Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A B-Tree index extension to enhance response time and the life cycle of flash memory

Hongchan Roh, Woo-Cheol Kim, Seungwoo Kim, Sanghyun Park *

Department of Computer Science, Yonsei University, South Korea

A R T I C L E I N F O

Article history:
Received 8 November 2007
Received in revised form 15 January 2009
Accepted 4 May 2009

Keywords:
B-Tree index extension
Flash memory life cycle
Flash memory response time
Write count reducing
BFTL
MB-Tree

A B S T R A C T

Flash memory has critical drawbacks such as long latency of its write operation and a short life cycle. In order to overcome these limitations, the number of write operations to flash memory devices needs to be minimized. The B-Tree index structure, which is a popular hard disk based index structure, requires an excessive number of write operations when updating it to flash memory. To address this, it was proposed that another layer that emulates a B-Tree be placed between the flash memory and B-Tree indexes. This approach succeeded in reducing the write operation count, but it greatly increased search time and main memory usage. This paper proposes a B-Tree index extension that reduces both the write count and search time with limited main memory usage. First, we designed a buffer that accumulates update requests per leaf node and then simultaneously processes the update requests of the leaf node carrying the largest number of requests. Second, a type of header information was written on each leaf node. Finally, we made the index automatically control each leaf node size. Through experiments, the proposed index structure resulted in a significantly lower write count and a greatly decreased search time with less main memory usage, than placing a layer that emulates a B-Tree.

1. Introduction

Recently, flash memory, which has advantages in the mobile environment, has become more popular as the usage of mobile devices has increased. Flash memory is suitable for the mobile environment because it is non-volatile like hard disk drives, and more shock-resistant and efficient in energy consumption than hard disk drives.

However, flash memory has several disadvantages such as long write latency, a short life cycle, and impossible in-place updates (which means that overwriting as a unit of flash memory page is infeasible). A page is the smallest unit of input and output, and several contiguous pages compose a block (in this paper, a page means a flash-memory page). Table 1 indicates that writing a page is approximately 5.5 times slower than reading a page in the case of the MLC NAND type flash memory. Another disadvantage is that flash memory devices have to erase a block to overwrite even a single page. Thus, in order to rewrite a page, most flash memory storage systems do not erase the entire block but rewrite the updated data to a free page of another block. Finally, the maximum erase count of each block is limited to about 10,000–1,000,000 operations [10]. If the erase count of a block exceeds this threshold, flash memory can no longer store data on the block.

The B-Tree [2], one of the most popular indexes based on hard disk drives, focuses on minimizing the access count to a hard disk. However, a B-Tree on flash memory without any conversion causes a greater number of write operations than a B-Tree on a hard disk. Consequently, the response time to user requests on the index increases because of the long write latency of flash memory, and its life cycle is shortened by frequent erase operations caused by numerous write operations.

* Corresponding author. Tel.: +82 2 2123 7757; fax: +82 2 365 2579.
E-mail address: sanghyun@cs.yonsei.ac.kr (S. Park).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.05.007

Table 1
The characteristics of recent NAND flash memory (SLC small block NAND [33], SLC large block NAND [15], MLC NAND [15]).

Features	SLC small block NAND	SLC large block NAND	MLC NAND
Page size	512 B	2 KB	4 KB
Block size	16 KB	128 KB	512 KB
Page-read latency	348 µs	78 µs	166 µs
Page-write latency	909 µs	253 µs	906 µs
Block erase latency	1.9 ms	1.5 ms	1.5 ms

To address this problem, Wu et. al. proposed an efficient B-Tree layer for Flash memory Translation Layer (BFTL) [32] in 2003 and later they updated their findings with more experimental results in [33]. Emulating a B-Tree, the BFTL reduced the overall write count on flash memory. Even though the BFTL succeeded in reducing the write count, the layer inserted between a B-Tree and a flash memory file system caused the BFTL to occupy a large amount of main memory space and increased search time as well. Later, flashDB [26] extended the BFTL's idea so that the index can be self-tuned. By doing so, the flashDB efficiently reduced the excessively increased search time caused by the BFTL. However, the flashDB still has a problem: The tradeoff between the write count and search time is not efficient.

The goal of this paper is to design a B-Tree extension that reduces not only the overall write count to flash memory in order to respond faster and have a longer life cycle, but also decreases the main memory usage and search time. From now on, we will call the index structure an MB-Tree (Modified B-Tree).

