Enabling Write-Reduction Multiversion Scheme With Efficient Dual-Range Query Over NVRAM

I-Ju Wang, Yu-Pei Liang, *Member, IEEE*, Tseng-Yi Chen, *Member, IEEE*, Yuan-Hao Chang, *Senior Member, IEEE*, Bo-Jun Chen, Hsin-Wen Wei, *Member, IEEE*, and Wei-Kuan Shih, *Member, IEEE*

***Abstract*—Due to cyber-physical systems, a large-scale multiversion indexing scheme has garnered significant attention in recent years. However, modern multiversion indexing schemes have significant drawbacks (e.g., heavy write traffic and weak key- or version-range-query performance) while being applied to a computer system with a nonvolatile random access memory (NVRAM) as its main memory. Unfortunately, with the considerations of high memory cell density and zero-static power consumption, NVRAM has been regarded as a promising candidate to substitute for dynamic random access memory (DRAM) in future computer systems. Therefore, it is critical to make a multiversion indexing scheme friendly for an NVRAM-based system. For tackling this issue with modern multiversion indexing schemes, this article proposes a write-reduction multiversion indexing scheme with efficient dual-range queries. According to the experiments, our scheme effectively reduces the amount of write traffic generated by the multiversion indexing scheme to NVRAM. It offers efficient dual-range queries by consolidating the proposed version forest and the multiversion tree.**

***Index Terms*—Efficient query, multiversion indexing scheme, nonvolatile memory, write reduction.**

I. Introduction

IN CYBER-PHYSICAL applications, multiversion database systems have become the most critical data management tool for analyzing and mining. As all data in a multiversion database will be indexed by a multiversion indexing scheme, boosting the indexing scheme's query performance will also increase the multiversion database's performance. On the other hand, as nonvolatile random access memory (NVRAM) is recognized as a memory/storage technology for a computer system in the next decade, NVRAM features should be considered in the design of an NVRAM-friendly multiversion indexing scheme. However, almost all multiversion indexing schemes have two critical unsolved issues, while they are adopted to an NVRAM-based computer system.[1] First, the asymmetric read/write cost of NVRAM technologies has not been considered in developing a multiversion indexing scheme. As a multiversion indexing scheme will rewrite duplicate information for achieving an efficient multiversioned data query, it will generate a massive amount of write traffic to an NVRAM space. Second, all multiversion indexing schemes cannot simultaneously provide efficient key- and version-range queries. Note that the second unsolved issue is not only on an NVRAM-based system but also on general system architectures. By such motivations, this study aims to develop a write-reduction multiversion indexing scheme with efficient dual-range (i.e., key- and version-range) queries for an NVRAM-based system.

As some applications (e.g., cyber-physical systems, surveillance, and crowd computing [15]) will periodically collect data from end devices, collected data will be managed as different versions in a storage system. Such data management will become an urgent problem for database system designs. For dealing with such version data, multiversion database systems employ a multiversion indexing scheme, whose purpose is to effectively maintain the index of version data in a multiversion database. For enhancing the query performance of the multiversion indexing scheme, Backer *et al.* [2] proposed a multiversion B$^+$-tree, namely MVBT, that is developed based on the B$^+$-tree structure. Although MVBT offers asymptotically optimal query performance, the asymmetric read/write cost has not been studied in its design since NVRAM is regarded as a next-generation memory/storage technology in the past few years. With the consideration of NVRAM's features, Kuan *et al.* [4], [5] presented a space-efficient multiversion indexing scheme, called SEMI, that aimed to improve the space utilization and the version-range-query performance of

Manuscript received August 26, 2020; revised December 15, 2020 and March 3, 2021; accepted March 14, 2021. This work was supported in part by Academia Sinica under grant no. AS-CDA-107-M05 and Ministry of Science and Technology under grant nos. 107-2218-E-008-023 -MY3, 108-2221-E-008 -099-MY3, 109-2218-E-002-019, 107-2923-E-001-001-MY3, 108-2221-E-001-001-MY3, and 108-2221-E-001-004-MY3. *(Corresponding authors: Tseng-Yi Chen; Yuan-Hao Chang.)*

I-Ju Wang, Bo-Jun Chen, and Wei-Kuan Shih are with the Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan, and also with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan.

Yu-Pei Liang and Yuan-Hao Chang are with the Institute of Information Science, Academia Sinica, Taipei 115, Taiwan (e-mail: johnson@iis.sinica.edu.tw).

Tseng-Yi Chen is with the Computer Science and Information Engineering Department, National Central University, Taoyuan 320, Taiwan (e-mail: tychen@g.ncu.edu.tw).

Hsin-Wen Wei is with the Department of Electrical Engineering, Tamkang University, New Taipei 251, Taiwan.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2021.3072233.

Digital Object Identifier 10.1109/TVLSI.2021.3072233

[1]A computer architecture includes NVRAM to be its memory or storage.

MVBT scheme on NVRAM storage. Nevertheless, the SEMI is targeted on an embedded multiversion indexing scheme, so it cannot be applied to the database system with many data keys. In other words, the SEMI cannot yield efficient key-range-query performance. Consequently, to the best of our knowledge, no multiversion indexing scheme can offer efficient version- and key-range queries simultaneously.

For enabling efficient dual-range queries in a multiversion indexing scheme, this study proposes a write-reduction multiversion indexing scheme with efficient dual-range queries, namely Duery. In our solution, Duery empowers efficient dual-range queries in a multiversion indexing scheme by developing and integrating a multiversion tree structure and version forest. While users want to perform an exact-match query and a key-range query, Duery will dispatch the user request to the multiversion tree structure because the multiversion tree can provide high key-range-query performance. Differently, the version forest will deal with a version-range query because it aggregates the version data belonging to the same index together. With the consideration of the asymmetric read/write cost, Duery brings a pointer-based split mechanism to reduce the amount of write traffic to NVRAM so as to decrease the write latency and energy consumption. According to our experimental results, Duery can effectively reduce the amount of write traffic to NVRAM storage compared with a naive hybrid indexing scheme.

To better understand the contributions of this article, we listed the achievements of this article as follows.

1) This article first identifies the problem with dual-range (i.e., key- and version-range) query efficiency in a multiversion database indexing scheme over an NVRAM-based computer system.
2) The proposed dual-range query (referred to as Duery for short) jointly considers dual-range-query efficiency and write reduction in the design of NVRAM-based multiversion databases indexing schemes.
3) According to the experimental results, our proposed Duery not only reduces the amount of write traffic to the NVRAM storage by 11% but increases the key- and version-range-query efficiency by 44%.

The rest of this article is organized as follows. In Section II, we will give the introduction to the technical background of this work. Section III details our problem definition. Section IV presents the Duery indexing scheme in detail. Section VI shows the evaluation of our design. Section VII draws the conclusion.

II. Technical Background

A. Multiversion B$^+$-Tree

With the explosive growth of data, data management becomes a critical issue in recent years. For efficient data management, many indexing schemes (or structures) (e.g., B$^+$-tree, AVL-tree, and hash table) have been presented and integrated into modern database systems. Among these indexing schemes, the B$^+$-tree is the most widely applied to various database systems and applications. For instance, B$^+$-tree has been employed to govern multiversion data [1], [2], [16] in different data management systems (e.g., cyber-physical systems and multiversion control systems). In other words, the multiversion data are usually generated by user modifications and periodic information collection process. Such data will be stored back to multiversion (database) systems.[2] Because multiversion systems will utilize the same index key to maintain the data with different versions, it is not easy to efficiently query different version data in multiversion systems. Fortunately, Becker *et al.* [2] presented the variant B$^+$-tree, namely multiversion B$^+$-tree (MVBT), for multiversion database systems.

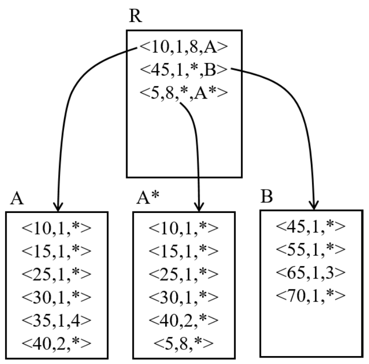

Fig. 1. MVBT structure. The entries are denoted as a form that ⟨idx, in_version, del_version, ptr⟩. For example, the data with index 35 are inserted in version 1 and deleted in version 4.

