B^p-Tree: A Predictive B^+-Tree for Reducing Writes on Phase Change Memory

Weiwei Hu, Guoliang Li, *Member, IEEE,* Jiacai Ni, Dalie Sun, and Kian-Lee Tan, *Member, IEEE*

Abstract—Phase change memory (PCM) has been considered an attractive alternative to flash memory and DRAM. It has promising features, including non-volatile storage, byte addressability, fast read and write operations, and supports random accesses. However, there are challenges in designing algorithms for PCM-based memory systems, such as longer write latency and higher energy consumption compared to DRAM. In this paper, we propose a new *predictive* B^+-tree index, called the B^p-tree, which is tailored for database systems that make use of PCM. Our B^p-tree reduces data movements caused by tree node splits and merges that arise from insertions and deletions. This is achieved by pre-allocating space on PCM for near future data. To ensure the space are allocated where they are needed, we propose a novel predictive model to ascertain future data distribution based on the current data. In addition, as in [4], when keys are inserted into a leaf node, they are packed but need not be in sorted order. We have implemented the B^p-tree in PostgreSQL and evaluated it in an emulated environment. Our experimental results show that the B^p-tree significantly reduces the number of writes, therefore making it write and energy efficient and suitable for a PCM-like hardware environment.

Index Terms—Phase change memory (PCM), non-volatile storage, B^p-tree, predictive model

1 Introduction

The current established memory technologies suffer from various shortcomings: DRAMs are volatile and flash memories exhibit limited write endurance and low write speed. The emerging next-generation non-volatile memory (NVM) is a promising alternative to the traditional flash memory and DRAM as it offers a combination of some of the best features of both types of traditional memory technologies.

There are some widely pursued NVM technologies: magneto-resistive random access memory (MRAM), ferroelectric random access memories (FeRAM), resistive random access memory (RRAM), and phase change memory (PCM) [13] and in this paper, we will focus on PCM technology. Like DRAM, PCM is byte addressable and supports random accesses. However, PCM is non-volatile and offers superior density to DRAM and thus provides a much larger capacity within the same budget [19]. Compared to NAND flash, PCM offers better read and write latency, better endurance and lower energy consumption. With these features, PCM can be used as a storage between DRAM and NAND flash, and we can expect it to have a big impact on the memory hierarchy and adopt it as a desirable storage for databases [4], [19]. In this paper, we focus on designing indexing techniques in PCM-based hybrid main memory systems.

Existing indexing techniques cannot be directly used in PCM efficiently and there are several main challenges in designing new algorithms for PCM. First, though PCM is faster than NAND flash, it is still much slower than DRAM, especially the write function, which greatly affects system performance. Second, the PCM consumes more energy because of the phase change of the material. We will elaborate on this point in Section 2.1. Third, compared to DRAM, the lifetime of PCM is shorter, which may limit the use of PCM for commercial systems. However, as mentioned in [4], [19], some measures could be taken to reduce write traffic as a means to extend the overall lifetime. In general, the long access latency and high energy consumption are the major factors that affect the performance of PCM-based memory systems.

We aim to design a write-optimized indexing technique for PCM-based memory systems by reducing the number of writes. As known to all, when nodes are full in traditional B^+-tree indexing, nodes are split and half of the data on these nodes are moved to new nodes which leads to many extra writes. In other words, some data were written to the wrong place initially and we can avoid such writes if we know the future data access in advance. In [14], Nadembega *et al*. proposed the Destination Prediction Model (DPM) to predict the future movements based on historical movement pattern which inspired us to propose a predictive model to predict the future data distribution and reduce the number of writes caused by nodes splits. We can make

- *W. Hu and K.-L. Tan are with the School of Computing, National University of Singapore, Singapore 117417. E-mail: {huweiwei, tankl}@comp.nus.edu.sg.*
- *G. Li and J. Ni are with the Department of Computer Science, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing 100084, China. E-mail: {liguoliang@; njc10@mails.}tsinghua.edu.cn.*
- *D. Sun is with the Department of Computer Science, Harbin Institute of Technology, Harbin 150001, China. E-mail: sdl@hit.edu.cn.*

Manuscript received 11 Dec. 2012; revised 27 Nov. 2013; accepted 30 Nov. 2013. Date of publication 8 Jan. 2014; date of current version 29 Aug. 2014.
Recommended for acceptance by L. Chen.
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 10.1109/TKDE.2014.5

the prediction based on historical data, e.g., user query log or database access log.

In this paper, we propose B^p-tree, an adaptive indexing structure for PCM-based memory. We aim to devise new algorithms to reduce the number of writes without sacrificing search performance. Our B^p-tree is able to achieve much higher overall system performance than the classical B^+-tree. To summarize, we make the following contributions in this paper:

(1) We propose a new predictive B^+-tree index, called the B^p-tree, which is designed to accommodate the features of the PCM chip to allow it to work efficiently in PCM-based main memory systems. The B^p-tree can significantly reduce both number of writes and energy consumption.
(2) We develop a predictive model to predict data distribution based on current data, and we pre-allocate space on PCM for future insertions to reduce the key movements caused by node splits and merges encountered in a typical B^+-tree.
(3) We implemented our technique in the open source database PostgreSQL, and run it in an emulated environment. The experimental results show that our B^p-tree index significantly outperforms the state-of-the-art indexes.

The remainder of the paper is organized as follows. Section 2 introduces PCM and related work. Section 3 presents the overview and main components of the B^p-tree. We propose the predictive model for index warm-up and updates in Sections 4 and 5, respectively. Section 6 describes the metrics for evaluating the predictive model and how we can adjust the predictive strategy based on these metrics. In Section 7, we present the main experimental evaluation. Section 8 concludes the paper.

2 Background and Related Work

In this section, we will introduce the PCM technology and review some related work about the database design with new memory hierarchy.

2.1 PCM Technology

PCM is a non-volatile memory that exploits the property of chalcogenide glass to switch between two states, amorphous and crystalline. The amorphous phase tends to have a high electrical resistivity, while the crystalline phase exhibits a low resistivity, giving rise to the so-called phase change materials. The two different phases can be switched back-and-forth reliably and quickly. For a large number of times, the difference in resistivity is typically about five orders of magnitude [22], which can be used to represent the two logical states of binary data.

We present a brief comparison on the properties between PCM and other layers in the memory hierarchy, including DRAM, NAND Flash and HDD. Table 1 summarizes the properties of these different memory technologies, as presented in recent work [3], [4], [19] and the data all corresponds to raw memory chips.

From Table 1, we see that the PCM has promising characteristics. Compared with DRAM, PCM has a density advantage over DRAM. The read latency of PCM is comparable to that of the DRAM. Although writes are almost an order of magnitude slower than that of DRAM, some techniques like buffer organization or partial writes could be used in algorithms design to reduce the performance gap. Unlike NAND, PCM does not have the problem of erase-before-writes and supports random reads and writes more efficiently. Reads and writes are orders of magnitude faster than those of NAND and the endurance is also higher than that of NAND. In most cases, PCM is between DRAM and NAND Flash layer, and play a major role in the memory hierarchy, impacting system performance, energy consumption and reliability.

TABLE 1
Comparison of Memory Technologies

	DRAM	PCM	NAND	HDD
Density	1X	2-4X	4X	N/A
Read Latency	20-50ns	$\sim 50ns$	$\sim 25\mu s$	$\sim 5ms$
Write Latency	20-50ns	$\sim 1\mu s$	$\sim 500\mu s$	$\sim 5ms$
Read Energy	0.8J/GB	1J/GB	1.5J/GB	65J/GB
Write Energy	1.2J/GB	6J/GB	17.5J/GB	65J/GB
Endurance	∞	$10^6 - 10^8$	$10^5 - 10^6$	∞

Recent studies on embedding PCM in the memory hierarchy can be broadly divided into two categories. One is to replace DRAM with PCM directly to achieve larger main memory capacity. Though PCM is slower than DRAM, Lee *et al*. [10] have shown that some optimizations like buffer organization and partial writes can be used to improve the system performance while preserving the high density and non-volatile property. The other proposal is to replace the DRAM with a large PCM and a small DRAM (3% - 8% size of the PCM capacity [19], [20]) and use the DRAM as a buffer to keep some frequently accessed data to improve the system performance. In this paper, we will adopt the second approach with novel revisions.

From Table 1, we can also observe that if we want to replace the DRAM with PCM in the main memory system, one of the major challenges is to reduce the number of writes. Compared with reads, PCM writes incur higher energy consumption, higher latency and limited endurance. In this paper, the limited endurance is not our focus, since some optimizations like round robin or write leveling algorithms can be utilized when designing the PCM driver. We mainly focus on how to reduce energy consumption, latency and the number of writes.

2.2 Related Work

Database Algorithms Design for PCM. The recent study [4] has outlined new database algorithm design considerations for PCM technology and initiated the research on algorithms for PCM-based database systems. The analytic metrics for PCM endurance, energy and latency are presented and the techniques to modify the current B^+-tree index and Hash Joins based on the PCM-based database has been proposed.