The MB-Tree consists of a Batch Processing Buffer (BPB) and Leaf Node Header (LNH) that reduce the write count and decrease the search time, respectively. Furthermore, the leaf node size of the MB-Tree is automatically controlled.

This paper is organized as follows: The second section surveys related work. In the third section, we define the problem as well as explain our index design principles and strategies. To ascertain these design strategies, we describe the overall structures of the MB-Tree, BPB, and LNH and algorithms for those structures in the fourth section. We discuss several MB-Tree related issues in the fifth section. The sixth section reports the experimental results when the criteria of the overall page-read count and page-write count on flash memory were applied to the B-Tree, BFTL, MB-Tree, and flashDB. In the last section, the conclusion is presented.

2. Related work

Until recently, efforts to overcome the drawbacks of flash memory have focused mainly on developing an efficient file system [1,3,5,7,13,14,16,18,21,22,30,31]. Also, a few studies [15,26,32–34], with the goal of adapting a B-Tree and R-Tree [12] to flash memory, have been performed regarding efficient index structures on flash memory.

2.1. File systems for flash memory

Recently, several file systems for flash memory have been proposed and a few additional studies derived. The file systems are generally classified in two types: block-mapping techniques and flash-specific file systems [10].

Block-mapping techniques have an advantage, because application programs based on block device file systems can be directly applied to flash memory since these techniques insert a layer that emulates a block device. This methodology includes FTL [14], CompactFlash [7], SmartMedia [30], and other file systems.

Conversely, flash-specific file systems, based on log-structured file systems [29], can substitute the legacy file systems for flash memory and have shorter write latency. Their examples include JFFS2 [31], JFFS3 [3], LFM [13], YAFFS [1], FAST [21], and CFFS [22]. Among these file systems, JFFS3 is notable due to its state of the art architecture as a log-structured file system and adopted B+-Tree as the basic storage structure. JFFS3 improved the original design of JFFS2 by supporting more massive storage. JFFS2 is problematic, however, because every log node has to be loaded to main memory when the file system is mounted. Therefore, the more data were stored on the file system, the more main memory was required. In order to solve this problem, JFFS3 adopted the B+-Tree structure for storing the file system objects to storage devices, which is called the JFFS3 tree. JFFS3 needs less flash memory space than JFFS2, since the file system objects are stored as the leaf nodes of the JFFS3 tree on flash memory. However, this JFFS3 tree structure has the so-called wandering tree problem, which originates from the B-Trees' feature such that a child node's update can be propagated to its parent node. This problem becomes worse on flash memory, since an update of the child node essentially requires an update of its parent node, which is propagated up to the root node because the flash memory has the nature of impossible in-place updates. To address this, JFFS3 employed the journaling idea, dividing its storage area into the journal space and the space storing the JFFS3 tree. In the journal space, the file system changes are logged instead of directly updating the leaf nodes of the JFFS3 tree. For faster access, JFFS3 adopted a main memory based index structure called the journal tree, which summarizes the information of the file system objects in the journal area and indicates how to access for the given query between the journal and the JFFS3 tree area.

These new file systems handle garbage collection and wear leveling. Garbage collection chooses and erases a certain block in order to make the pages of blocks that have invalid data empty. Wear leveling distributes erase operations to flash memory in order to prevent its erase count from exceeding the maximum threshold on each block. To handle these procedures, various policies have been suggested [5,16,18]. However, these procedures cause another write cost (in this paper, 'cost' is used in terms of time) while copying valid data from the pages to other pages.

In this regard, write operations to flash memory cause not only heavy overhead such as the page-write latency but also an additional cost such as copying valid data to other pages through garbage collection and wear leveling.

2.2. B-Tree indexes on flash memory

B-Tree indexes [2] and its several variants, including B+-Tree [6] and B*-Tree [20], cause excessive write operations on flash memory since they write one leaf node per insert operation to flash memory. Moreover, just one insert operation necessitates an update of all the pages allocated for a leaf node in the worst case because the previous entries of the leaf node have to be interleaved with the new entry in order to maintain the sorted order between leaf node entries.