As the MVBT has been developed based on the typical B$^+$-tree structure, the MVBT inherits the features of B$^+$-tree structure. For example, the MVBT and the typical B$^+$-tree are the structure of a directed acyclic graph. Similar to the B$^+$-tree structure, the MVBT includes multiple tree nodes, and each node contains multiple entries, each of which maintains the data index and version information. Nevertheless, the MVBT structure contains different designs from the typical B$^+$-tree structure even though the design of MVBT is from the B$^+$-tree structure. Specifically, unlike the node structure of B$^+$-tree, an MVBT node includes the extra information of the data version for querying different version data. Fig. 1 shows the structure of MVBT.

As shown in Fig. 1, an MVBT node is composed of multiple version entries, each of which consists of *data index*, *insert version*, *delete version*, and *pointer*. For simplicity's sake, the information of an version entry could be denoted as ⟨*idx, in_version, del_version, ptr*⟩. Specifically, the *idx* information is the data index information and the *ptr* records a storage address where stores either another MVBT node or user data. Moreover, the *in_version* and the *del_version* are insert version and delete version, respectively. For instance, in Fig. 1, the data with index 35 are inserted in version 1 and deleted in version 4. Please note that the version number is incremental increase in a multiversion database system. However, if the data have not been deleted, the version entry will record an asterisk sign (i.e., *) in the information of *del_version*. Please note that an entry with the asterisk sign in its *del_version* filed is called

[2] A database system records multiple version data by keeping the modifications of data in a log file or creating the snapshot of current version data.

current entry. In Fig. 1, the data entry with index 10, for example, is not deleted, and thus, the asterisk sign is recorded in the *del_version* within the version entries of the data with index 10, so this data entry is also known as a current entry.

To keep the performance efficiency of B^+-tree structure, the MVBT sets an overflow condition and underflow condition. The overflow condition (also known as *strong version overflow*) will be triggered, while the number of entries in an MVBT node reaches the maximum capacity of an MVBT node. In other words, if an MVBT node does not have enough space to accommodate new version data entry, the overflow condition will be satisfied. Then, the MVBT structure splits the overflow node into multiple MVBT nodes by *version split*[3] and *key split*[4] operations. On the other hand, if the number of current entries in an MVBT node is smaller than the predefined threshold ($\min_{WT}$), the underflow condition will be satisfied, and the current entries within the underflow MVBT node will be merged to its sibling node. In the original MVBT work [2], $\min_{WT}$ equals β/α, where β is the capacity of an MVBT node and α is the nonzero positive value. Because the underflow situation will be resulted from version-split operations and entry deletions, the MVBT categorized the underflow condition into the *strong version underflow* (resulted from *version-split operations*) and the *weak version underflow* (resulted from entry deletions). No matter strong or weak version underflow, the MVBT will merge the copied current entries to another sibling node. By the MVBT node structure design and the overflow/underflow conditions definition, multiversion data can be efficiently maintained by the MVBT structure. Although the MVBT can efficiently manage multiversion data in multiversion database systems, it should be revised for emerging NVRAM storage devices. This is because the original MVBT does not include NVRAM's features (e.g., asymmetric read–write costs) in its design considerations.

B. Features of Nonvolatile Memory

With the consideration of low-power computer system design, NVRAM has been regarded as next-generation memory technology to replace dynamic random access memory (DRAM)-based main memory in future computer systems. As listed in Table I, one of the famous NVRAM technologies, namely phase-change memory (PCM), has comparable read performance and energy consumption with DRAM technology; however, NVRAM only needs 1% of the idle power required by DRAM. In other words, an NVRAM-based computer system barely needs power, while it is in an idle state. Therefore, with the considerations of zero-static power consumption and byte addressability, NVRAM-based storage devices have great potential to replace DRAM as the storage media in green system architectures.

Due to NVRAM's nice features, many storage vendors and research institutes are dedicated to NVRAM-based computer system development. For example, Intel and Micron jointly released the memory/storage product, namely 3-D XPoint [6], for an NVRAM-based computer system in the past few years. Because of the hardware support, researchers start to investigate different NVRAM-based system architectures. For example, because of the byte addressability and nonvolatility of NVRAM, some previous works [10], [26], [29] considered the structure that using NVRAM to be the only storage media in the system. On the other hand, there are also some studies [27], [28] adopted NVRAM to be a part of a hybrid main memory. Note that, no matter what kind of system architectures, once a system adopting NVRAM as its storage media can be referred to NVRAM-based storage system. Despite NVRAM technology development, NVRAM's natural shortcomings (e.g., high write energy and long write latency) raise the difficulty in the NVRAM-based system design. To accelerate NVRAM technology developments, many researchers focus on reducing the amount of write traffic to NVRAM for alleviating the impact of NVRAM's shortcomings on system performance. For instance, on reducing the amount of write traffic to NVRAM-based storage, many researchers proposed the NVRAM-friendly management systems by redesigning the journaling mechanisms [7]–[9], developing the one memory file system[5] [10], [14], and integrating the data compression module [11], [19], [20]. Although the NVRAM-friendly storage systems can effectively reduce the amount of write traffic to an NVRAM space, it still suffers from heavy write traffic if the NVRAM storage space is directly managed by database systems [12], [21]–[23] and key-value indexing schemes [13], [24], [25]. In other words, while a multiversion indexing scheme is not optimized for NVRAM-based storage, the system performance will be a critical issue. The reason is that the multiversion indexing scheme will write the same index (or version) entry multiple times to the NVRAM-based storage space during version-split operations. As a result, the multiversion database system's indexing scheme should be rethought for minimizing the amount of write traffic to NVRAM-based devices for boosting NVRAM-based storage system performance.

TABLE I
COMPARISON OF VARIOUS MEMORY TECHNOLOGIES [17], [18], [30], [31]

Properties	DRAM	PCM
Read latency	15 ns	50-70 ns
Write latency	15 ns	150-220 ns
Read energy	12.48 J/TB	16 J/TB
Write energy	3.12 J/TB	153.6 J/TB
Static power	$\sim$100 mW/GB	$\sim$1 mW/GB
Endurance	$\gg 10^{16}$	10^8-10^{10}
Cell size (F^2)	6-8	4-5

C. Related Works

Although MVBT achieved asymptotically optimal performance in time and space while being applied to block-based storage devices (e.g., hard-disk drives) [2], it still has some

[3]The version split will copy current entries from the overflow node to a new node space.

[4]The key split will evenly separate current entries within the overflow block into two node spaces and it always be performed after merge and version-split operations.

[5]The one memory file system jointly manages main memory and storage spaces.

problem while being conducted to NVRAM-based storage devices. Specifically, Kuan *et al.* [4] revealed that the MVBT structure has low space utilization on byte-addressable storage devices. In addition to the poor space utilization, the performance of version-range query[6] is the drawback of the MVBT structure. As a result, for rethinking the multiversion indexing scheme over an NVRAM-based storage device, many prior excellent works have been proposed in increasing the space utilization of MVBT [4], [5], improving the efficiency of serving version-range queries [3], and reducing the overhead for managing the MVBT index in processing update operations on NVRAM [1], that is to say, many researchers considered the characteristics of NVRAM memory technology in rethinking the MVBT structure over an NVRAM-based computer system. However, while the prior research works revised the MVBT structure for making it more friendly for the NVRAM-based system, these research works also ignored the intrinsic benefits of the MVBT design. For instance, Kuan *et al.* [4] improved MVBT's version-range-query operations by utilizing the version page data structure proposed by themselves. Nevertheless, the revised multiversion indexing scheme [4] made the performance of the key-range-query operations[7] worse in a large-scale multiversion system. The reason is that their proposed indexing scheme also abandoned the virtues of the MVBT structure. To the best of our knowledge, none or a few kinds of literature have proposed the write-reduction multiversion indexing scheme with efficient dual-range-query operations (i.e., version- and key-range-query operations) over NVRAM-based storage devices.

III. Problem of Dual-Query Over NVRAM

This section discovers the technical problems of developing a write-reduction multiversion indexing scheme with efficient dual-range queries over the NVRAM-based storage system. As Becker *et al.* [2] followed the spirit of B^+-tree to design the MVBT indexing scheme, the MVBT inherits the characteristics of B^+-tree indexing scheme. In the MVBT design, version data are managed by version entries within an MVBT node. For making the query more efficient, the MVBT indexing scheme contains a fixed number of version entries within an MVBT node. As the space of an MVBT node is static, the MVBT design introduces two split operations (i.e., version split and key split) to create new nodes for maintaining the incoming version entry, while the MVBT node does not have enough (free entry) space to store a new version entry. However, a version-split operation results in duplicate entries in the MVBT indexing scheme because current entries are copied from the to-be-split (or original) MVBT node to another MVBT node, while a version-split operation is performed. In other words, while we apply the MVBT indexing scheme to an NVRAM-based storage system, the performance of the NVRAM-based system will be degraded because the MVBT writes the same entry information multiple times to the NVRAM-based device during version-split operations.