PCM-based Main Memory System. Several recent studies from the computer architecture community have proposed new memory system designs on PCM. They mainly focused on how to make PCM a replacement or an addition to

the DRAM in the main memory system. Although these studies mainly focused on the hardware design, they provided us the motivation on the use of PCM in the new memory hierarchy design for database applications. The major disadvantages of the PCM for a main memory system are the limited PCM endurance, longer access latency and higher dynamic power compared to the DRAM. There are many relevant studies addressing these problems [10], [18], [19], [27]. In [19], Qureshi designed a PCM-based hybrid main memory system consisting of the PCM storage coupled with a small DRAM buffer. Such an architecture has both the latency benefits of DRAM and the capacity benefits of PCM. The techniques of partial writes, row shifting and segment swapping for wear leveling to further extend the lifetime of PCM-based systems have been proposed to reduce redundant bit-writes [10], [27]. Qureshi *et al*. [18] proposed the Start-Gap wear-leveling technique and analyzed the security vulnerabilities caused by the limited write endurance problems. Zhou *et al*. [27] also focused on the energy efficiency and their results indicated that it is feasible to use PCM technology in place of DRAM in the main memory for better energy efficiency. There are other PCM related studies such as, [24] focusing on error corrections, and [25] focusing on malicious wear-outs and durability.

Write-optimized B$^+$-tree Index. Write-optimized B$^+$-tree index has been an intensive research topic for more than a decade. For the B$^+$-tree index on hard disks, there are many proposals to optimize the efficiency of write operations and logarithmic structures have been widely used. In [16], O'Neil *et al*. proposed the LSM-tree to maintain a real-time low cost index for the database systems. It was designed to support high update rate efficiently. Arge proposed the buffer tree for the optimal I/O efficiency in [2]. Graefe proposed a new write optimized B$^+$-tree index in [9] based on the idea of the log-structured file systems [23]. Their proposals make the page migration more efficient and retain the fine-granularity locking, full concurrency guarantees and fast lookup performance as well.

Recently, there are some proposals on the write-optimized B$^+$-tree index on SSDs [1], [11], [26]. The major bottleneck of the B$^+$-tree index for SSD is to reduce the small random writes because of the erase-before-write. In [26], an efficient B-tree layer (BFTL) was proposed to handle the fine-grained updates of B-tree index efficiently. The implementation is in the flash translation layer (FTL) and thus there is no need to modify the existing applications. FlashDB proposed in [15] used a self-tuning database system optimized for sensor networks. The self-tuning B$^+$-tree index in the FlashDB uses two modes, Log and Disk, to make the small random writes together on consecutive pages. Li *et al*. [11] proposed the FD-tree which consists of two main parts, a head B$^+$-tree and several levels of sorted runs in the SSDs. It limits the random writes to the small top B$^+$-tree and convert many small random writes into sequential ones by merging.

There are also some proposals focusing on the Write-Once-Read-Many (WORM) storage [12], [17]. In [12], Mitra proposed an inverted index for keyword-based search and a secure jump index structure for multi-keyword searches. In [17], Pei proposed the TS-Trees and they also built the tree structure based on a probabilistic method. These WORM indexing proposals mainly focused on designing mechanisms to detect adversarial changes to guarantee trustworthy search. Unlike WORM indexing, we want to reduce the number of writes. Moreover, we can update the index and afford the small penalty of adjustments due to data movement if the prediction is no longer accurate because of changes in data distributions.

Cache-optimized Tree Indexing for Main Memory System. There are several proposals related to the cache-optimized tree indexing for main memory systems like Cache Sensitive B$^+$-Trees (CSB$^+$-Trees) [21], fractal prefetching-B$^+$-Trees (fpB$^+$-Trees) [5], BD-Trees [7], [8]. The target of them aim to make the index cache conscious. In [21], CSB$^+$-Trees only stores the address of the first child in each node and the rest of the children can be found by adding an offset to that address, because it can raise the utilization of a cache line and further make the whole tree cache-efficient. In [5], fpB$^+$-Trees is proposed to optimize both cache and disk performance. To make itself cache efficient, fpB$^+$-Trees increased the node size to be multiple cache lines wide and prefetched all cache lines within a node before accessing it. These B$^+$-Tree variants also focused on the main memory indexing, they want to make the structure cache-efficient. However, our focus is to make our indexing write-optimized.

3 Predictive B$^+$-Tree

In this section, we will first present the overview of B^p-tree index and then introduce the basic phases when constructing the index.

3.1 Overview of the B^p-Tree

Design Principle: Our goal is to reduce the number of writes of both insertions and updates without sacrificing query performance. This is achieved in two ways. First, we adopt the `Unsorted Leaf` strategy in [4]. Essentially, newly inserted keys are simply appended to the end of the key entries. They are not necessarily in sorted order. Second, we develop a predictive model to minimize data writes caused by node splits and merges.

Basic Idea: The general idea is to predict the data distribution based on the past insertions and pre-allocate space on PCM for accommodating future tree nodes, which can reduce the key movements caused by node splits and merges. Fig. 1 illustrates the main architecture of a B^p-tree. We use the following two techniques to implement a B^p-tree.

1) *DRAM Buffer*. We use a small DRAM buffer to maintain a small B$^+$-tree for current insertions. We also record the summary of previously inserted keys and use them to predict the structure of the B^p-tree. If the buffer is full, we will merge it into the B^p-tree on PCM.
2) *B^p-tree on PCM*. Like a standard B$^+$-tree, a B^p-tree is also a balanced multiway search tree. The key differences between the B^p-tree and the B$^+$-tree include: (1) The structures and nodes in a B^p-tree can be pre-allocated; (2) Given a branching factor $2M$ of a B^p-tree, the number of children of an internal

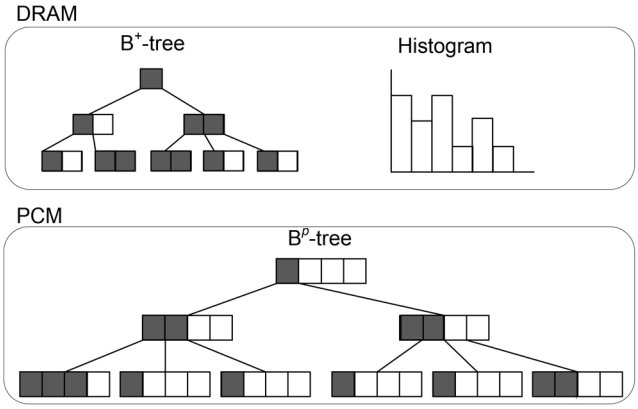


Fig. 1. B^p-tree architecture.

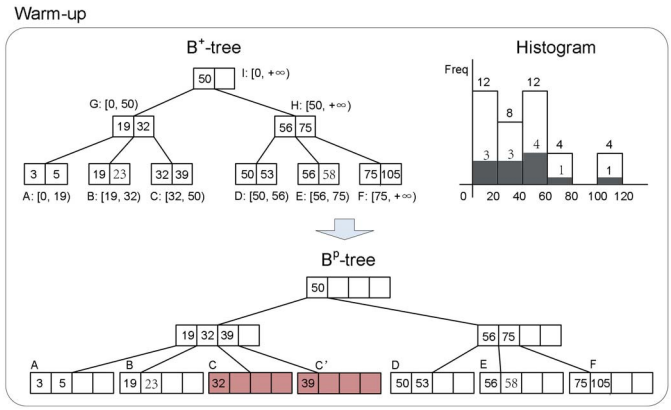


Fig. 2. Example of a warm-up phase.

node may be smaller than M, and the real number of children is between $[0, 2M]$; (3) The insertions and deletions are different from the B$^+$-tree (see Section 5); (4) The tree construction process consists of two phases: (*i*) **warm-up phase**: The first N keys are initially inserted into the tree as a warm-up process; (*ii*) **update phase**: All new keys are first inserted into a DRAM buffer. Each time the buffer is full, the keys in DRAM would be merged into the main tree on PCM. For a search query, we will find them from both the B$^+$-tree in DRAM and the B^p-tree in PCM.

3.2 Main Components of B^p-Tree

In this section, we will introduce the details of the construction process of the B^p-tree. It consists of two phases, the warm-up phase and update phase which will be described in Sections 3.2.2 and 3.2.3 respectively. For ease of presentation, we summarize the notations used throughout this paper in Table 2.

3.2.1 DRAM Buffer

As new keys are inserted into the the DRAM buffer, a small standard B$^+$-tree with branching factor $2m$ is built in buffer. If the buffer is full, we will flush the keys in the B$^+$-tree to the B^p-tree on the PCM.

To capture the data distribution, we also maintain a histogram. Suppose the range of the keys is $[L, U]$. If we want to partition the keys into buckets $B_1, B_2, \ldots, B_{|B|}$, the bucket width is $\frac{U-L}{|B|}$. For each bucket B_i, we maintain the number of keys that fall in this bucket. We will use the histogram to "forecast" the data distribution.

