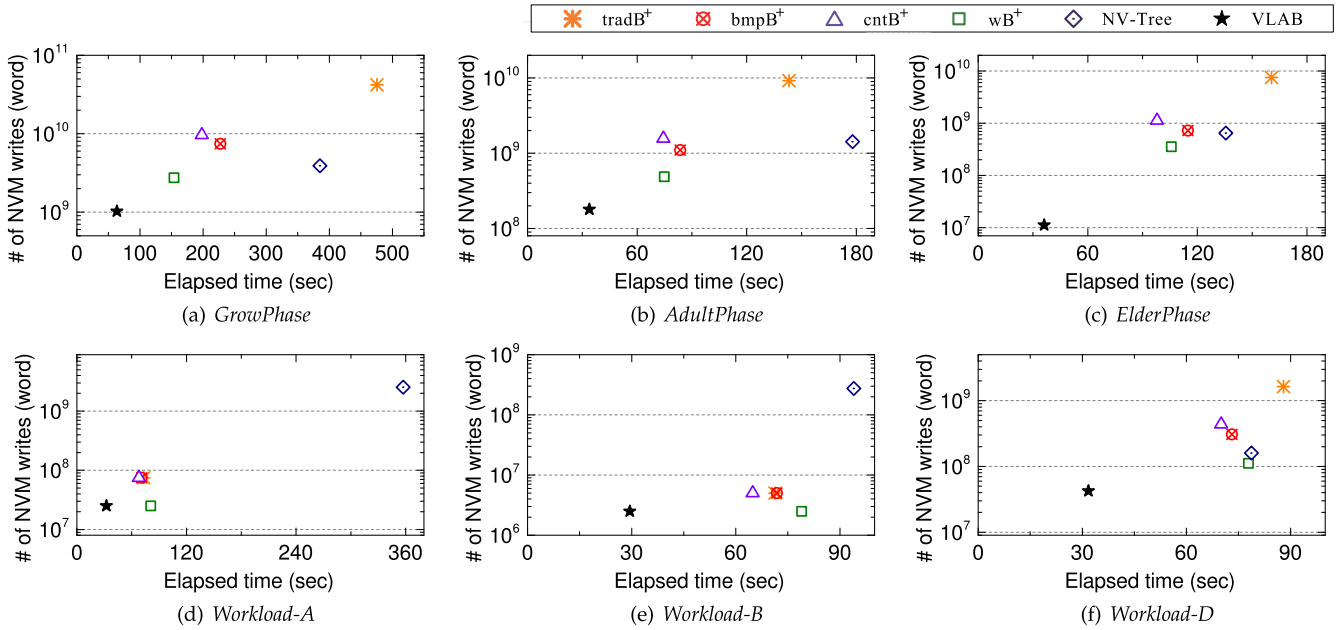


Fig. 18. Comparisons on the number of NVM writes (words) and elapsed time for different indexing schemes for mixed workloads: (a) *GrowPhase*; (b) *AdultPhase*; (c) *ElderPhase*; (d) *Workload-A*; (e) *Workload-B*; (f) *Workload-D*.

Meanwhile, the performance of *VLAB* is better than others. These results are accordance with the previous analysis.

As for the update intensive workloads (*Workload-A*, *Workload-B*), *VLAB* and *wB*$^+$ have the same number of NVM writes. However, *NV-Tree* incurs much more NVM writes than others, meanwhile, having the worst timing performance. This is because that *NV-Tree* implements update operation by using one insertion and one deletion sequentially.

As for workload *AdultPhase*, the number of query, update, delete, and insert operations are the same. Fig. 18b illustrates the results of this workload. We can see that *tradB*$^+$ has the largest number of NVM writes, while *NV-Tree* takes more elapsed time to finish these operations. Since *VLAB* outperforms other indexes for each operation, *VLAB* has the best overall system performance on this workload.

From the above results, we can conclude that our proposed *VLAB* is efficient and practical for real world workloads. Specifically, it achieves significant reduction in NVM writes and improvement on timing performance, compared with the state-of-the-art indexes. In the following sections, we will give the detailed data for these workloads.