[6]A version-range query is to find all data within the given range of versions.

[7]A key-range-query operation finds all versions data within the given range of keys.

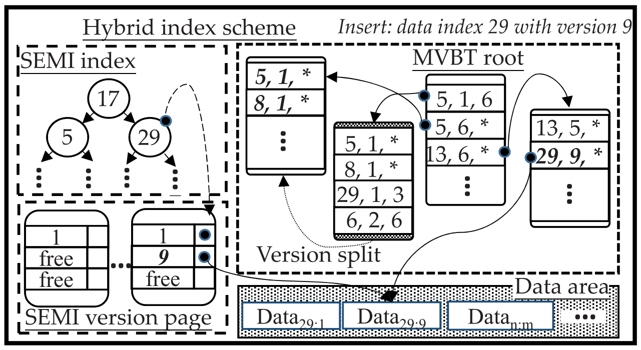

Fig. 2. Hybrid indexing scheme.

In addition, the MVBT indexing scheme does not aggregate the version entries belonging to the same data index together; hence, the MVBT structure has poor version-range-query performance.

To tackle the problem of the MVBT indexing scheme over NVRAM-based storage, Kuan *et al.* [5] has presented an SEMI scheme. In the SEMI design, the version page concept has been proposed to put all version entries belonging to the same data index together. By applying the version page to the multiversion indexing scheme, the SEMI enhances a version-range query's performance. Furthermore, the SEMI increases space efficiency by breaking the nature of the MVBT. Although the SEMI has better space utilization and version-range-query performance, its shortcoming (i.e., poor key-range-query performance) becomes a significant obstacle in applying to large-scale multiversion database systems. Because the SEMI maintained all data indexes in a binary tree structure, it is only appropriate for (small) embedded multiversion systems. To sum up, developing the write-reduction multiversion indexing scheme with efficient dual-range (i.e., key- and version-range) query is still an open issue over NVRAM-based storage systems.

The most intuitive solution for the multiversion indexing scheme with efficient dual-range queries is a hybrid index structure that combines the MVBT and the SEMI designs. As shown in Fig. 2, the hybrid indexing scheme can achieve better key-range-query and version-range-query performance by querying in the MVBT side and the SEMI data structure, respectively. Though the naive solution (i.e., hybrid indexing scheme) put the MVBT and the SEMI strengths together, it also inherits the MVBT scheme's drawback (i.e., the unnecessary amount of write traffic). Specifically, as shown in Fig. 2, the hybrid indexing scheme also produces duplicate version entries, while the version-split operation is performed. According to these observations, the goal of this study is to propose *a write-reduction multiversion indexing scheme with efficient dual-range-query operations over NVRAM-based storage.* More specifically, the technical problems of this study lie in: 1) enabling a multiversion indexing scheme with an efficient dual-range query for large-scale multiversion database systems; 2) avoiding duplicate entries in the multiversion index structure; and 3) minimizing the amount of write traffic issued by the multiversion indexing scheme to NVRAM-based storage.

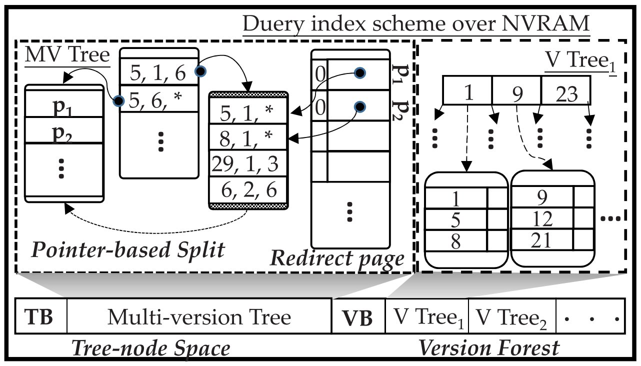

Fig. 3. Duery indexing scheme. An entry within the MV tree node contains three types of information: index, inserted, and deleted versions. In addition, each V tree records all version numbers of the same index in a B^+ tree and it is pointed by every entry with that index in the MV tree.

IV. Dual-Query Index Scheme Design

A. Overview

This study rethinks the current multiversion index structure for developing a write-reduction multiversion indexing scheme with efficient dual-range queries. According to our observation, due to NVRAM's drawbacks (e.g., long write latency and high write energy), our proposed NVRAM-friendly multiversion indexing scheme should avoid unnecessary write traffic to NVRAM-based storage. In addition, our proposed indexing scheme should perform efficient version- and key-range queries on large-scale multiversion database systems. With these considerations, this study proposes an efficient **Du**al-range qu**ery** indexing scheme, namely *Duery*, for the large-scale multiversion database systems over NVRAM-based storage. As shown in Fig. 3, the Duery scheme separates the index area on the NVRAM-based storage system into the *tree-node space* and the *version forest*. To support an efficient key-range query, the multiversion tree within the tree-node space preserves the partial nature of the MVBT. On the other hand, the version forest is composed of multiple version trees, each of which aggregates all version information belonging to the same index, so as to boost the version-range-query efficiency.

As shown in Fig. 3, Duery includes the multiversion tree and the version forest to boost the performance of key- and version-range queries simultaneously. The multiversion tree follows the spirit of the MVBT to provide an efficient key-range query. Nevertheless, unlike the MVBT, our proposed multiversion tree considers write reduction during version-split operations. With the consideration of write reduction, Duery brings in a pointer-based split mechanism without breaking the multiversion tree's nature. Explicitly, the pointer-based split mechanism minimizes the amount of write traffic issued by the MVBT design by avoiding copying full entry information during split operations. Instead of copying current entries, the pointer-based split only creates version pointers to address the position of to-be-split entry during split operations (refer to Section IV-C for more details).

SEMI [4], [5] takes the advantage of building the version entry to provide the efficient version-range-query operations; therefore, in addition to the multiversion tree, Duery also employs a version tree structure to collect all version entries with the same index together for enhancing the performance of the version-range query. However, SEMI [4], [5] used a binary tree as the indexing structure, so the height of the indexing tree would be very high. Once the number of keys is extremely large, it would cause a huge overhead on the exact-match-query, key-range-query, and dual-range-query operations. To apply the Duery to the large-scale multiversion database system, Duery developed the new version tree structure (i.e., *V Tree*) based on the B^+ tree. In the V Tree, the index in the node of the version tree represents the version number. Because a data index has its version tree in our proposed scheme, the version forest space is reserved for storing all version trees. As a result, the efficient key- and version-range queries can be supported by the multiversion tree and the version forest, respectively (refer to Section IV-B in detail).

In addition, while users remove old version data from the multiversion database, Duery needs to technically support a purge operation to release the space of dead version entry.[8] Since the proposed pointer-based split mechanism within Duery has changed the physical multiversion index layout over the NVRAM-based storage, Duery should develop new purge management for the pointer-based split mechanism. Specifically, the pointer-based split mechanism will not copy the information of current entries during split operations because the purpose of the pointer-based split mechanism is to minimize the amount of write traffic to NVRAM-based storage. Many version pointers might refer to a (dead) version entry because of the pointer-based split mechanism. Therefore, if a dead version entry is purged, all pointers pointing to the dead entry will be null. For avoiding pointer loss after a purging operation, the entry-based redirect page is proposed in Duery to create a translation page for keeping the purged version entry that is still referred by other version pointers (please refer to Section IV-D for more details). In addition, in order to manage the space of the multiversion tree and the version forest efficiently, we utilize the tree block bitmap (TB) and the version block bitmap (VB) to record the (node) block state. If Duery allocates a fixed-size block within the multiversion tree area (or the version forest area), the corresponding bit within the TB (or VB) will be set to one. Otherwise, the bit will be set to zero, while purge operations reclaim the allocated block.

To sum up, Duery is a write-reduction multiversion indexing scheme with efficient key- and version-range queries by consolidating the multiversion tree and the version forest. With the consideration of NVRAM's drawbacks, Duery employs the pointer-based split mechanism for reducing the amount of write traffic to NVRAM-based storage and the purge management for efficiently managing the deleted version information on the NVRAM-based storage devices. Moreover, the purge management within Duery conquers the problem of entry information loss (resulted from the pointer-based split mechanism). Briefly, Duery will be an NVRAM-friendly multiversion indexing scheme with efficient dual-range-query operations.