The main function of DRAM buffer is to adaptively adjust our predictive model based on the currently inserted keys in a time window. Then we can use the updated predictive model to merge all the keys in the time window in the B$^+$-tree to the B^p-tree on PCM.

TABLE 2
Notations

Parameter	Description
h	Height of the B^p-tree and B$^+$-tree
$2M$	The branching factor of the B^p-tree on PCM
$2m$	The branching factor of the B$^+$-tree on DRAM
K	M divided by m (K is an integer and $K \geq 1$)
B_i	The i-th bucket
N	The number of keys in leaf nodes buffer can hold
A	The maximum number of keys one bucket can hold
W	The bucket width
n	The node of the tree
d	The duplicate factor of key values

3.2.2 Warm-Up Phase

Initially, the B^p-tree on PCM is empty. We use a DRAM buffer for warm-up. We create a standard B$^+$-tree for supporting insertions, deletions and search. Before the buffer is full, we use the conventional B$^+$-tree for the initial operations. Once the DRAM buffer is full, all the keys in the buffer are moved to PCM, and this step is called the warm-up process. The main function of the warm-up phase is to construct the skeleton of the B^p-tree on PCM. Suppose the DRAM buffer can accommodate N keys. We first predict the total number of possible keys. Then, for each B$^+$-tree node, we use our predictive model to decide whether to split it in an eager manner to avoid writes for subsequent insertions. We will provide the details for constructing the initial B^p-tree in Section 4.

Fig. 2 shows an example for the warm-up phase. The B$^+$-tree and histogram are in the DRAM and B^p-tree is in the PCM. In this example, N is 12 and the buffer is full and thus we need to flush all the keys in the B$^+$-tree to the PCM. The black portion of the histogram bar indicates the number of inserted keys in each range so far, while the whole bar indicates the predicted number of keys in each range based on our predictive model. We find the structure of the B^p-tree is similar to that of the original B$^+$-tree. However, there are two key distinctions. First, the node could be split in an early manner if it meets the requirement of node splits. Second, some of the nodes could *underflow* due to either an enlargement of the node size or an early split. These are guided by our predictive model and tree construction strategy.

In the example, node C in the B$^+$-tree is split into nodes C and C' when it is moved to the B^p-tree, nodes B and E become underflow because of the enlargement of the node size, while node C and node C' are underflow because of the early split.

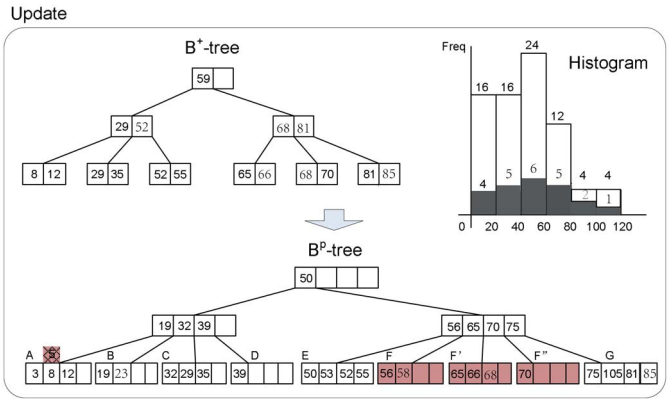


Fig. 3. Example for update phase.

3.2.3 Update Phase

After the warm-up phase, we have a B^p-tree structure on the PCM. Then for new operations, we use both the DRAM buffer and B^p-tree to handle the operations. For an insertion, we insert it into the B$^+$-tree. For a search query, we search the key from both the B$^+$-tree on DRAM and the B^p-tree on PCM. If we find it, we return the answer; otherwise we return "null". For delete, we search it from both the B$^+$-tree and the B^p-tree. If we find it, we remove it from the B$^+$-tree and the B^p-tree. However, even if a node "underflows" after deletions, we do not merge it with its siblings. The reason is that since the read latency of PCM is much less than the write latency, the overhead caused by empty nodes during query processing is negligible. Furthermore, space could be reserved for future insertion keys to reduce subsequent writes. For update operation, like other indexes, we treat it as a deletion operation followed by an insertion. Note that we need to update the histogram for the insertion and deletion operations. If the DRAM buffer is full, we need to merge the B$^+$-tree into the B^p-tree.

Fig. 3 shows an example for update phase following the earlier example described in Fig. 2. The case in Fig. 3 is that the buffer is full for the second time and all the keys in the B$^+$-tree are merged into the B^p-tree described in Fig. 2. In this example for the update phase, we want to delete the key 5 in the B^p-tree index. First, we search the B$^+$-tree in the buffer and cannot find it. Then we search B^p-tree on PCM and find it in node A and subsequently remove it from B^p-tree. As depicted in the figure, the histogram is updated to reflect the effect of this deletion and the new round of prediction is performed based on all the keys inserted currently including the keys in the buffer. Node F in the B^p-tree is split because of the similar reason as that of the node C in Fig. 2. We will describe the details in Section 5.

4 Predictive Model for Warm-Up

In this section, we introduce a predictive model to construct a B^p-tree structure in the warm-up phase. We first discuss how to predict the B^p-tree skeleton in Section 4.1, and then illustrate how to construct the B^p-tree in Section 4.2.

4.1 Predicting the B^p-Tree Skeleton

Suppose there are N keys in the DRAM B$^+$-tree, the height of the B$^+$-tree is h, and the branching factor is $2m$. Since we want to predict the skeleton first, the B^p-tree will have the same height as the B$^+$-tree, but with a larger branching order $2M = K * 2m$, where K is an integer and $K \geq 1$. K can be set by the administrator. We can also predict K as follows.

Let T denote the total number of possible keys in the whole dataset. We estimate T using the numbers of keys in the histogram. Suppose the maximal number of keys inside one bucket among all the buckets is A. Suppose the bucket width is $W = \frac{U-L}{|B|}$, thus there are at most W keys in a bucket. For duplicate keys, we calculate a duplicate factor d which means how many percent of keys among all the keys have duplicate values. We use $N \times \frac{(1+d)\times W}{A}$ to predict the possible key number T. As there are T keys in B^p-tree and N keys in B$^+$-tree, we set $K = \log_h \frac{T}{N}$.

Obviously, if we overestimate K, the B^p-tree will be quite sparse; on the contrary, if we underestimate the number, we may need to do more splits. We assume that each tree node in the final B^p-tree is expected to be μ% full, i.e., each leaf node has $E = \mu\% \times 2M$ keys. Thus the i-th level is expected to have E^i nodes (the root is the first level, which has only one node).

4.2 B^p-Tree Construction

In this section, we discuss how to construct the B^p-tree based on the current B$^+$-tree structure. We traverse the B$^+$-tree in post-order. For each node, we predict whether we need to split it based on our predictive model (which will be introduced later). If we need to split it, we split it into two (or more) nodes, and insert separators (or keys) into its parent node, which may in turn cause the parent node to be split. As we employ a post-order traversal, we can guarantee that the child splits are before the parent split, and our method can keep a balanced tree structure.

Next we discuss how to split a B$^+$-tree node. For ease of presentation, we first introduce a concept.

Definition 1 (Node Extent). *Each node n in the index tree is associated with a key range $[n_l, n_u)$, where n_l and n_u are respectively the minimum key and the maximum key that could fall in this node. We call this range the extent of the node.*

The extents of node A and B of the B$^+$-tree in Fig. 2 are $[0, 19)$ and $[19, 32)$ respectively. If a node is not the leftmost or the rightmost child of its parent, we can get its extent from its parent except for the root node; otherwise we determine it from its ancestors.

Consider a B$^+$-tree node n on the i-th level. Suppose its extent is $[\text{key}_{min}, \text{key}_{max})$ and currently it has $|n|$ keys, $\text{key}_1, \text{key}_2, \cdots, \text{key}_{|n|}$. We access the keys in order. Suppose the current key is key_j. We next discuss whether to split the node according to key_j as follows. As key_{min} is the possible minimum key in the node, we first estimate the number of possible keys between key_{min} and key_j, denoted by $P(\text{key}_{min}, \text{key}_j)$. Then we estimate the number of keys that can be accommodated between key_{min} and key_j on the B^p-tree, denoted by $A(\text{key}_{min}, \text{key}_j)$.

Obviously if $P(\text{key}_{min}, \text{key}_j) < A(\text{key}_{min}, \text{key}_j)$, we do not need to split the node according to key_j; otherwise we need to split the node. We generate a B^p-tree node with

keys $\mathsf{key}_{min}, \ldots, \mathsf{key}_{j-1}$, remove the keys $\mathsf{key}_{min}, \ldots, \mathsf{key}_{j-1}$ from the DRAM B$^+$-tree node, insert the key key_j to its parent on DRAM B$^+$-tree, and update the pointers of the B$^+$-tree node: the left pointer of this key points to the B^p-tree node, and the right pointers of this key points to the B$^+$-tree node. Next we repeatedly split the node with keys $\mathsf{key}_j, \ldots, \mathsf{key}_{|n|}$ (Note that key_j turns to the first key in the new node). If we cannot split the node for the last key, we will create a B^p-tree node with the same keys in the B$^+$-tree node, and update the pointer of its parent to the B^p-tree node. Next we discuss how to predict $A(\mathsf{key}_{min}, \mathsf{key}_j)$ and $P(\mathsf{key}_{min}, \mathsf{key}_j)$.