The B-Tree index structure was first proposed in 1972. It was later extended to the B*-Tree in 1973 and to the B+-Tree in 1979, with many other studies ensuing. The B+-Tree is a well known B-Tree variant whose leaf nodes are linked to each other in order to enhance search performance especially for range queries. The B*-Tree is a B-Tree variant in which each node is at least 2/3 full (instead of just 1/2 full). B*-tree insertion employs a local redistribution scheme to delay splitting until 2 sibling nodes are full. Then the 2 nodes are divided into 3. This scheme guarantees that storage utilization is at least 66%. Among the other studies of concurrency and recovery issues, in-depth studies have been performed [17,23–25]. Keller and Wiederhold [17] suggested a B-Tree variant that can ensure more concurrency with variable length data, and later, Lomet and Salzberg [24] extended their research to more generalized one. In these papers, they tried to reduce the propagated node updates by delaying parent nodes' splitting. Manolopoulos [25] proposed a B-Tree variant to simultaneously enhance its capability of concurrency and recovery. Lim and Kim [23] proposed a recent B-Tree variant for enhancing concurrency. In their research, they proposed a way to prevent any locks on the search path by backing up the previous consistent sate of the B-Tree. Rosenberg et al. suggested B-Tree extensions [27,28] that could maximize the storage rate while sacrificing search time. Recently, the μ-Tree [15] was proposed, which focused on the wandering problem reported by JFFS3. To handle the problem, the μ-Tree modified the B-Tree structure so that all the leaf and internal node entries that need to be read for a single search can be stored on the same single flash-memory page. This idea, however, cannot be generally utilized even though it is helpful to alleviate the propagated updates. This is due to the fact that the wandering problem is meaningless on file systems that apply the block mapping technique. Some of these related studies have mechanisms such as deferring parent nodes' splitting and resolving the wandering problem, which can reduce the write count on storage devices, but they effect only a small decrease in the write count on storage media as indirect methods in order to reduce the write count. However, the BFTL [32] proposed a way that not only minimizes the overall write count on storage media by adapting the ideas of log-structured file system, but also is applicable to any file system. Later, the flashDB [26] enhanced the BFTL's search time that had been excessively increased by the BFTL's own log-structured design, even though the flashDB's enhanced performance does not promise to dominate others'.

2.3. BFTL and flashDB

The BFTL emulates a B-Tree index whose nodes consist of a single page. On flash memory file systems, the BFTL places a new layer, which is composed of a Reservation Buffer (RB) and Node transition Table (NT).

The BFTL reduces the overall write count by gathering insert operations through the RB's buffering procedure and simultaneously flushing the entries of gathered insert operations to contiguous pages of flash memory. The NT logically binds the entries of a node, which are distributed over different pages through the RB's flush procedure, into the node. The NT maintains linked-lists, each of which is composed of the link of the page identification (ID) numbers where the entries of the node are stored. Fig. 1 shows a simple B-Tree that a BFTL emulates and the NT of the BFTL. For each node alphabetically expressed, the NT is represented by a linked-list consisting of the page ID numbers.

Accordingly, the BFTL can reduce the overall page-write count by emulating a B-Tree, using the RB and NT. However, the search time increases proportionately to the length of each linked list of the NT. This is because a longer linked-list causes

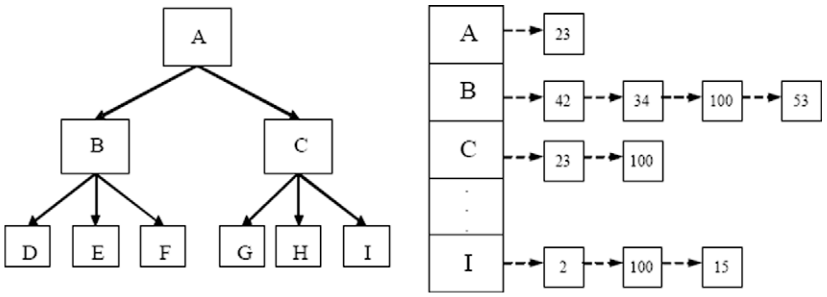

Fig. 1. A simplified B-Tree and its node transition table.

more page-read operations when search operations are executed. Additionally, the BFTL occupies excessive main memory proportionate to the linked-list length. To prevent the linked-list from becoming longer, the BFTL performs a compaction operation, which limits the linked-list length to a certain threshold (*C*). Specifically, if the length exceeds *C*, then the BFTL reads all pages that store the entries of the node and stores them on contiguous pages of flash memory. The BFTL creates a trade-off between the page-write count and search time by adjusting *C*.