[8]A version entry does not contain asterisk sign in its *del_version* information.

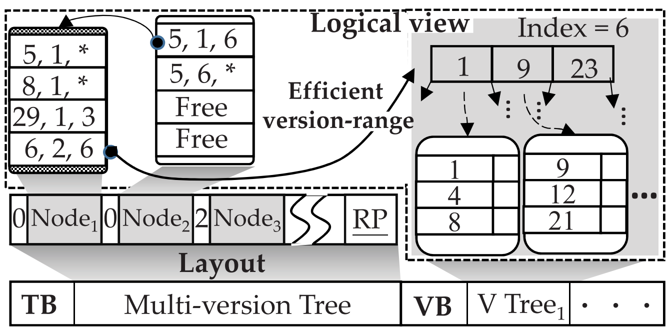

Fig. 4. Efficient dual-range query. In the figure, the V tree shown in the right is the V tree of the data with index 6 and pointed by all entries with index 6 in the multiversion tree. When doing the version query of index 6, Duery would first access the V tree by any entry of MV tree with index 6 and then return the required range of version to answer the query requirement.

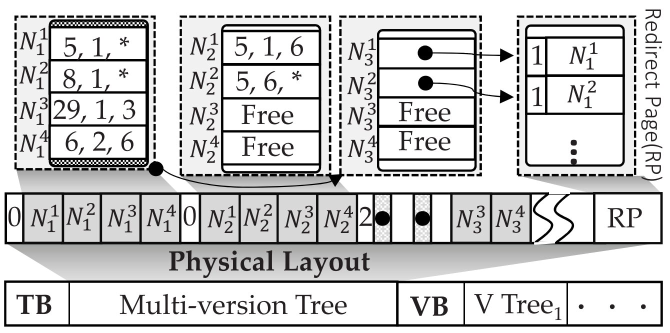

Fig. 5. Pointer-based split mechanism. In the figure, when Node 1 is full, Duery first creates 2 (the number of life entries in Node 1) entries in the redirect page. Second, Duery creates a new node (Node 3) in MV tree and points to the redirect page.

B. Efficient Dual-Range Query

For providing efficient dual-range queries, Duery proposes the version forest and combines it with the multiversion tree. Specifically, by inheriting the good nature of the MVBT, our multiversion tree can perform an efficient key-range query. More specifically, while users want to request a key-range query, Duery will search the multiversion tree by the given key range. Note that Duery has excellent performance on key-range-query operations because it follows the design of the key split operation within the MVBT. In other words, Duery can offer the efficient (key) range query provided by the MVBT as well. However, the MVBT cannot support efficient version-range-query operations because the dead entries within the MVBT structure will be scattered among different multiversion tree nodes by version-split operations, that is to say, the MVBT cannot only query a few tree nodes to obtain the result of the version-range query. In the worst case, the MVBT might traverse the whole multiversion index tree for a version-range query. To tackle this issue, Duery unites the version forest design, which is a version-oriented management scheme, as presented in Fig. 4. Note that an entry within the multiversion tree node contains three types of information: index, inserted, and deleted versions.

As shown in Fig. 4, the version forest is composed of multiple version trees (called *V Trees*), each of which contains a version B^+ tree and multiple version blocks. In Fig. 4, all version index belonging to the same index (e.g., index 6) will be maintained in the same *V Trees* structure. Therefore, the efficient version-range query can be done by searching the requested version range on the version B^+ tree and obtaining the requested version data via the version blocks. More specifically, in Fig. 4, if users want to query all version data belonging to index 6, Duery will perform a key query on the multiversion tree and then get one version entry belonging to index 6. Because all version entries within the multiversion tree structure contain a pointer pointing to its corresponding *V Trees*, Duery could visit the *V Trees* belonging to index six by getting the version entry belonging to index six within the multiversion tree structure. While Duery gets the V tree address belonging to index 6, Duery can return all version data belonging to index six by accessing all version blocks within the V Tree. As a result, Duery efficiently accomplishes the dual-range query by exploiting the proposed version forest's benefits with the multiversion tree structure.

C. Pointer-Based Split Mechanism

As mentioned in Section II-A, the MVBT indexing scheme has an excellent key-range-query performance. To inherit the fantastic performance from the MVBT design, Duery preserves the good natures of the MVBT. Unfortunately, the original MVBT design has some drawbacks (i.e., the unnecessary amount of write traffic to NVRAM-based storage and poor version-range-query performance) to obstruct the development of its application on NVRAM-based storage devices. For reducing the amount of write traffic to the NVRAM-based storage performance, Duery employs the pointer-based split mechanism (also known as pointer-based version split). It only creates a version point rather than copies complete current entry information during split operations. To detail the pointer-based split mechanism, Fig. 5 shows the NVRAM-based storage space managed by Duery during a version-split operation. Please note that a key split operation will not produce duplicate entry information, so the pointer-based split mechanism only targets the version-split operation.

For decreasing the amount of write traffic to the NVRAM-based storage device, the pointer-based split mechanism creates an entry in the redirect page, and the entry records the addresses of current entries within the to-be-split multitree node. As a result, Duery can only create a pointer entry in a new node for avoiding copying full version information from the to-be-split multitree node. As shown in Fig. 5, while Node 1 is full, a pointer-based version split is performed by Duery. The pointer-based version split creates two entries (e.g., N_1^1 and N_1^2) on the redirect page. These two entries record the addresses of nodes N_1^1 and N_1^2 in the storage space. As the locations of the nodes N_1^1 and N_1^2 are kept in the redirect page, the pointer-based version split does not need to copy the full version information of the current entries (e.g., N_1^1 and N_1^2) from the to-be-split node (e.g., Node 1) to the new node (e.g., Node 3). Instead of copying the version information, the pointer-based version split creates two-pointer entries in the new node (e.g., Node 3) to address the location of the redirect entries (e.g., N_1^1 and N_1^2) in the redirect page. Although the pointer-based split mechanism can effectively lower the amount of write traffic to the storage, it still has one problem identifying a pointer entry and a version entry. As shown in Fig. 5, because the size of a multiversion tree node is fixed, the prefix bits (e.g., first 8 bits) will represent the number of pointer entries in the multiversion node. For instance, in Fig. 5, the prefix bits of Node 3 store decimal 2, which means that the Node 3 contains version pointers in the first two entries. In other words, the version information will be stored starting from the third entry within the Node 3. Due to the prefix, Duery can identify a pointer entry and a version entry. Briefly, the pointer-based split mechanism can subdue the amount of write traffic to NVRAM-based storage during version-split operations. In addition, by collaborating with the prefix design and the redirect page, a variety of query operations can be performed on the multiversion tree as well.

D. Purge Management

Generally, version data will be written to and removed from a multiversion database system. Similarly, the version information will be inserted into and deleted from a multiversion indexing scheme. For efficiently handling the version information eliminated by a purge operation, Duery recruits innovative purge management that can avoid version information loss resulted from the pointer-based split mechanism. In our purge management, we need to reserve the space, whose size is enough for storing whole version entry, in each entry of the redirect page. Specifically, while the pointer-based split mechanism allocates space for recording the address of a current version entry within the to-be-split multitree node, the allocated space's size is enough for keeping the whole current version entry. Hence, while the to-be-split multitree node is purged, our purge management only needs to check whether other version pointers currently reference the purged version information in other multiversion tree nodes. Duery only needs to check the prefix value of the entry within the redirect page for confirming whether or not the purged version information is currently referenced. While the prefix value is not zero, our purge management will copy the version information from the to-be-purged tree node to the reserved entry within the redirect page because the purged version information is currently referenced. Otherwise, our purge management can directly reclaim the space of the purged tree node. After the purge operation, Duery will set the corresponding bit in the TB to 0. Fig. 6 shows the example of purge management.