Predicting the number of possible keys between key_{min} and key_j: If key_{min} and key_j are in the same bucket B_s, we can estimate $P(\mathsf{key}_{min}, \mathsf{key}_j)$ as follows. Based on the histogram, there are n_s keys in the bucket. Then the number of keys between key_{min} and key_j can be estimated by $(\mathsf{key}_j - \mathsf{key}_{min}) \times \frac{n_s}{W}$. Thus the number of possible keys in the range is

$$P(\mathsf{key}_{min}, \mathsf{key}_j) = K \times (\mathsf{key}_j - \mathsf{key}_{min}) \times \frac{n_s}{W}, \tag{1}$$

if key_{min} and key_j are in the same bucket B_s.

On the contrary, if key_{min} and key_j are in different buckets, we estimate the number as follows. Without loss of generality, suppose key_{min} is in bucket B_s and key_j is in bucket B_e. Let B_s^u denote the upper bound of keys in bucket B_s and B_e^l denote the lower bound of keys in bucket B_e. Thus the number of keys between key_{min} and key_j in bucket B_s is $(B_s^u - \mathsf{key}_{min}) \times \frac{n_s}{W}$. The number of keys between key_{min} and key_j in bucket B_e is $(\mathsf{key}_j - B_e^l) \times \frac{n_b}{W}$. Thus the total number of keys between key_{min} and key_j is $(B_s^u - \mathsf{key}_{min}) \times \frac{n_s}{W} + \sum_{t=s+1}^{e-1} n_t + (\mathsf{key}_j - B_e^l) \times \frac{n_b}{W}$. Thus the number of possible keys between key_{min} and key_j is

$$P(\mathsf{key}_{min}, \mathsf{key}_j) = K \times \Big((B_s^u - \mathsf{key}_{min}) \times \frac{n_s}{W} + \sum_{t=s+1}^{e-1} n_t + (\mathsf{key}_j - B_e^l) \times \frac{n_b}{W}\Big), \tag{2}$$

if key_{min} and key_j are in different buckets.

Predicting the number of keys that can be accommodated between key_{min} and key_j: Node n has $|n|$ keys and it is expected to have E keys, thus the number of accommodated keys in this node is $E - |n|$. Thus

$$A(\mathsf{key}_{min}, \mathsf{key}_j) = \min(\mathsf{key}_j - \mathsf{key}_{min}, E - |n|),$$

if n is a leaf node.

If node n is a non-leaf node, we can directly add j children between the two keys. In addition, we can also add some keys between key_{min} and key_j as there are $E-|n|$ positions which are not used in the node. Obviously, we insert at most $\min(\mathsf{key}_j - \mathsf{key}_{min}, E - |n|)$ keys in the node. Thus we can add at most $c = j + \min(\mathsf{key}_j - \mathsf{key}_{min}, E - |n|)$ children under the node between key_{min} and key_j. As node n is in the i-level, the children are on $i+1$-level. As each child can have E keys and $E+1$ children, each node can have $(E+1)^{h-i-1}$ descendants. Thus there are $c \times \sum_{t=0}^{h-i-1}(E+1)^t$ nodes between the two keys. As each node can accommodate E keys, the total number of accommodated keys is

$$A(\mathsf{key}_{min}, \mathsf{key}_j) = E \times c \times \sum_{t=0}^{h-i-1} (E+1)^t, \tag{3}$$

if node n is a non-leaf node.

To summarize, we can use the predicted numbers to split the nodes. Iteratively, we can split all the nodes and insert the new nodes into PCM. Fig. 4 illustrates the algorithm. The Warm-up algorithm first traverses the B$^+$-tree in post-order by calling its function PostOrder (line 4). Function PostOrder splits the nodes iteratively. Given a node n on level i, it checks whether the node should be split by calling function Split (line 4), which is used to split a node based on our predictive model. If our model decides to split node n, we generate a B^p-tree node with keys, $\mathsf{key}_{min}, \ldots, \mathsf{key}_{j-1}$ (line 6), remove the keys, $\mathsf{key}_{min}, \ldots, \mathsf{key}_{j-1}$, from the DRAM B$^+$-tree node (line 7), insert the key key_j to its parent on DRAM B$^+$-tree (line 8), and update the pointers of the B$^+$-tree node: the left pointer of this key to the B^p-tree node, and the right pointers of this key to the B$^+$-tree node. Next we repeatedly split the node with keys, $\mathsf{key}_j, \ldots, \mathsf{key}_{|n|}$ (line 9). If we cannot split the node for the last key, we will create a B^p-tree node with the keys, and update the pointer of its parent to the B^p-tree node (line 10). Iteratively, we can construct the B^p-tree structure.

As shown in Fig. 2. The previous node C is split into two nodes C and C', though it is not full. Since there are only two keys in previous node C, we shall calculate $\mathrm{A}(\mathsf{key}_{min}, \mathsf{key}_2)$ and $\mathrm{P}(\mathsf{key}_{min}, \mathsf{key}_2)$ as follows. $\mathrm{A}(\mathsf{key}_{min}, \mathsf{key}_2) = \min(39-32, 4-2) = 2$, $\mathrm{P}(\mathsf{key}_{min}, \mathsf{key}_2) = (39-32) * \frac{8}{20} = 2.8$. As $\mathrm{A}(\mathsf{key}_{min}, \mathsf{key}_2) < \mathrm{P}(\mathsf{key}_{min}, \mathsf{key}_2)$, according to the algorithms in Fig. 4, node C needs to be split and a new node C' is created and then we update the pointers.

5 Predictive Model for Updates

In this section we propose a predictive model for the update phase and we will describe the basic operations of the B^p-tree. We will discuss the search and deletion operations in Sections 5.1 and 5.2 respectively. The insertion will be finally illustrated in Section 5.3. The search and deletion operations are generally the same as those in the standard B$^+$-tree except that we have to deal with unordered leaf entries. While the insertion is quite different and is the major point we will discuss in the following section.

5.1 Search

Since we use a small DRAM as a buffer, some newly inserted keys will still be in the main memory B$^+$-tree and have not been merged into the main B^p-tree on PCM. Thus, both the B$^+$-tree and B^p-tree need to be searched. We first lookup the small B$^+$-tree in the buffer, and then search the main B^p-tree. As noted, since the entries within a leaf node of the B^p-tree may not be sorted, the search will examine every key entry. If neither of the two steps

Algorithm 1: Warm-up(B^+-tree, Histogram)

Input: B^+-tree and Histogram in DRAM Buffer
Output: B^p-tree on PCM
begin
  Let r denote the root of the B^+-tree;
  Level $i = 0$;
  PostOrder (r, i, Histogram) ;
end

Function PostOrder (n, i, *Histogram*)

Input: n: B^+-tree node; i: Level of n; Histogram
Output: B^p-tree nodes
begin
  for *each child* c *of* n **do**
    PostOrder (c, $i+1$, Histogram);
  key_j = Split (n, i, Histogram);
  while key_j ! = ϕ **do**
    Generate a B^p-tree node with $key_{min}, \cdots, key_{j-1}$;
    Remove keys before key_j from node n;
    Insert key_j to the parent of n on B^+-tree and update the pointers ;
    Split(n, i, Histogram);
  Generate a B^p-tree node with $key_{min}, \cdots, key_{j-1}$, remove node n, update the pointer of n's parent;
end

Function Split (n, i, *Histogram*)

Input: n: B^+-tree node; i: Level of n; Histogram
Output: Key: Split Key
begin
  Let key_{min} denote the first key in node n ;
  for $j = 2, 3, \cdots |n|$ **do**
    if *PredictTwoKeys* (key_{min}, key_j, i) **then**
      return key_j ;
  return ϕ ;
end

Function PredictTwoKeys (key_{min}, key_j, *Histogram*)

Input: key_{min}; key_j; Histogram
Output: True or false
begin
  Compute A(key_{min}, key_j) ;
  Compute P(key_{min}, key_j) ;
  if $A(key_{min}, key_j) < P(key_{min}, key_j)$ **then** return true;
  else return false;
end

Fig. 4. Warmup algorithm.

return any results, null will be returned. The above steps are summarized in Fig. 5. Obviously the time complexity of the search operation is $\mathcal{O}(h)$, where h is the height of the B^p-tree.

Algorithm 2: Search(B^+-tree, B^p-tree, key)

Input: B^+-tree; B^p-tree; A search key
Output: Search result
begin
  Search both B^+-tree and B^p-tree using key ;
  if *Find key in* B^+*-tree or find key in* B^p*-tree* **then**
    return the entry or entries;
  else
    return null;
end

Fig. 5. B^p-tree: Search operation.