Table 6 summaries the experiments on mixed workloads. In the table, columns "WRT Red." represent the times of reductions on NVM writes achieved by *VLAB*. And columns "Speedup" represent the improvement on timing performance. For workload *GrowPhase* under column " v.s. *tradB*$^+$", the number under column "WRT Red." is 40.92X, which indicates that *VLAB* achieves 40.92X reductions on NVM writes, compared with *tradB*$^+$. The number under column "Speedup" is 7.51X, indicating that *VLAB* achieves 7.51X reductions on elapsed time.

6.6 Real-World Trace Evaluations on Throughput

In this section, we employ five real-world traces to evaluate the performance of indexes, including *Pumsb* [2] (49,046 trans/73.62 items), *Pumsb_star* [2] (49,046 trans/50.10 items), *Retail* [6] (88,162 trans/10.30 items), *Kosarak* (990,002 trans/8.10 items) [2], and *WebDoc* [23] (15,291,728 trans/19.61 items), where "trans/items" indicates the total number of transactions and the average length of each transaction.

We execute these traces by using different indexes to compare their throughput (the number of transactions per

TABLE 6
Times of Reductions on the Number of NVM Writes and Elapsed Time Achieved by *VLAB* for Mixed Workloads Using 128 B Entry Size

Workloads	v.s. *tradB*$^+$		v.s. *bmpB*$^+$		v.s. *cntB*$^+$		v.s. *wB*$^+$		v.s. *NV-Tree*	
	WRT Red.	Speedup	WRT Red.	Speedup	WRT Red.	Speedup	WRT Red.	Speedup	WRT Red.	Speedup
GrowPhase	40.92X	7.51X	7.26X	3.59X	9.38X	3.12X	2.68X	2.43X	3.79X	6.08X
AdultPhase	51.43X	4.23X	6.16X	2.47X	8.74X	2.20X	2.73X	2.21X	7.97X	5.26X
ElderPhase	670.82X	4.43X	65.04X	3.17X	100.82X	2.71X	31.88X	2.92X	57.91X	3.75X
Workload-A	3.00X	2.26X	3.00X	2.19X	3.00X	2.09X	1.00X	2.49X	101.48X	11.01X
Workload-B	3.00X	2.42X	3.00X	2.44X	3.00X	2.20X	1.00X	2.68X	110.29X	3.19X
Workload-D	38.61X	2.77X	7.23X	2.30X	10.29X	2.20X	2.62X	2.45X	3.75X	2.48X
Average	**134.63X**	**3.94X**	**15.28X**	**2.69X**	**22.54X**	**2.42X**	**6.98X**	**2.53X**	**47.53X**	**5.29X**

TABLE 7
Performance Degradation (secs) Caused by Data Consistency in Performing Insert Operations Using 128 B Entry

	with consistency	w/o consistency	Degradation
tradB$^+$	324.79	79.22	245.57
bmpB$^+$	166.42	100.15	66.27
cntB$^+$	133.38	71.95	61.43
wB$^+$	102.93	80.45	22.48
NV-Tree	244.97	71.31	173.66
VLAB	**54.53**	**36.59**	**17.94**

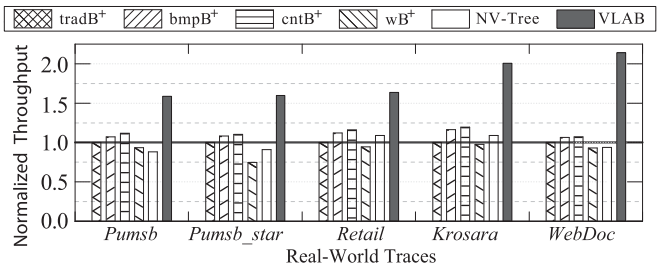


Fig. 19. Comparisons of real-world traces in normalized throughput.