As presented in Fig. 6, in this case, we purge the Node 1 in the multiversion tree structure. Before purging the Node 1, our purge management will check whether the version entries within the Node 1 have been referenced. In this example, by referencing to Fig. 5, two version entries (i.e., N_1^1 and N_1^2) has been referenced by Node 3. Because Node 1 keeps a pointer to address the location of Node 3 (which is split from Node 1 by a version-split operation), we can utilize the pointer to check whether the version-split node (i.e., Node 3) has any pointer entry that refers to the version entry in the original

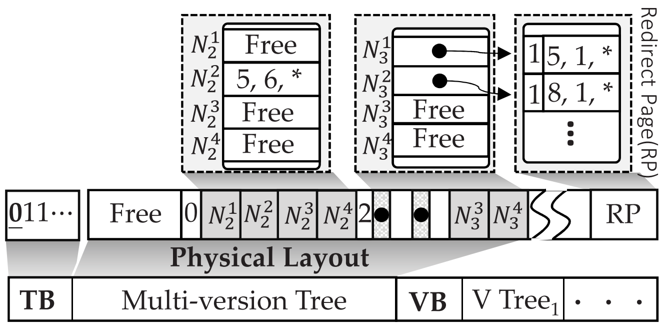

Fig. 6. Purge management. When a node is needed to be purged, Duery should check whether the version entries within the to-be-pruned Node have been referenced. For example, in the figure, before purging Node 1, Duery has to write the entries that are referenced by other nodes in Node 1 (⟨5,1,*⟩, ⟨8,1,*⟩) in the redirect page.

TABLE II
SYMBOLS FOR ANALYSIS

Symbol	Definition
d	The minimum number of entries in a node of the MV tree
N_e	The number of entries in a block
N_i	The number of entries in version i
S_b	The size of a block (byte)
S_p	The size of pointer
S_e	The size of an entry
r	The number of data items returns by a query
v	The minimum number of entries in a node of a V tree
N_{vk}	The number of Version of key k
R_v	The number of versions in version-range-query
S_v	The size of a key of V tree in byte

node (i.e., Node 1). If the version-split node has referred to some version entries in the original (or to-be-purged) node, the purge management can find the corresponding redirect entry and check its prefix value. In Fig. 5, the prefix value of the redirect entries N_1^1 and N_1^2 is one; therefore, our purge management will copy the version information from Node 1 to the corresponding redirect entries and then reclaim the storage space of Node 1. Due to purge management, Duery can protect the multiversion indexing scheme from version information loss, while a purge operation is performed.

V. Theoretical Analysis of Duery

The purpose of this section is to theoretically analyze the time efficiency of Duery under the different query-operation types, including exact-match-query, key-range-query, and version-range-query operations. The symbols for analyzing are shown in Table II.

A. Time Efficiency of Exact-Match Query

For providing an efficient exact-match query, Duery used the partial MVBT structure to maintain the entries of the key with lifetime information in a B-tree structure. Furthermore, for maximizing the benefit of NVRAM architecture, Duery exploits the byte addressability of NVRAM. In other words, instead of reading data in block granularity, Duery can access data by the granularity of byte. However, in order to reduce the write traffic of maintaining the multiversion index, the pointer-based split method within Duery avoids copying many duplicate data during the version-split process. However, it would incur the additional reading cost because of the pointer reading. To sum up, when an exact-match query is conducted on Duery, there are two operations that may contribute to the reading cost and influence time efficiency: 1) traversing the MV tree to find the entry and 2) looking up the redirect page. In the worse case, the number of accessed blocks is $\lceil \log_d N_i \rceil$ because the MV tree is traversed during the exact-match query. Therefore, accessing these blocks should read $\lceil \log_d N_i \rceil \times S_b + \lceil \log_d N_i \rceil \times S_p$ bytes in the worse case. In the worst case analysis, time to look up the redirect page would be $N_e \times \lceil \log_d N_i \rceil$ in the case of considering all entries redirected by looking up the redirected page. Therefore, the worst reading overhead of looking up the redirected page is $N_e \times \lceil \log_d N_i \rceil \times (S_e + S_p)$. Summing up these two costs would be the total reading time of the exact-match query. Note that, in most cases, Duery would not read all entries in one block since NVRAM provides byte addressability. In other words, when the entry satisfies the query, Duery would return the result immediately.

B. Time Efficiency of Key-Range Query

Duery can reach high efficiency of key-range-query operation because the MV tree maintains partial nature of MVBT structure and inherit its merit. More specifically, when a key-range-query operation is conducted, the cost of accessing MV tree blocks in the worst case would be $k(\lceil \log_d N_i \rceil + r/d) \times S_b$ bytes, where k is a constant. On the other hand, as aforementioned, Duery exploits a pointer-based split mechanism to avoid huge write traffic, so there would be some additional reading cost. The additional cost causing by the pointer reading would be at most $N_e \times S_p$ when all the entries in accessing blocks need to redirect. Therefore, the total cost of doing key-range query would be $k(\lceil \log_d N_i \rceil + r/d) \times S_b + N_e \times S_p$.

C. Time Efficiency of Version-Range Query

For accelerating the performance of version-range query, Duery adopts the V tree structure to index the version entries of each entry in MV tree. Note that all entries in the MV tree have a pointer to its V tree. Therefore, when a version-range query is required, Duery can find the V tree location through the pointer in any entry of MV tree with the same key. The largest number of accessed blocks in MV tree is also $\lceil \log_d N_i \rceil$, where i can be any version. Therefore, the reading cost of finding a V tree entry is $\lceil \log_d N_i \rceil \times S_b + S_p$ bytes. Since a V tree is a B$^+$ tree structure, Duery can find the smallest query version by accessing at most $\lceil \log_v N_{vk} \rceil$ nodes of V tree. Then, it reads $R_v \times S_v$ version entries in V tree to answer the version query. To sum up, the total cost of version-range query in Duery is $\lceil \log_d N_i \rceil \times S_b + S_p + \lceil \log_v N_{vk} \rceil \times (S_v + S_p) + R_v \times S_v$ in the worst case.

VI. Performance Evaluation

A. Experiment Setup

The purpose of this section is to evaluate the capabilities of Duery indexing scheme over the NVRAM-based storage system, in terms of the amount of write traffic to NVRAM storage, write energy consumption, write latency, and query performance. All the experimental results would be compared with MVBT and SEMI. In addition, for emphasizing the benefits of Duery indexing scheme, we compare our Duery indexing scheme with a naive solution (i.e., the combination of MVBT and SEMI). This is because only these two solutions (i.e., Duery index and the naive solution) simultaneously provide efficient key- and version-range queries. Furthermore, we were not directly combined two solutions instead we implemented the naive solution with the consideration of the byte-addressable property of NVRAM to improve the performance of nature MVBT. All experimental result is collected from an in-house simulation program that is modified from the previous works [4], [5], and all compared solutions will be performed on this simulator. The data set, which includes the information of data index and the corresponding I/O operation (e.g., insertion, update, deletion, and query), is collected from the previous works [4], [5]. More specifically, our simulation program is implemented in C^{++} programming language based on the DRAM-based simulator used in previous works [4], [5]. In this simulator, we realized the proposed design and the comparison methods. We collected the I/O operations by counting the writing and reading in the byte unit while running the experimental traces on our simulator and then estimated the system performance and energy consumption by referencing PCM specifications [30], [31]. Note that, multiversion indexing scheme needs to be preserved persistently. Due to the nonvolatility of NVRAM, NVRAM is suitable to apply to keeping the multiversion indexing scheme. Moreover, PCM is one of the most mature technologies of NVRAM; therefore, it is selected to be the case study in our experiments. Briefly, our experiments will generate different workloads by a workload synthesizer, which can produce multiple different workloads with the different number of keys and versions. After that, the workload is fed to the DRAM-based simulator to collect reading/writing bytes issued by the different multiversion indexing schemes. On the other hand, Intel Optane is a commercial PCM-like device, and therefore, we also referred to its published data to show the results of the proposed method. Note that, it is hard to know the read/write latency and energy of Optane, because of its hybrid architecture, so we took its read/write maximum bandwidth [32] to be the reference parameter in the experiment. Notably, because Duery reduces a lot of write traffic and provides high efficient query operations by well exploiting the byte addressability of NVRAM, Duery can gain the profit from the most NVRAM technologies (e.g., STT-RAM, ReRAM, and PCM) with the features of byte addressability and asymmetric read/write performance. Note that we do not use a special-purpose data set because we cannot generate a variety of user access behaviors to evaluate the Duery indexing scheme's performance. A node contains 15 entries in the experimental settings, and the page size for SEMI is 4 kB. In addition, the number of I/O operations in our experimental data set is ranged between 750k and 1.5M. Also, to show the capacity of Duery, we varied the number of

TABLE III
EXPERIMENTAL CONFIGURATIONS

System environments	
Node Size	15 entries
Page Size	4KB
NVRAM specification [30], [31], [32]	
PCM Write latency	150 ns
PCM Write energy	153.6 J/TB
PCM Read latency	50 ns
PCM Read energy	16 J/TB
Optane Write bandwidth	2.3 GB/s

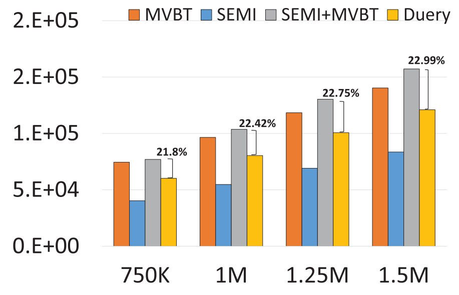

Fig. 7. Write traffic (bytes) to NVRAM.

unique keys between 1k and 500k. The details of experimental settings are summarized in Table III.