5.2 Deletion

Similar to the search operation, the deletion operation also requires searching both B^p-tree and B^+-tree. A deletion on the B^p-tree is handled in a similar way as that for standard B^+-tree with two differences. First, the deleted entry can be replaced by the last key entry in the node. This is to pack the entries within the leaf node. Second, if the corresponding leaf node has fewer than M keys, we will not borrow keys from its siblings. This can avoid the merge operations. The reason is that since the read latency of PCM is much shorter than the write latency, the overhead caused by the empty node in the query processing stage is negligible. Furthermore, the space could be reserved for the future keys to reduce subsequent writes. Note that we also need to update the histogram.

Given a key to delete, we first search it in B^+-tree. If we find the entry, we directly remove it. Otherwise we search it in B^p-tree. If we find the leaf node in the B^p-tree, we remove the key from the node. Note that we will not do merge operations even if the node has less than half (M) keys. We do not propagate the deletion operation to its ancestors. The above steps are summarized in Fig. 6. Obviously the time complexity of the deletion operation is $\mathcal{O}(h)$.

Algorithm 3: Delete(B^+-tree, B^p-tree, key)

Input: B^+-tree, B^p-tree, key
Output: Delete status
begin
  result $\leftarrow$ false ;
  Search the B^+-tree using key ;
  if *Find key in* B^+*-tree* **then**
    delete the entry in a B^+-tree operation manner;
    result $\leftarrow$ true ;
  Search the B^p-tree using key ;
  if *Find key in* B^p*-tree* **then**
    remove the entry from the leaf node;
    result $\leftarrow$ true ;
  return result ;
end

Fig. 6. B^p-tree: Deletion operation.

Algorithm 4: Insert(B$^+$-tree, B^p-tree, key)

Input: B$^+$-tree, B^p-tree, key
Output: Updated B^p-tree
1 **begin**
2 Search the leaf node for key, L ;
3 Upgrade (L, Histogram);
4 **end**

Function Upgrade (n, *Histogram*)

Input: n: A B^p-tree node, Histogram
1 **begin**
2 $PNO_n \leftarrow$ GetPredictedNumber(n);
3 $ANO_n \leftarrow$ GetAccomodatedNumber(n);
4 **if** $PNO_n < ANO_n$ **then**
5 insert key into n;
6 return;
7 **else**
8 $midKey \leftarrow GetMiddleKey(n)$;
9 $midKey$ is pushed upward to the parent p ;
10 A new leaf node n' is created and new pointer from the parent node to n' ;
11 Remove keys larger than $midKey$ from n to n' ;
12 Upgrade (p, Histogram) ;
13 **end**

Fig. 7. B^p-tree: Insertion operation.

5.3 Insertion

B^p-tree is designed by adding the DRAM buffer and a predictive model, thus both the B$^+$-tree in buffer and the histogram of the predictive model need to be instantly updated. When the buffer is full, the B$^+$-tree will be merged into the main B^p-tree on PCM.

All the keys in the B$^+$-tree will be inserted into the main tree one by one. Once a key is to be inserted, we first look up the leaf node L that the new key belongs to as the standard B$^+$-tree. Then we predict whether it should be directly inserted into the node or the node should be split. We first compute the number of keys that can be accommodated in this node L, denoted by ANO_L. We then predict the number of keys that could fall in this node, denoted by PNO_L. If $PNO_L \geq ANO_L$, we need to split the node; otherwise, we will not. If we need to split the node, a new leaf node will be created and a "middle" key will be chosen based on the predictive model and pushed upward to the parent node. Existing keys in the node L needs to be adjusted according to the "middle" key. The middle key is not the key in the median position as B$^+$-tree. Instead, we need to select a median key based on the data distribution. As we insert a middle key into its parent, it may cause its parent to split. The above steps are summarized in Fig. 7. Next, we discuss how to compute PNO_n and ANO_n for node n.

Computing the accommodated key number of node n, ANO_n: Suppose node n is in the i-th level. Each node has at most $2M$ keys and 2M+1 pointers, thus node n has $\sum_{t=1}^{h-i}(2M+1)^t$ descendants. Thus the accommodated key number of node n is

$$ANO_n = 2M * \sum_{t=0}^{h-i}(2M+1)^t. \quad (4)$$

Predicting the possible key number occupancy in node n, PNO_n: Next we predict the total number of keys that could potentially belong to this node. We first find the extent of this node, denoted by [key$_{min}$, key$_{max}$], where key$_{min}$ and key$_{max}$ are respectively the minimum key and the maximum key in this node. Based on the two bounds, we can compute the number of possible keys fell into this node as discussed in Section 4.2.

If key$_{max}$ and key$_{min}$ are in the same bucket B_s,

$$PNO_n = K \times (\text{key}_{max} - \text{key}_{min}) \times \frac{n_s}{W}; \quad (5)$$

otherwise if key$_{min}$ and key$_{max}$ are respectively in two different buckets B_s and B_e.

$$PNO_n = K \times \Big((B_s^u - \text{key}_{min}) \times \frac{n_s}{W} + \sum_{t=s+1}^{e-1} n_t + (\text{key}_{max} - B_e^l) \times \frac{n_b}{W}\Big). \quad (6)$$

Based on ANO_n and PNO_n, we can decide whether to split a node n. Next we discuss how to select a middle key if we need to split a node.

Computing the middle key in node n, *midKey*: Consider the keys in n are key$_1$, key$_2$, ..., key$_{|n|}$. Without loss of generality, suppose key$_1 \leq$ key$_2 \leq \cdots \leq$ key$_{|n|}$. Based on extent of a node, we define the middle key formally.

Definition 2 (Middle Key). *A key key$_i$ in node n is called a middle key if*

$$P(\text{key}_{min}, \text{key}_i) \leq \frac{P(\text{key}_{min}, \text{key}_{max})}{2},$$

$$P(\text{key}_{min}, \text{key}_{i+1}) > \frac{P(\text{key}_{min}, \text{key}_{max})}{2},$$

where P(key$_i$, key$_j$) denote the number of predicted keys between key$_i$ and key$_j$.

A straightforward method to find the middle key from a node is to compute P(key$_{min}$, key$_i$) for each i from 1 to $|n|$ until we find the middle key. The complexity is $\mathcal{O}(M)$. If the keys are sorted, e.g., the keys in an internode, we can use an alternative method. We have an observation that if the keys are sorted, P(key$_{min}$, key$_i$) $\leq P$(key$_{min}$, key$_j$) for $i < j$ as formalized in Lemma 1. Thus we can employ a binary search method to find the middle key and reduce the time complexity to $\mathcal{O}(\log M)$. If the keys are unsorted, the complexity is $\mathcal{O}(M)$.

Lemma 1. *Given a node n with keys ordered as key$_1$, key$_2$, ..., key$_{|n|}$, and two keys key$_{min} \leq$ key$_1$ and key$_{max} \geq$ key$_{|n|}$, we have*

$$P(\text{key}_{min}, \text{key}_i) \leq P(\text{key}_{min}, \text{key}_j)$$

for $i < j$.

Proof. Based on the definition of $P(\mathsf{key}_{min}, \mathsf{key}_i)$ in Equation 1, $P(\mathsf{key}_{min}, \mathsf{key}_i)$ monotonically increases with the increase of i. Then the lemma is proved. □

Thus the worst-case time complexity of an insertion operation is $\mathcal{O}(M + h \times \log M)$, where M is the branching factor and h is the height.

6 Evaluating B^p-Tree

In this section, we will first introduce some metric to evaluate the status of our index. Then we will discuss the concurrency problem of our index. Finally, the recovery techniques of our index are illustrated.

6.1 Metric Evaluation

In this section, we introduce several metrics to evaluate the status of B^p-tree and use them to guide prediction which is necessary to keep the tree balanced.

The first metric is insertion overflow. When inserting a new entry into the B^p-tree, we employ a leaf-to-root way, that is we always insert a key into leaf node first. If the node overflows, it needs to be split and some of the keys need to be moved to other nodes. Obviously, the larger the number of keys in a node is, the higher the probability to split will be. Thus we can use the number of keys in a node to evaluate the degree of insertion overflow.

The second metric is unqualified-node ratio. A node is called an *unqualified node* if its key number is smaller than M. If there are many unqualified nodes, the constructed B^p-tree is very sparse. For a node n, the smaller the value of n_{keys}, the sparser the tree will be. To evaluate the overall B^p-tree, we need to consider all tree nodes. Let n_{un} denote the number of unqualified nodes. The larger the value of n_{un}, the sparser the B^p-tree is. We can easily determine n_{un} as follows. Initially, all nodes are unqualified nodes and n_{un} is the total number of nodes. When inserting a key, if an unqualified node turns to be a qualified node (with key number no smaller than M), we decrease the number n_{un} by 1.