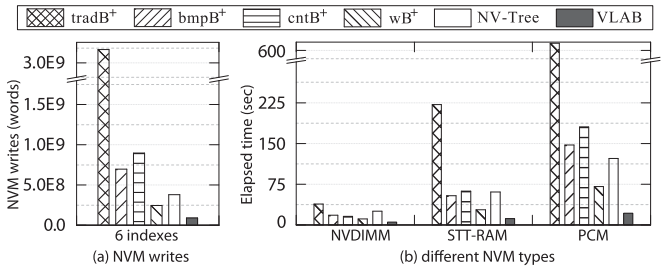


Fig. 20. Comparisons of real-world trace WebDoc [23]: (a) NVM writes; (b) elapsed times on different types of NVMs.

TABLE 8
Comparisons on VLAB and Tree-Based Indexes in the Memory Usage

Workloads	Essential Mem (Data)	Extra Mem—Trees		Extra Mem—VLAB		
		Internal Nodes	(%)	Multi Braids	Page Table	(%)
InsertLoad	17.74 GB	765.64 MB	4.04%	**4 KB**	**35.56 MB**	**0.19%**
DeleteLoad	9.88 GB	384.43 MB	3.66%	**4 KB**	**19.82 MB**	**0.20%**
WebDoc	884.00 MB	47.43 MB	5.09%	**4 KB**	**1.72 MB**	**0.19%**

second). In order to clearly illustrate the comparison results, we normalize the throughput, using *tradB*$^+$ as the baseline. Fig. 19 reports the results. As shown in this figure, VLAB outperforms other indexes significantly. Specifically, for traces *Pumsb*, *Pumsb_star* and *Retail*, VLAB achieves 1.6X higher throughput than *tradB*$^+$ on average. For *Kosarak* and *WebDoc*, the improvements on throughput reach up to 2.1X.

6.7 Performance with Different Types of NVMs

We compare the elapsed time using the real-world trace, WebDoc [23], in terms of different types of NVMs. As with the evaluations in NV-Tree [39], we explicitly add extra delay before every memory writes to show the performance improvements with different types of NVMs: NVDIMM (same as DRAM), STT-RAM (50 ns), and PCM (180 ns).

Fig. 20 reports the comparison results. Results show that VLAB can obtain higher speedup than competitors with the increases of write latency, since VLAB has less NVM writes than others, as shown in Fig. 20a. Specifically, using NVDIMM, the speedups obtained by VLAB are 7.54X, 3.53X, 3.07X, 2.19X, and 4.95X against *tradB*$^+$, *bmpB*$^+$, *cntB*$^+$, *wB*$^+$, and *NV-Tree*, respectively. These figures are increased to 19.21X, 4.67X, 5.40X, 2.43X, and 5.28X for STT-RAM. When the write latency is increased to 180 ns (PCM), these figures reach up to 31.66X, 6.79X, 8.33X, 3.26X and 5.64X.

6.8 Memory Requirements

In an index, the memory used to store the entries is called "essential memory", while the memory used to store structures for locating buckets (nodes) is called "extra memory". Since the number of entries in different indexes is the same, we focus on the usage of extra memory. In a tree-based structure, extra memory is used to store internal nodes, which keep key-pointer pairs. Whereas, in VLAB, we build a page table for Dual-VA list, which only keeps pointers. Hence, the extra memory requirement of VLAB is smaller. Note that VLAB allocates a head in Multi-braids to accelerate the search process, which is a constant (4 KB in our experiments).

Table 8 reports the memory usage of tree-based structures and VLAB, where the entry size is 128 B. Results clearly demonstrate that VLAB has much less memory requirement than trees. Specifically, the ratio of extra memory in overall memory usage of VLAB is roughly 0.2 percent. In contrast, the ratio of tree-based structures reaches up to 5 percent.