B. Experimental Results of Write-Efficiency Improvements

1) Amount of Write Traffic: For investigating the Duery indexing scheme's performance, we mainly compare Duery with the naive solution (i.e., the combination of the MVBT and the SEMI), MVBT and SEMI. Note that, only SEMI + MVBT can offer efficient either key- or version-range query. In other words, SEMI and MVBT cannot keep the same performance to respond to the user's version- and key-range queries. Therefore, the following discussion would mainly focus on the comparison of SEMI + MVBT and Duery. As the goal of this study is to develop a write-reduction multiversion indexing scheme with efficient dual-range (i.e., key- and version-range) queries, we should evaluate the amount of write traffic to NVRAM. To show the effect of Duery indexing scheme on write reduction, we evaluate the capability of Duery index by running the experimental data set. Fig. 7 shows the results of write reduction. In Fig. 7, the x-axis is the number of I/O operations issued to the compared multiversion indexing scheme, and the y-axis is the write traffic in byte issued by the Duery index to that generated by the baseline solution (i.e., the naive scheme). As shown in Fig. 7, the proposed Duery can reduce the amount of write traffic to NVRAM storage by about 21%, compared with the naive design. This is because Duery brings in the concept of the pointer-based split mechanism that avoids copying the whole version entry to a new node during version-split operations. According to this experimental result, Duery presents its ability in write reduction so as to prove that Duery is an NVRAM-friendly multiversion indexing scheme.

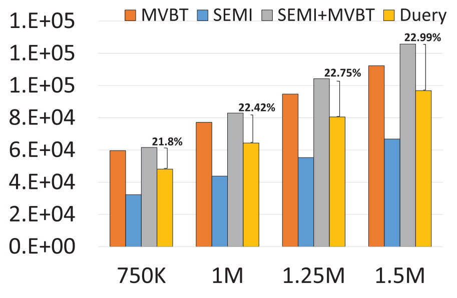

Fig. 8. Total write time (ms).

2) Total Write Time: Although the proposed Duery can effectively decrease the amount of write traffic to the NVRAM space, we also wonder its benefits of energy and performance efficiencies. Therefore, this experiment investigates the ability of Duery indexing scheme in shortening total write execution time. In this experiment, we also mainly compare our proposed solution with the naive design in the writing performance because only our proposed Duery and the naive solution can support efficient key- and version-range query operations. In other words, MVBT and SEMI cannot finish dual-range (i.e., key- and version-range) query in one operation. Similarly, we conducted the same experimental data sets on both Duery and the naive solution. Fig. 8 shows the result of total write time. Explicitly, the x-axis of Fig. 8 is the number of I/O counts in the experimental data sets, and the y-axis of Fig. 8 represents the amount of writing time. As shown in Fig. 8, when the number of I/O counts is increased, the total write time of our Duery and the naive solution is also raised. As expected, Duery still has better write performance than the naive solution because Duery can decrease the amount of write traffic to the NVRAM storage space. Furthermore, compared with the naive solution, the Duery indexing scheme can shorten the total write time by 20%–23%.

In addition to PCM, Intel Optane [32] is a well-known PMC-like product. Therefore, we would also like to use its published data to evaluate the proposed methods. However, Intel Optane adopts a hybrid architecture that contains a DRAM cache and PCM memory in a device, so it is hard to know the latency of it. Therefore, we take the maximum bandwidth as the parameter to simulate the performance of the proposed method while running on the Intel Optane. As shown in Fig. 9, the execution time of writing is more less than Fig. 8. It is because Intel Optane uses the DRAM cache to provide high performance. However, even though Intel Optane using the DRAM cache to achieve high performance according to the published analysis of Intel Optane [32], its overall bandwidth also suffers from the read/write asymmetric problem. More specifically, the maximum read bandwidth of Optane is 6.6 GB/s, whereas the maximum write bandwidth is 2.3 GB/s. Therefore, the advantage of the proposed Duery also exists when running it on Intel Optane.

3) Write Energy Consumption: Reducing the amount of write traffic to NVRAM storage not only increases storage performance but also decreases the write energy consumption. This is because an NVRAM-based system has high write energy and long write latency. Therefore, we wonder how

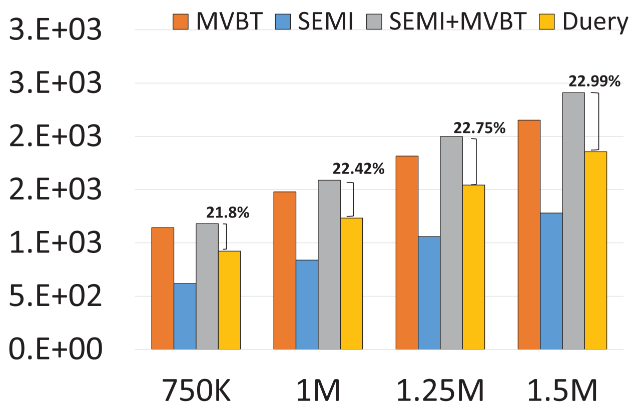

Fig. 9. Total write time on Intel Optane (ms).

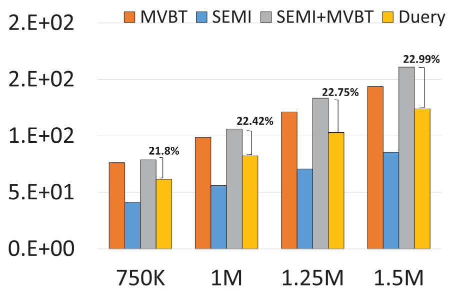

Fig. 10. Total write energy (mJ).

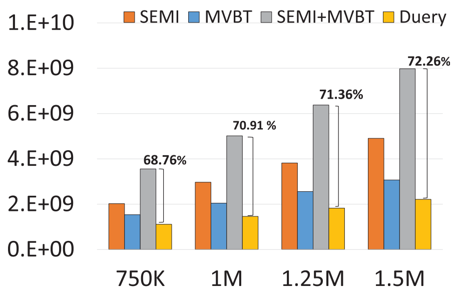

Fig. 11. Read traffic (bytes) to NVRAM.

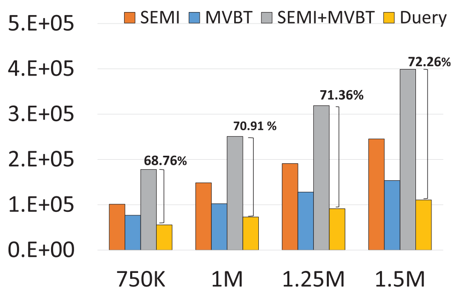

Fig. 12. Total read time (ms).

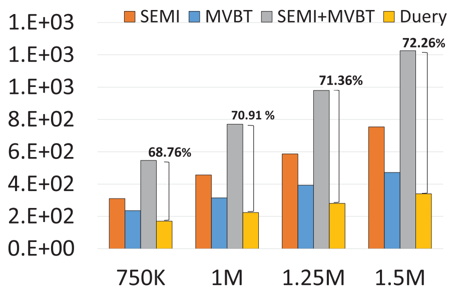

Fig. 13. Total read energy (mJ).

the proposed Duery effectively reduces the energy consumption of the multiversion database system. For estimating the write energy consumption, we referred to the specification of NVRAM storage listed in Table III. The result of write energy consumption is shown in Fig. 10. Similarly, the x-axis of Fig. 10 is the number of I/O operations, and the y-axis of Fig. 10 is the write energy consumption. The Duery indexing scheme results in less write energy consumption than the naive solution because Duery efficiently lowers the amount of write traffic to the NVRAM storage space. As shown in Fig. 10, Duery only needs 77%–80% energy consumption required by the naive solution.