Next we combine the above two factors to evaluate a B^p-tree. As the expected utilization is $\mu\%$ and then the average key number of a node is $\mu\% \times 2M$, we can use the following equation to evaluate the B^p-tree,

$$Q = \sum_{n} \delta \times (n_{keys} - \mu\% \times 2M), \tag{7}$$

where

$$\delta = \begin{cases} \frac{n_{keys}}{\mathsf{key}_{max} - \mathsf{key}_{min}} & (n_{keys} \geq \mu\% \times 2M) \\ \frac{\mathsf{key}_{max} - \mathsf{key}_{min}}{n_{keys}} & (n_{keys} < \mu\% \times 2M) \end{cases} \tag{8}$$

and $[\mathsf{key}_{min}, \mathsf{key}_{max}]$ is the extent of node n and μ is 69 for the standard B$^+$-tree in [6].

Formula (8) is used to reflect the filling degree (sparse or dense) of the nodes. If Q is larger than 0, then it means that the B^p-tree is very dense. The larger the value of Q, the denser the B^p-tree will be. However it may involve many more numbers of writes when the tree needs to be reorganized. If Q is larger than an upper bound τ_u, we need to tune our model to do more splits when merging B$^+$-tree with B^p-tree. For example, in the extreme case when many keys are inserted in order, they will be firstly inserted into the top DRAM B+-tree. After the DRAM B+-tree becomes full, if they are merged into the B^p-tree, some nodes in B^p-tree may become very dense (i.e., the metric Q is larger than an upper bound τ_u), then we will tune our model to do more splits during the real merge process. Thus the extreme case can be solved and performance of our index can be guaranteed.

Conversely, if Q is smaller than 0, B^p-tree is very sparse. The smaller the value of Q, the sparser the B^p-tree will be. If Q is smaller than a lower bound τ_l, we need to tune our method to reduce the number of splits. Thus in the worst searching cases, when each of the nodes in the index reaches the lower bound τ_l, the searching complexity can be $log_{(\mu\% \times 2M - |\tau_l|)}\ N_t$, N_t is the number of the total entries. However, in most cases, the searching bound can be $log_M N_t$.

6.2 Concurrency Discussion

Next we briefly discuss the concurrency performance of our B^p-tree. Admittedly, we have added the DRAM B+ tree part. The accesses to DRAM B+ tree are excluded by write lock for a short period when merging happens. The range of entries to be merged in the DRAM B+ tree can be pre-calculated and a write lock that would happen on the range can be easily utilized. Then the CPU time can be well saved in a batch fashion during the merging process without attempts to consider each entry individually. In addition, the size of the DRAM B+ tree is very small. Above all, the basic structure of our proposed index has few modifications compared with B+ tree index. Therefore our proposed index has comparable concurrency performance with B+ tree and the traditional concurrency techniques used for the B+ tree can be applied to our index with minor revisions.

6.3 Recovery Discussion

Of course, we are faced with the classical recovery problem: to recover the work that has been done in memory after a crash. Actually, we do not need to create special logs for the newly inserted indexes. Because the transactional insert logs for the newly entries can be written out to a sequential log file during the normal course of events. It is the matter to treat the insert logs as a logical base for reconstruction. This minor revision can be easily applied to the system recovery algorithm. Further, the size of the DRAM B+ tree is very small and this will make little sense to overall performance of our proposed index.

7 Experimental Evaluation

In this section, we evaluate the performance of our proposed B^p-tree. We start by introducing the simulation platform and experimental setup. Then extensive experiments are conducted to evaluate the effectiveness and efficiency of our B^p-tree index.

TABLE 3
Parameters and Their Value Ranges

Parameter	Value Ranges
Size of DRAM buffer	5% of the size of the PCM used
Size of cache line	64B
Size of the B$^+$-tree node	256B (4 cache lines)
Size of the B^p-tree node	256B, 512B, 1024B (4, 8, 16 cache lines)
K	1, 2, 4
Number of keys in the data set	5 millions

7.1 Experimental Setup

Experimental platform. Similar to the method proposed in [4], we also extended a cycle-accurate out-of-order X86-64 simulator, PTLsim[1], with PCM support. We model data comparison writes for PCM writes. When writing a cache line to PCM, we compare the new line with the original line to compute the number of modified bits and the number of modified words. The former is used to compute PCM energy consumption, while the latter impacts PCM write latency. In addition, we record the execution time in cycles for the entire operation. Based on the benchmark used in [3], [4], [19], the parameters are set as follows: the read latency of a PCM cache line is 288 cycles; the write latency of PCM is 281 cycles for every 4 bytes; the read latency of a DRAM cache line is 115 cycles and the write latency of DRAM is 7 cycles for every 4 bytes. In PCM, the energy consumption is estimated as: the read energy per bit is 2pJ and the write energy per bit is 16pJ. In Table 3, we list the other parameters used in our experiments and their value ranges.

The experiments were conducted in CentOS release 5.6 with g++ 4.1.2. Our system is powered with a 16-core Intel Xeon E5620 2.4GHz CPU and 64GB main memory.

Datasets and workloads. In order to fully evaluate the performance of our proposed index, we utilize two synthetic and one real datasets in our experiments and next we will introduce the details.

For the synthetic datasets, one is generated to follow the uniform distribution, the other follows the skewed distribution. The skewed data is with the Zipf factor 1. We generate 5 millions distinct keys in each dataset. Each index entry contains a 4-Byte key and a 4-Byte pointer. We generate various workloads to evaluate the performance of our index. The search queries are composed of both the point queries and range queries. Based on the experimental results, we find that similar to the skewed dataset the performance of our proposed index on uniform dataset also shows obvious priority over the other indexes. Thus due to the space limitation we only discuss the results on the skewed.

The real dataset is obtained from the TPC-C Order table. The TPC-C workload showed higher temporal and spatial localities of index operations than synthetic workloads.

In our experiments, the node size of the DRAM B$^+$-tree is 256B, which is equivalent to 4 cache lines; whereas the node size of all tree structures on the PCM varies from 256B, 512B to 1024B. The size of the DRAM buffer used is approximately 5% of the size of the PCM. The update operation is processed as a deletion operation followed by an insertion operation.

Algorithms compared. We compare four different indexing structures including our B^p-tree, the traditional B$^+$-tree, the proposed Unsorted Leaf tree in [4] and our B^p-tree with sorted leaf nodes.

As the B^p-tree is a composite structure including the main B^p-tree on PCM and the small buffer B$^+$-tree on DRAM, we need to determine how to compute each performance metric first. In the performance evaluation, we only consider the PCM cost while calculating the number of writes and the energy consumption. As for the CPU cost, the CPU cycles occupied for manipulating the DRAM B$^+$-tree are recorded, which can represent a more accurate processing time.

In all figures presented in this section, "BP-tree" represents our B^p-tree; "B-tree" represents the traditional B$^+$-tree; "Unsorted" represents the proposed unsorted leaf tree in [4]; and "BP-minus" represents B^p-tree with sorted leaf nodes. The x-axis represents the node size of the corresponding tree, e.g., x-coordinates 4 indicates that the node size of the corresponding tree is 4 cache lines.

7.2 Results and Analysis

7.2.1 Insertion

First we evaluate the insertion performance of B^p-tree. In Fig. 8, we compare the insertion performance of four tree indexing schemes. The three subfigures correspond to our three metrics respectively. In each subfigure, we present the performance of the four tree structures with three different node sizes. The scale of y-axis in Fig. 8(a) and (b) are both

1. http://www.ptlsim.org

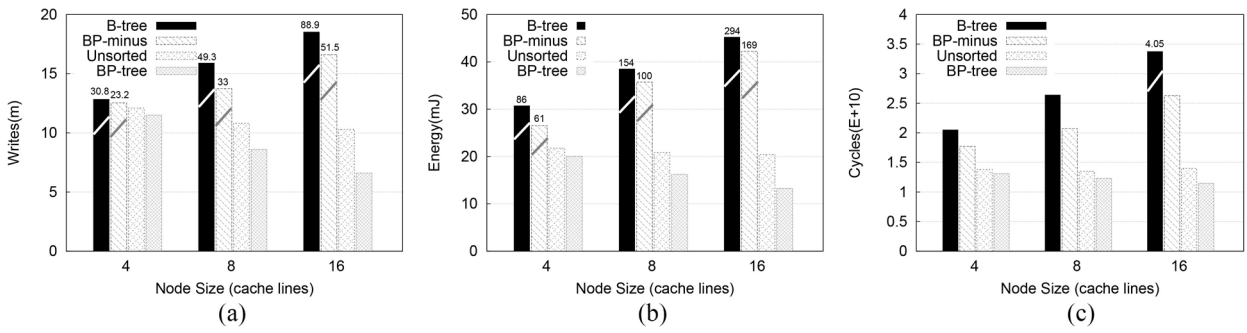


Fig. 8. Insertion performance. (a) Writes. (b) Energy. (c) Cycles.