The flashDB focused on addressing the substantial main memory occupancy and the increased search time of the BFTL. It came up with a self-tuning mechanism that converts each node to between the disk mode and log mode according to the variable calculating the cost and benefit from the mode stored in each node. In the disk mode, it operates as does an original B-Tree node, whereas it logs update-requests as does a BFTL node in the log mode. The variable accumulates the loss from maintaining the current mode since the last mode is switched. Therefore, if the number of search operations becomes greater than the number of insert operations, the node switches to the disk mode, and vice versa. Since the flashDBs do not need to read the linked list of the NT in the disk mode, the search time is enhanced. Furthermore, in the log mode, the logging scheme reduces the page-writes on flash memory. However, even though it successfully enhances the BFTL's performance, it still has the following problems: First, It is inevitable that the flashDB has a similar degradation problem, which is caused by the log-structured index record, to the BFTL. As specified in its paper, the log-structured index record has 4 additional attributes (*NodeID*, *LogType*, *SequenceNumber*, and *LogVersion*) except the basic attributes, key-value and pointer. In the paper, only a micro benchmark was conducted, which used at most 30,000 index records, so the attributes can be stored more compactly. However, in order to be widely used, the flashDBs have to consider more than 30,000 index records, and consequently, the attributes become to occupy larger space, resulting in obstructive bottleneck in the performance (This degradation is worse in the case of the BFTL). Second, the switching mechanism has several drawbacks. Because the variable counting the loss and benefit accumulates the current mode's benefits since the last switching, the node mode cannot be switched until the loss from the current mode dominates the accumulated benefit in the variable, even if the queries disadvantageous to the current mode continuously emerge. Therefore, immediately switching to the suitable mode for the current query pattern is not feasible, and thus the enhanced performance becomes insignificant compared with the BFTL. On the other hand, if the query pattern changes very fast, then the mode switching happens too frequently, so the cost caused by the mode switching increases.

3. Strategies and principles of an MB-Tree

To address the drawbacks of the B-Tree, BFTL, and flashDB on flash memory, the following design principles have to be satisfied, avoiding the log-structured index record design: First, an MB-Tree should reduce the overall write count of a B-Tree in order to enhance response time and the life cycle of flash memory. Second, the MB-Tree should be able to operate with limited main memory space regardless of input data amount. Third, the MB-Tree should perform a more efficient trade-off between the overall page-write count and search time than the BFTL and flashDB. In other words, the MB-Tree should be able to obtain a maximum reduction of the page-write count with a minimum sacrifice of search time.

In order to achieve the principles stated above, we established the following strategies.

First, in order to reduce the page-write count, after gathering as many entries that belong to the same leaf node as possible, an MB-Tree simultaneously writes them to flash memory. In this manner, the MB-Tree can reduce the page-write count compared with B-Trees that write entries individually.

Second, in order to reduce the page-write count, the MB-Tree contiguously stores a set of entries regardless of their key order when inserting new entries to a leaf node. This means that the leaf node entries of an MB-Tree are not sorted. The B-Trees' original way to handle insert operations, such that the previously existing entries of leaf nodes are interleaved with the newly inserted entries in order to be sorted together with the new entries, causes updates to almost all the leaf node space (in the worst case, all the pages allocated for the leaf node have to be updated per insert operation). However, if an MB-Tree contiguously stores the inserted entries in any empty space of a leaf node without considering the key order between the previous entries and the new entries, then the entries occupy a smaller space and cause less page-writes than that of the B-Trees' original way.

Third, in order to limit main memory usage, the MB-Tree stores both the entries and logical structure information on flash memory except for temporarily buffered input data, in contrast with the BFTL, which stores the entries on flash memory but stores the logical structure information on main memory.

Finally, for an efficient trade-off between the page-write count and search time, the MB-Tree greatly reduces the page-write count with a small increase in search time, compared with the BFTL that reduces the page-write count while increasing search time multiple times that of the B-Tree's.