C. Effect of Read Operations During Insertion

On the other hand, during the insertion process, the read operations also contribute to the execution time and energy consumption. Therefore, we measured the read bytes, which is induced by insertion operations. The result of read traffic, the total read time, and the total read energy consumption is shown in Figs. 11–13, respectively. First of all, as shown in Fig. 11, when compared to SEMI + MVBT, the proposed Duery can reduce the amount of read traffic during the insertion process by about 70%. In addition, when compared to SEMI and MVBT, the proposed Duery also induces less read traffic because the structure of Duery has a fewer redundant pointer. Note that, the y-axis of Fig. 11 is the read traffic in bytes. Similarly, the trends of total read time and total read energy consumption are the same as that of the amount of read traffic. Therefore, the total read time and energy consumption can also be reduced by approximately 70% when compared to SEMI + MVBT. The details of the total read time and read energy consumption are, respectively, summarized in Figs. 12 and 13. Note that, the y-axis of Fig. 12 is the total read time in ms, and the y-axis of Fig. 13 denotes the energy consumption of read in mJ.

D. Results of Query-Efficiency Improvements

To investigate the Duery indexing scheme's query performance, we perform exact-match, key-range, and version-range queries on the Duery indexing scheme and the naive solution. Note that, Duery aims to provide the efficient dual-range query on the NVRAM device, and therefore, all the range queries in the experiments are both key- and version-range queries. For example, the queries in key-range-query are a query containing a large range of key queries and a small range of version queries. Similarly, the version-range query would query a large range of versions with a small range of keys. To evaluate the query performance, we count the number of reading bytes per query, while exact-match, version-range, and key-range queries are performed on the compared solutions. More specifically, in the first step, we construct the multiversion

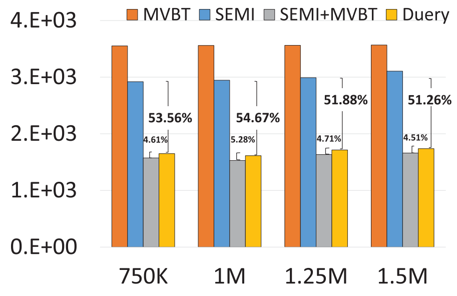

Fig. 14. Exact-match-query performance (bytes).

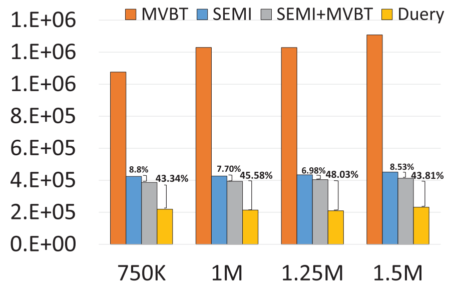

Fig. 15. Key-range-query performance (bytes).

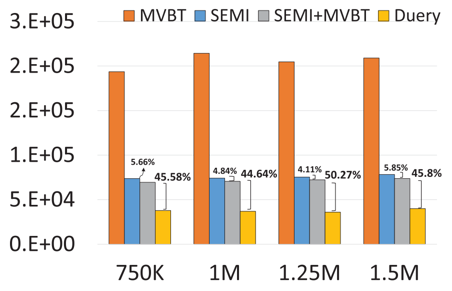

Fig. 16. Version-range-query performance (bytes).

indexing scheme by running the experimental data set on the compared solutions. We then perform a different number of query operations to the compared multiversion indexing scheme and count the number of reading bytes. In the exact-match results and version- and key-range queries, we will show the number of reading bytes on average.

1) Efficiency of Exact-Match Query: First of all, we investigate the performance of Duery indexing scheme on the exact-match query. In this query operation, users will give a specific key with a specific version, and the multiversion indexing scheme should return the corresponding version data by the given key and version. In this comparison, we evaluate the Duery index's capability, the naive solution (SEMI + MVBT), MVBT, and SEMI. Note that, MVBT used to be adopted on the block-based device; therefore, the I/O unit of MVBT would be a block. On the other hand, the MVBT in the naive solution is the modified version by taking the benefit of byte addressability of NVRAM. The experimental results are shown in Figs. 14–16, and all x-axes are the number of query operations, and all y-axes are the number of reading bytes per query on average. Note that, the y-axis of Figs. 10–12 is the number of reading bytes per query on average. In addition, the data set of this experiment contains a fixed number of unique keys (i.e., 1000) and repeatedly queries the indexing scheme to reach the number of I/O operations. Since each indexing scheme maintains the same amount of keys in its structure and is coped with the different amounts of query operations, the average value (i.e., the value of the y-axis) of the query related to "key" would be very similar and negligible dependence on the x-axis. As shown in Fig. 14, the Duery indexing scheme can reduce the number of reading bytes more than 50%, compared with the SEMI structure and reduce about 70% than the MVBT structure. The reason is that Duery exploits the benefits of NVRAM's byte addressability. However, the pointer-based split mechanism of Duery may cause additional read overhead when compared to SEMI + MVBT because we also implemented SEMI + MVBT with byte addressability. More specifically, when doing exact-match query, Duery would take about 5% more read overhead than SEMI + MVBT. Note that, in the beginning, Duery already reduces more than 20% write traffic to NVRAM when compared to SEMI + MVBT; therefore, the additional read overhead causing by the pointer-based split mechanism could be ignored.

2) Efficiency of Dual-Range Query: In addition to the exact-math query performance, the efficiency of the key- and version-range queries will be more important for evaluating Duery scheme's abilities as Duery features write reduction and efficient dual-range queries. Note that users will give a small range of version and a large range of keys while performing a key-range query. As presented in Fig. 15, the Duery indexing scheme still has the best performance among these solutions. This is because the pointer-based split mechanism reduces the amount of write traffic to NVRAM storage; in other words, our split mechanism simplifies the multiversion index structure. As a result, according to the experimental result, the Duery indexing scheme can effectively reduce the number of read bytes by 43%–48%, compared with the SEMI + MVBT structure.

3) Efficiency of Version-Range Query: In a version-range query, users will give a small range keys and a large range of versions. We perform version-range queries on Duery, the naive solution, MVBT, and the SEMI scheme. Fig. 15 shows the result of version-range-query performance. In this experiment, we still count the number of reading bytes per query as the query performance. As shown in Fig. 15, Duery has the best version-range-query performance because it reads fewer bytes per query. Compared with the naive scheme, Duery can improve version-range-query performance by up to 50%.

E. Advanced Discussion

In Section VI-D, we have discussed the query efficiency with a different amount of I/O operations. The purpose of this discussion is to investigate the influence of the number of unique keys on the indexing scheme performance for further examining the scalability of the proposed method. Therefore, we generate the operations with the different numbers of unique keys from 1k to 500k and also investigate the performance of three different query-operation types under the

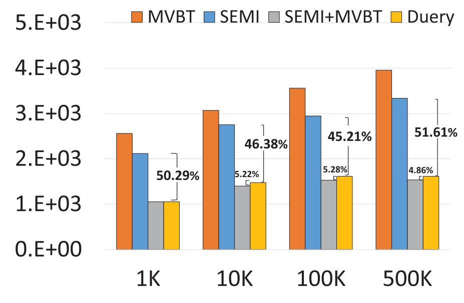

Fig. 17. Exact-match-query performance (bytes).

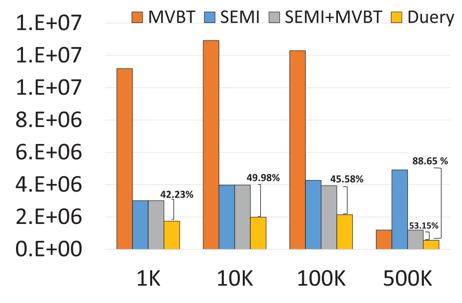

Fig. 18. Key-range-query performance (bytes).