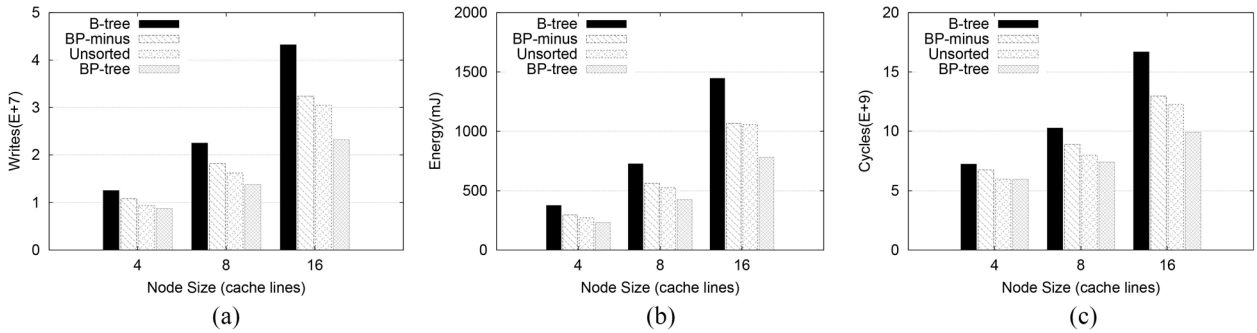


Fig. 9. Update performance. (a) Writes. (b) Energy. (c) Cycles.

in millions. We get two interesting observations from the results.

First, our B^p-tree achieves the best performance on all the three metrics and the performance gap increases as the node size becomes larger. The reason is that for large node sizes, our predictive model can estimate the splits more accurately, which can significantly reduce the number of writes by avoiding online splitting. On the other hand, Unsorted outperforms B$^+$-tree and BP-minus. This is because most writes will appear in leaf nodes and Unsorted can reduce the number of writes on leaf nodes. Our B^p-tree outperforms the Unsorted scheme, as it splits the nodes in advance, which can reduce the numbers of future splits. B^p-tree incurs about 5%, 22%, 37% less PCM writes than the Unsorted scheme on the three different node sizes respectively. For energy consumption, the result is very similar to that of the writes. For CPU cycles, the gap becomes slightly smaller because B^p-tree incurs extra CPU costs on the small B$^+$-tree in the DRAM buffer. However, B^p-tree still performs better than the Unsorted tree by a factor of 18% when the node size is 16 cache lines.

Second, we compare the performance of the two tree indexes with sorted leaf nodes, BP-minus and B$^+$-tree. BP-minus outperforms B$^+$-tree in all metrics. BP-minus reduces about 25%, 33%, 42% of writes compared with B$^+$-tree. Similar trend is observed in energy consumption. For CPU cycles, the gap is not that significant due to the extra cost on the small B$^+$-tree in DRAM buffer. However, BP-minus still reduces 14%, 22%, 35% more cost than that of B$^+$-tree.

7.2.2 Update

In this section, we evaluate the update performance of B^p-tree. We first insert all the keys as the previous insertion experiment and then we generate 100k update queries randomly. The update query consists of two keys, *oldKey* and *newKey*. We first search the *oldKey*. If we find it, we delete it and insert the *newKey*. Otherwise, we will ignore the insertion request. In Fig. 9, we compare the average update performance of our B^p-tree with the other three indexes. The result is similar to that of the insertion performance.

Our B^p-tree still achieves the best performance among all the three measures. The main reason is that our B^p-tree can predict future insertions and can pre-allocate space to reduce the number of writes. Compared to Unsorted, our B^p-tree reduces 24% of the writes, 26% of the energy and 19% of the CPU cycles, when the node size is 16 cache lines.

Compared with the traditional B$^+$-tree, the performance of BP-minus is better. It reduces 14% of the writes, 22% of the energy and 7% of the CPU cycles and the gap increases as the node size becomes larger.

7.2.3 Search

The philosophy of B^p-tree is two-fold: 1) B^p-tree is designed to reduce the number of writes on PCM, and 2) B^p-tree should be efficient for query processing as well. In this section, we evaluate the search performance of B^p-tree. The experiments include point queries and range queries. We experiment on both the uniform and skewed datasets. We first insert all keys into the index. Then for both point query and range query, we randomly generate 50k queries and calculate the CPU cycles during the processing.

In Fig. 10, we compare the search performance of the four tree indexes. The left subfigure is for point query and the right one is for range query. The y-axis represents the total CPU cycles to run these search queries. For point query, the performance of B^p-tree is better than Unsorted.

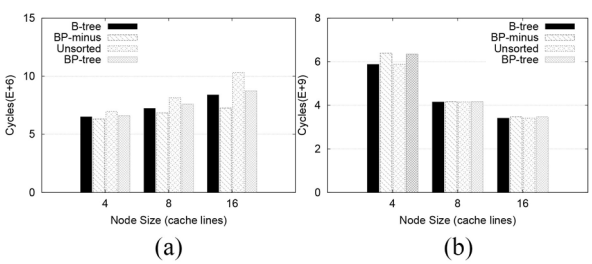


Fig. 10. Search performance. (a) Point query. (b) Range query.

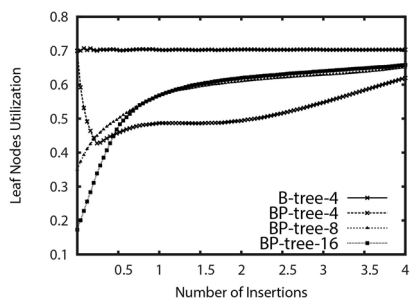


Fig. 11. Leaf nodes utilization.

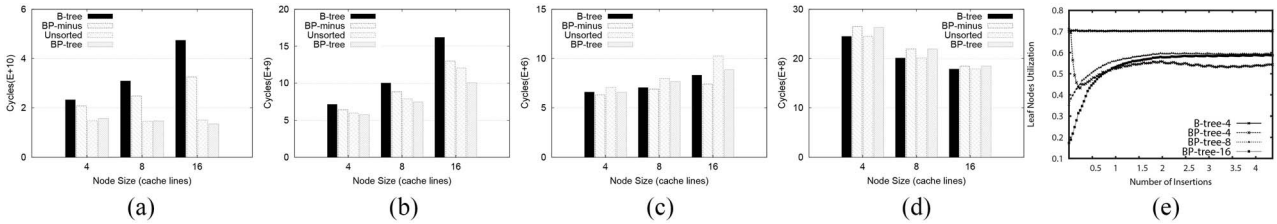


Fig. 12. Sensitivity to data distribution changes. (a) Insertion. (b) Update. (c) Point search. (d) Range search. (e) Leaf node utilization.

This is because when we process the search query, we simply scan the node and once we find the key, we will return the associated data. If the B^p-tree has more leaf nodes than Unsorted, some keys located in the right part of some nodes in Unsorted may be in the left part of some nodes in B^p-tree and thus more cache line reads are needed. The performance of BP-minus is better than that of Unsorted, which is expected since each search in Unsorted should read all the keys in the leaf node.

For range query, we can find that when the node size is 4 cache lines, the performance of our B^p-tree and BP-minus is worse than that of the B$^+$-tree and Unsorted. Because when the node size is small, the tree will be more sensitive to the split strategy and generate more leaf nodes which could affect the range search performance. When the node size is larger, all the four tree indexes show a similar performance.

7.2.4 Node Utilization

In this experiment, we compare the leaf node utilization of the B^p-tree and the traditional B$^+$-tree. Fig. 11 shows the results. The experiments are same as the insertion performance experiment and we build the two trees based on the same data set and calculate the leaf nodes utilization periodically during the insertion. The scale of the x-axis is 0.5 million. The suffixes -4, -8, -16 in the figure indicate different node sizes. As we can see in the figure, the leaf node utilization of the B$^+$-tree is stable, around 70% which is close to our assumption in Section 6. When the node size of the B^p-tree is 4 cache lines which is the same as that of the B$^+$-tree, the utilization is similar to that of the B$^+$-tree at first and then decreases as early splits happen and then it increases as the evaluation metrics described in Section 6 starts to work. When the node size is 8 cache lines, the utilization is smaller than that of the B$^+$-tree at first because of the enlargement of the node size and then it starts to increase. The result for the node size of 16 cache lines is similar. The stable utilization of all the three different B^p-tree indexes are all slightly smaller than that of the traditional B$^+$-tree, but according to the previous range search experiment, the influence of the utilization gap on the range search performance is not obvious.

7.2.5 Sensitivity to Data Distribution Changes

In this section, we evaluate the sensitivity of our predictive model when data distribution changes. We change the dataset as follows. The size of the dataset is 5 millions and the dataset follows a skewed (Zipf) distribution. However, we gradually change the Zipf factors and add a random offset every one million keys generated, resulting in a change of the data distribution. We did the insertion, update, search experiments as in previous sections. In Fig. 12, we show comparisons of the CPU cycles of all the four tree indexes with respect to different operations.

Meanwhile we find the performance of insertion, update and point search is very similar to that of the previous experiments. For the leaf node utilization, when the node size is 4 cache lines, the trend of the first half is similar to that of the previous result, but the utilization decreases slightly as the second half starts and increases again at last. Because the changes of data distribution caused a wrong prediction from the predictive model and further caused some improper splits. After that the predictive model adjusts its prediction via the evaluation metrics and makes the structure normal again.