7 Conclusion

This paper studied what is and how to design an NVM-friendly index structure. Specifically, we presented an efficient and consistent indexing scheme, called "**V**irtual **L**inear **A**ddressable **B**uckets". The efforts and understandings that we presented in this paper might be useful for the design of any NVM-friendly data structures. To minimize the number of NVM writes, the basic structure of VLAB is the simplest data structure, i.e., linked list. Since the linked list suffers from inefficiency in searching, this paper presented a new technique, called Dual-VA, to employ the binary search on it. For a list of buckets, Dual-VA technique builds a contiguous virtual address space in addition to the default non-linear one. In the experiments, we implemented VLAB in Linux-based systems and implanted it into MySQL. The evaluation was conducted under YCSB and real-world traces. Results show that VLAB can significantly reduce the number of NVM writes, meanwhile achieving higher performance, compared with state-of-the-art indexes.

Acknowledgments

The authors are grateful to the anonymous reviewers for their constructive comments. This work is partially supported by National 863 Program [Grant No. 2015AA015304], National Natural Science Foundation of China [Grant No. 61472052], China Scholarship Council [No. 201706050116].

REFERENCES

[1] 3D Xpoint technology revolutionizes storage memory. (2015). [Online]. Available: http://www.intel.com

[2] Frequent itemset mining dataset repository. (2004). [Online]. Available: http://fimi.ua.ac.be/data/

[3] Mysql. (1995). [Online]. Available: https://www.mysql.com/

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 707–722.

[5] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A comparison of sorting algorithms for the connection machine CM-2," in *Proc. ACM Symp. Parallel Algorithms Archit.*, 1997, pp. 3–16.

[6] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets, "Using association rules for product assortment decisions: A case study," in *Proc. 5th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 1999, pp. 254–260.

[7] G. W. Burr, et al., "Phase change memory technology," *J. Vacuum Sci. Technol. B*, vol. 28, no. 2, pp. 223–262, 2010.

[8] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM J. Res. Develop.*, vol. 52, pp. 449–464, 2008.

[9] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proc. Biennial Conf. Innovative Data Syst. Res.*, 2011, pp. 21–31.

[10] S. Chen and Q. Jin, "Persistent B+-trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[11] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 347–357.

[12] J. Coburn, et al., "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 105–118, 2011.

[13] J. Condit, et al., "Better I/O through byte-addressable, persistent memory," in *Proc. ACM 22nd Symp. Operating Syst. Principles*, 2009, pp. 133–146.

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

[15] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor, "A prolegomenon on OLTP database systems for non-volatile memory," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 57–63, 2014.

[16] S. R. Dulloor, et al., "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, Art. no. 15.

[17] H.-W. Fang, M.-Y. Yeh, P.-L. Suei, and T.-W. Kuo, "An adaptive endurance-aware-tree for flash memory storage systems," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2661–2673, Nov. 2014.

[18] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious B+-trees," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, pp. 283–294, 2003.

[19] I. Intel, "Intel 64 and IA-32 architectures software developers manual," *Volume 3A: Syst. Program. Guide, Part*, vol. 1, 2013, Art. no. 64.

[20] S. Kannan, M. Qureshi, A. Gavrilovksa, and K. Schwan, "Energy aware persistence: Reducing energy overheads of memory-based persistence in NVMs," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2016, pp. 165–177.

[21] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving journaling of journal anomaly in android I/O: Multi-version B-tree with lazy split," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 273–285.

[22] I.-H. Lee, J. Shim, S.-G. Lee, and J. Chun, "CST-Trees: Cache sensitive T-trees," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2007, pp. 398–409.

[23] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "WebDocs: A real-life huge transactional dataset," in *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations*, 2004.

[24] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proc. 1st ACM SIGOPS Conf. Timely Results Operating Syst.*, 2013, pp. 15–22.

[25] S. T. On, H. Hu, Y. Li, and J. Xu, "Lazy-update B+-tree for flash devices," in *Proc. 10th Int. Conf. Mobile Data Manage.: Syst. Services Middleware*, 2009, pp. 323–328.

[26] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 265–276, 2014.

[27] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 14–23.

[28] J. Rao and K. A. Ross, "Making B+-trees cache conscious in main memory," *ACM SIGMOD Rec.*, vol. 29, pp. 475–486, 2000.