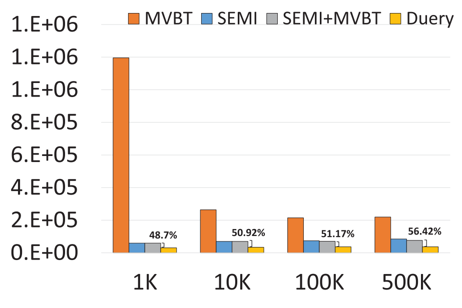

Fig. 19. Version-range-query performance (bytes).

experimental methods. The experimental results are shown in Figs. 17–19, and all x-axes are the number of unique keys in the data set, and all y-axes are the number of reading bytes per query on average. As shown in Fig. 17, Duery can outperform the SEMI by over 45% when executing exact-match query, and our solution only suffers from a little overhead (i.e., about 5%), compared with the native method. Moreover, Fig. 18 particularly presents the poor scalability of SEMI. As shown in Fig. 18, when the number of unique keys raised, SEMI would get worse performance. While the number of unique keys reaches a threshold, the SEMI performance is even worse than MVBT with the block-based I/O unit. In other words, SEMI cannot perform well when the number of unique keys increased. In addition, SEMI is proposed for achieving high performance of version-range query. However, when the number of unique keys increased, the performance of version-range query would also be degraded. In contrast, Duery can still perform well when the number of unique keys increased. To sum up, with the outstanding scalability, Duery can perform stably when being applied to various applications.

VII. Conclusion

Efficient dual-range (i.e., key- and version-range) queries gradually become important to a multiversion indexing scheme in cyber-physical systems due to the explosive growth of version data. However, current multiversion indexing schemes cannot simultaneously support efficient key- and version-range queries. Besides, as NVRAM storage has been regarded as the next-generation memory/storage technology in a computer system, we should include the asymmetric read/write cost of NVRAM in our multiversion indexing scheme design. Therefore, this work aims to develop a write-reduction multiversion indexing scheme with efficient dual-range queries, namely Duery. By developing and integrating the proposed multiversion tree structure and version forest, Duery can efficiently perform version- and key-range queries on the multiversion index system. In addition, Duery includes the pointer-based split mechanism that reduces the amount of write traffic to NVRAM storage by 20%, compared with a straightforward solution. According to the experimental results, Duery has the best performance on the exact-match, version-range, and key-range queries by comparing with other solutions (i.e., the MVBT, the SEMI, and the naive solution).

References

[1] J. Wang, K. Lam, Y. Chang, J. Hsieh, and P. Huang, “Block-based multiversion B$^+$-tree for flash-based embedded database systems,” *IEEE Trans. Comput.*, vol. 64, no. 4, pp. 925–940, Apr. 2015.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, “An asymptotically optimal multiversion B-tree,” *VLDB J. Int. J. Very Large Data Bases*, vol. 5, no. 4, pp. 264–275, Dec. 1996.

[3] C. J. Zhu, K.-Y. Lam, Y.-H. Chang, and J. K. Y. Ng, “Linked block-based multiversion B-tree index for PCM-based embedded databases,” *J. Syst. Archit.*, vol. 61, no. 9, pp. 383–397, Oct. 2015.

[4] Y.-H. Kuan, Y.-H. Chang, P.-C. Huang, and K.-Y. Lam, “Space-efficient multiversion index scheme for PCM-based embedded database systems,” in *Proc. 51st ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2014, pp. 1–6.

[5] Y.-H. Kuan, Y.-H. Chang, T.-Y. Chen, P.-C. Huang, and K.-Y. Lam, “Space-efficient index scheme for PCM-based multiversion databases in cyber-physical systems,” *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 1, pp. 1–26, Nov. 2016.

[6] *Intel and Micron Produce Breakthrough Memory Technology*, Intel Corporation, Mountain View, CA, USA, 2015.

[7] E. Lee, S. H. Yoo, and H. Bahn, “Design and implementation of a journaling file system for phase-change memory,” *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1349–1360, May 2015.

[8] T.-Y. Chen *et al.*, “WrJFS: A write-reduction journaling file system for byte-addressable NVRAM,” *IEEE Trans. Comput.*, vol. 67, no. 7, pp. 1023–1038, Jul. 2018.

[9] T.-Y. Chen *et al.*, “Enabling write-reduction strategy for journaling file systems over byte-addressable NVRAM,” in *Proc. 54th Annu. Design Automat. Conf.*, Austin, TX, USA, Jun. 2017, pp. 1–6.

[10] S.-H. Chen, T.-Y. Chen, Y.-H. Chang, H.-W. Wei, and W.-K. Shih, “UnistorFS: A union storage file system design for resource sharing between memory and storage on persistent RAM-based systems,” *ACM Trans. Storage*, vol. 14, no. 1, pp. 1–22, Apr. 2018.

[11] N. K. Edel, D. Tuteja, E. L. Miller, and S. A. Brandt, “MRAMFS: A compressing file system for non-volatile RAM,” in *Proc. IEEE Comput. Soc. 12th Annu. Int. Symp. Modeling, Anal., Simul. Comput. Telecommun. Syst., (MASCOTS)*, Oct. 2004, pp. 596–603.

[12] S. Gao, J. Xu, T. Harder, B. He, B. Choi, and H. Hu, “PCMLogging: Optimizing transaction logging and recovery performance with PCM,” *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 12, pp. 3332–3346, Dec. 2015.

[13] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, “Exploring storage class memory with key value stores,” in *Proc. 1st Workshop Interact. NVM/FLASH Operating Syst. Workloads (INFLOW)*, Farmington, PA, USA, Nov. 2013, pp. 1–8.

[14] S.-H. Chen, T.-Y. Chen, Y.-H. Chang, H.-W. Wei, and W.-K. Shih, "Enabling union page cache to boost file access performance of NVRAM-based storage device," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.

[15] Y. Tong, C. C. Cao, C. J. Zhang, Y. Li, and L. Chen, "CrowdCleaner: Data cleaning for multi-version data on the Web via crowdsourcing," in *Proc. IEEE 30th Int. Conf. Data Eng.*, Chicago, IL, USA, Mar. 2014, pp. 1182–1185.

[16] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen, "Transactions on the multiversion B$^+$-tree," in *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol. (EDBT)*, Saint Petersburg, Russia, Mar. 2009, pp. 1064–1075.

[17] J.-T. Yun, S.-K. Yoon, J.-G. Kim, B. Burgstaller, and S.-D. Kim, "Regression prefetcher with preprocessing for DRAM-PCM hybrid main memory," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 163–166, Jul. 2018.

[18] M.-C. Yang, Y.-H. Chang, and C.-W. Tsao, "Byte-addressable update scheme to minimize the energy consumption of PCM-based storage systems," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 3, pp. 1–20, Jul. 2016.

[19] J. Kim, C. Min, and Y. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 217–224, May 2014.

[20] M. Zarubin, P. Damme, T. Kissinger, D. Habich, W. Lehner, and T. Willhalm, "Integer compression in NVRAM-centric data stores: Comparative experimental analysis to DRAM," in *Proc. 15th Int. Workshop Data Manage. New Hardw. (DaMoN)*, Amsterdam, The Netherlands, 2019.

[21] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the NVRAM era," *Proc. VLDB Endowment*, vol. 7, no. 2, pp. 121–132, Oct. 2013.

[22] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware logging in transaction systems," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 389–400, Dec. 2014.

[23] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Atlanta, GA, USA, Mar. 2016, pp. 385–398.

[24] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A hybrid index key-value store for DRAM-NVM memory systems," in *Proc. USENIX Conf.*, Santa Clara, CA, USA, Jul. 2017, pp. 349–362.

[25] Y. Shi, Z. Shen, and Z. Shao, "SQLiteKV: An efficient LSM-tree-based SQLite-like database engine for mobile devices," in *Proc. 23rd Asia South Pacific Design Automat. Conf. (ASP-DAC)*, Jeju, South Korea, Jan. 2018, pp. 28–33.

[26] J.-Y. Jung and S. Cho, "Memorage: Emerging persistent RAM based malleable main memory and storage architecture," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. (ICS)*, Eugene, OR, USA, Jun. 2013, pp. 115–126.

[27] 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. (FAST)*, Santa Clara, CA, USA, 2016, pp. 323–338.

[28] C. J. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+ DRAM hybrid main memory," in *Proc. 12th Conf. Hot Topics Operating Syst. (HotOS)*, Monte Verità, Switzerland, May 2009, pp. 4–14.

[29] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "NVM Duet: Unified working memory and persistent store architecture," *ACM SIGARCH Comput. Archit. News*, vol. 49, no. 4, pp. 455–470, Apr. 2014.

[30] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, Jun. 2009, pp. 2–13.

[31] S. Eilert, M. Leinwander, and G. Crisenza, "Phase change memory: A new memory enables new memory usage models," in *Proc. Int. Memory Workshop (IMW)*, May 2009, pp. 1–2.

[32] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B$^+$-tree," in *Proc. 16th USENIX Conf. File Storage Technol. (FAST)*, 2018, pp. 187–200.