Fig. 12(e) reveals the stable utilization value is a bit smaller than that of the previous experiments, which may have also caused the range search performance to degrade slightly as shown in Fig. 12(d).

To sum up, the B^p-tree achieves excellent performance under different data distribution.

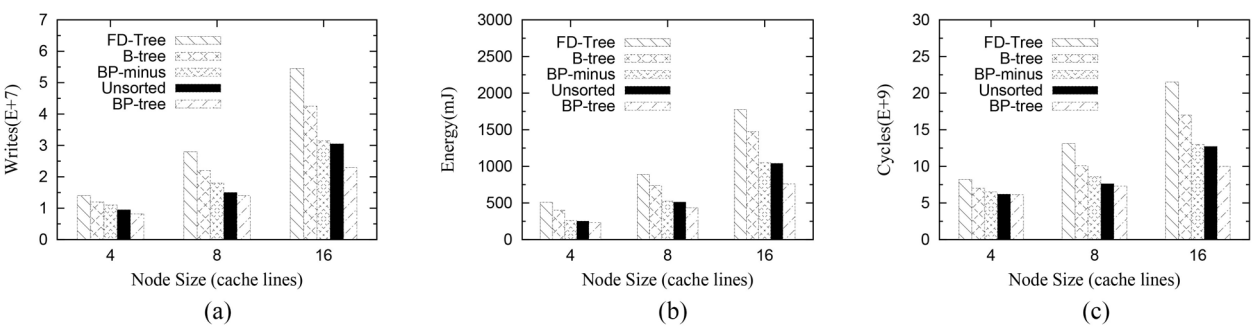


Fig. 13. TPC-C trace. (a) Writes. (b) Energy. (c) Cycles.

7.2.6 Scalability Analysis

In this section, we use the real TPC-C trace to evaluate B^p-tree. Besides, in order to fully prove that our proposed indexes can be excellently suited to PCM, we have added the FD-Tree index experiments to make comparison. Because the FD-Tree is almost the best representative index which adopts the batch or lazy idea to convert the random writes into sequential ones which has been proved in [11]. The experiments is depicted in Fig. 13. However, the FD-Tree shows the worst performance in the PCM. Because the core idea of FD-Tree is to convert the small random writes into sequential ones which is quite appropriate for the disk or SSD. However, the index has to incur large numbers of write operations. In FD-Tree, the update entry has to be copied from the top level to the lowest level and produce many unnecessary writes. For PCM, the random and sequential writes have few performance differences then this idea is not helpful.

8 Conclusion

In this paper, we proposed the B^p-tree index for the nonvolatile memory PCM. The major design objective of our B^p-tree is to reduce the number of writes and energy consumption while keeping the tree construction and search efficiency. We developed a predictive model to predict future data distribution based on the current data access and pre-allocate space to reduce the possible writes caused by node splits or merges. We presented the model and show some metrics to evaluate the performance of our model. The experiments on PostgreSQL database system showed that our B^p-tree indexing scheme achieves better performance than the traditional B^+-tree and outperforms the state-of-the-art solutions. Additionally, our B^p-tree can be easily implemented in the current commercial database with minor revisions.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China under Grant 61272090 and Grant 61373024, National Grand Fundamental Research 973 Program of China under Grant 2011CB302206, Beijing Higher Education Young Elite Teacher Project under Grant YETP0105, a project of Tsinghua University under Grant 20111081073, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, and the "NExT Research Center" funded by MDA, Singapore, under Grant WBS:R-252-300-001-490.

References

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, "Lazy-adaptive tree: An optimized index structure for flash devices," *Proc. VLDB*, vol. 2, no. 1, pp. 361–372, 2009.

[2] L. Arge, "The buffer tree: A new technique for optimal I/O-algorithms," in *Proc. WADS*, Kingston, ON, Canada, 1995, pp. 334–345.

[3] F. Bedeschi *et al.*, "A multi-level-cell bipolar-selected phase-change memory," in *Proc. IEEE ISSCC 2008 Dig. Tech. Papers*, San Francisco, CA, USA, pp. 428–625.

[4] S. Chen, P. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proc. CIDR*, Asilomar, CA, USA, 2011.

[5] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal prefetching B+-trees: Optimizing both cache and disk performance," in *Proc. 2002 ACM SIGMOD Int. Conf. Management Data*, New York, NY, USA, pp. 157–168.

[6] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[7] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Main memory indexing: The case for BD-tree," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 7, pp. 870–874, Jul. 2004.

[8] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Indexing high-dimensional data for efficient in-memory similarity search," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 3, pp. 339–353, Mar. 2005.

[9] G. Graefe, "Write-optimized B-trees," in *Proc. VLDB*, Toronto, ON, Canada, 2004, pp. 672–683.

[10] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. 36th Annu. ISCA*, Austin, TX, USA, 2009, pp. 2–13.

[11] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proc. VLDB*, vol. 3, no. 1, pp. 1195–1206, 2010.

[12] S. Mitra, W. W. Hsu, and M. Winslett, "Trustworthy keyword search for regulatory-compliant record retention," in *Proc. VLDB*, Seoul, Korea, 2006, pp. 1001–1012.

[13] G. Muller, N. Nagel, C. Pinnow, and T. Rohr, "Emerging non-volatile memory technologies," in *Proc. 29th ESSCIRC*, Estoril, Portugal, 2003, pp. 37–44.

[14] A. Nadembega, T. Taleb, and A. Hafid, "A destination prediction model based on historical data, contextual knowledge and spatial conceptual maps," in *Proc. IEEE ICC*, Ottawa, ON, Canada, 2012, pp. 1416–1420.

[15] S. Nath and A. Kansal, "Flashdb: Dynamic self-tuning database for NAND flash," in *Proc. 6th Int. Conf. IPSN* Cambridge, MA, USA, 2007, pp. 410–419.

[16] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inform.*, vol. 33, no. 4, pp. 351–385, 1996.

[17] J. Pei, M. K. M. Lau, and P. S. Yu, "TS-trees: A non-alterable search tree index for trustworthy databases on write-once-read-many (WORM) storage," in *Proc. AINA*, Niagara Falls, ON, Canada, 2007, pp. 54–61.

[18] M. K. Qureshi *et al.*, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. MICRO*, New York, NY, USA, 2009, pp. 14–23.

[19] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. ISCA*, Austin, TX, USA, 2009, pp. 24–33.

[20] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. ICS*, 2011, pp. 85–95.

[21] J. Rao and K. A. Ross, "Making B+- Trees cache conscious in main memory," in *Proc. 2000 ACM SIGMOD Int. Conf. Management Data*, New York, NY, USA, pp. 475–486.

[22] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, no. 4.5, pp. 465–479, 2008.

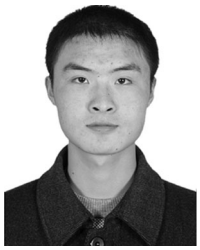
[23] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *Proc. 13th ACM SOSP*, 1991, pp. 1–15.

[24] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Proc. 37th Annu. ISCA*, Saint-Malo, France, 2010, pp. 141–152.

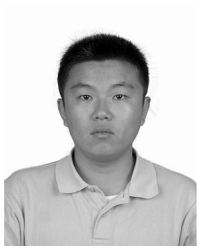
[25] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *Proc. 37th Annu. ISCA*, 2010, pp. 383–394.

[26] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An efficient B-tree layer implementation for flash-memory storage systems," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, Article 19, Jul. 2007.

[27] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. 36th Annu. ISCA*, 2009, pp. 14–23.



Weiwei Hu is a master's student at the National University of Singapore, Singapore. He received the bachelor's degree in computer engineering from Tsinghua University, Beijing, China, in 2010. His current research interests include indexing and query processing algorithms design of database systems.



Guoliang Li received the Ph.D. degree in computer science from Tsinghua University, Beijing, China, in 2009. Currently, he is an associate professor with the Department of Computer Science, Tsinghua University. His current research interests include data cleaning and integration, spatial databases, and crowdsourcing. He is a member of the IEEE.

Jiacai Ni is currently a Ph.D. candidate with the Department of Computer Science, Tsinghua University, Beijing, China. His current research interests include multi-tenant data management, schema mapping, and index techniques for new hardware database.

Dalie Sun is an associate professor with the Harbin Institute of Technology (HIT), Harbin, China. He received the bachelor's degree in mathematics from Northeast Normal University, Changchun, China, in 1987, the master's degree in computer application from HIT in 1996, and the Ph.D. degree in computer theory and software from HIT in 2012. His current research interests include peer-to-peer computing and distributed query, databases, and data mining.

Kian-Lee Tan received the B.Sc. (Hons.) and Ph.D. degrees in computer science from the National University of Singapore, Singapore, in 1989 and 1994, respectively. Currently, he is a professor with the Department of Computer Science, National University of Singapore. His current research interests include query processing and optimization, database security, and database performance. He is a member of the IEEE.