[29] S. Raoux, et al. "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, pp. 465–479, 2008.

[30] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 286–297, 2011.

[31] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for NVRAM," in *Proc. ADMS@VLDB*, 2015, pp. 61–72.

[32] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Trans. Comput.*, vol. 65, no. 10, pp. 2959–2972, Oct. 2016.

[33] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang, "In-memory databases: Challenges and opportunities from software and hardware perspectives," *ACM SIGMOD Rec.*, vol. 44, no. 2, pp. 35–40, 2015.

[34] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 61–75.

[35] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[36] X. Wu and A. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 39.

[37] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 323–338.

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

[39] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He, "NV-Tree: A consistent and workload-adaptive tree structure for non-volatile memory," *IEEE Trans. Comput.*, vol. 65, no. 7, pp. 2169–2183, Jul. 2016.

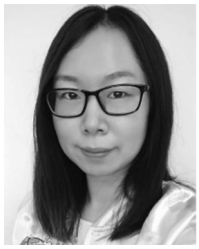
[40] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.-Dig. Tech. Papers*, 2009, pp. 264–268.

Edwin Hsing-Mean Sha received the PhD degree from the Department of Computer Science, Princeton University, in 1992. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering, University of Notre Dame. Since 2000, he has been a tenured full professor with the University of Texas at Dallas. Since 2012, he served as the dean of College of Computer Science, Chongqing University, China. He has published more than 390 research papers in refereed conferences and journals. He received Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award, and NSFC Overseas Distinguished Young Scholar Award, Chang-Jiang Honorary Chair Professorship and China Thousand-Talent Program. He is a senior member of the IEEE.

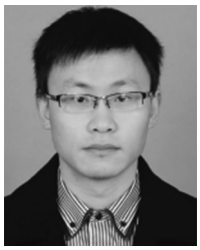
Weiwen Jiang received the BE degree from the Department of Computer Science, Nanjing Agricultural University, Nanjing, China, in 2012. He is currently working towards the PhD degree in the Department of Computer Science, Chongqing University, Chongqing, China, under the supervision of Dr. Edwin H.-M. Sha. Concurrently, he is a visiting scholar with the Department of Electrical and Computer Engineering, the University of Pittsburgh, Pittsburgh. His research interests include high-level synthesis, real-time systems, supply-chain management, non-volatile memories, and optimization algorithms. He is a student member of the IEEE.

Hailiang Dong received the BS degree from the Department of Mathematics and Statistics, Chongqing University, Chongqing, China, in 2015. He is currently working towards the master's degree in the Department of Computer Science, Chongqing University, Chongqing, China. His current research interests include non-volatile memory, embedded systems, and optimization algorithms.

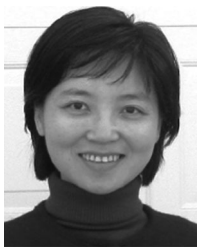
Runyu Zhang received the BE degree from the College of Computer Science, Chongqing University, Chongqing, China, in 2017. He is currently working towards the PhD degree in the Department of Computer Science, Chongqing University, Chongqing, China. His current research interests include non-volatile memory, parallel architectures, and optimization algorithms.



Zhulin Ma received the BE degree from the College of Computer Science, Chongqing University, Chongqing, China, in 2016. She is currently working towards the PhD degree in the Department of Computer Science, Chongqing University, Chongqing, China. Her current research interests include database design, non-volatile memory, and optimization algorithms.



Xianzhang Chen received the BS and MS degrees from the School of Computer Science and Engineering, Southeast University, Nanjing, China, and the PhD degree from the College of Computer Science, Chongqing University, in 2017. He is now a faculty with Chongqing University. His research interest include non-volatile memory-based file systems, memory management, and in-memory databases.



Qingfeng Zhuge received the BS and MS degrees in electronics engineering from Fudan University, Shanghai, China and the PhD degree from the Department of Computer Science, University of Texas at Dallas, in 2003. She is currently a professor at East China Normal University, China. She received Best PhD 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. She is a member of the IEEE.