4. Index structures and related algorithms of an MB-Tree

4.1. MB-Tree overview

Fig. 2 shows an overview of the MB-Tree designed by utilizing the aforementioned strategies. The MB-Tree consists of a Batch Process Buffer (BPB), a root node, internal nodes, leaf nodes, and Leaf Node Headers (LNHs). A BPB is maintained on

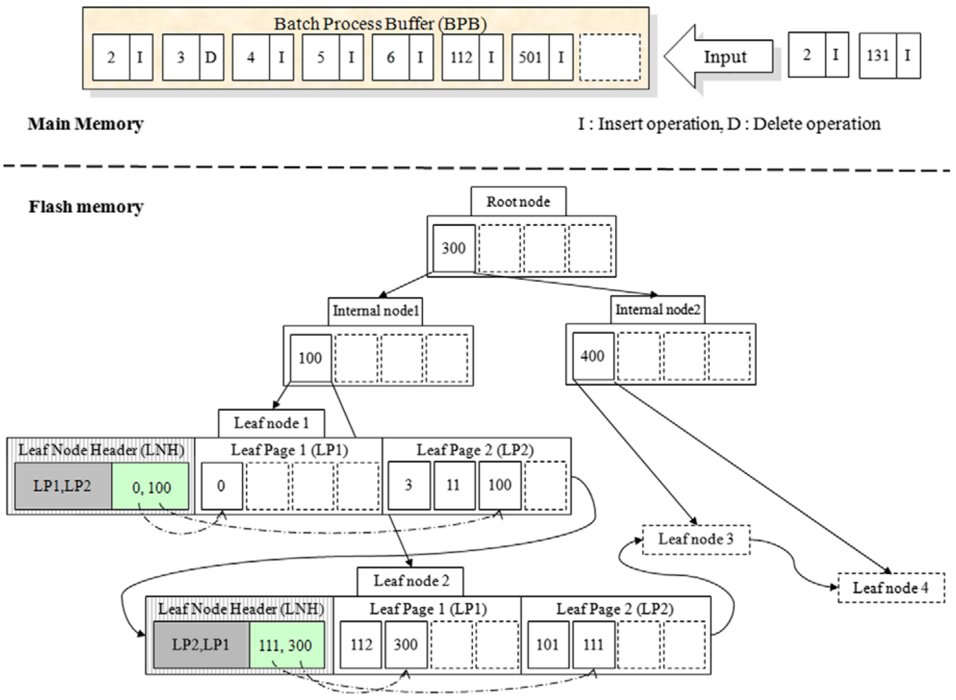

Fig. 2. An MB-Tree overview.

main memory, and the others are stored on flash memory. Though the root node and internal nodes are organized similarly to those of a B-Tree, the leaf nodes are differently organized. A leaf node consists of an LNH and a set of leaf pages. The leaf node is headed by an LNH, and Leaf Pages (LPs) contiguously follow the LNH. An LNH occupies a single page and stores the order information of the entries' keys on the leaf node. Each LP occupies a single page and is organized similarly to the B-Tree's leaf node. The maximum number of entries that can be stored on an LP is determined by dividing the size of a page by the size of an entry.

The purpose of the BPB is to reduce the overall page-write count. For this purpose, the entries of insert operations are processed as follows: The entry of each insert operation is inserted into the BPB until the BPB completely fills its allocated main memory space. If there is no space left to store another entry in the BPB, the BPB classifies the inserted entries on the basis of which leaf node each entry belongs to. Next, the BPB chooses the leaf node that has the most entries and then inserts the entries into the chosen leaf node. Finally, the updated leaf node is written to flash memory. In this manner, the MB-Tree writes a leaf node only once while processing a set of insert operations, in contrast with B-Trees that write a leaf node on flash memory for every insert operation.

In a further attempt to reduce the page-write count, the MB-Tree contiguously writes the entries on the remaining free space of the chosen leaf node regardless of the key-value order of already existing entries. This process can reduce the page-write count because new entries are not interleaved with existing ones, and thus the MB-Tree's leaf node updates less space than the B-Tree's. However, this approach causes disorder among the keys of the entries so that a search operation within a leaf node cannot execute a binary search but needs to check all the entries of the leaf node. To address this, the MB-Tree adds an LNH to the leaf node structure, which contains the entries' order information on each leaf node. In this way, the LNH structure makes the MB-Tree search the target entry faster by providing the order information, which enables a binary search.

In order to minimize the usage of main memory space, the MB-Tree stores all entries and information about logical structures among the nodes on flash memory. The MB-Tree achieves an efficient trade-off between the page-write count and search time through a reduced write count by the BPB as well as enhanced search time by the LNH.

4.2. Structure of a BPB and input–output operations

4.2.1. Structure of a BPB

User requests to an index can be classified into insert, update, or delete operations. To store the classification information of those operations in a BPB, we define a Task Entry (TE) as a data structure consisting of the following:

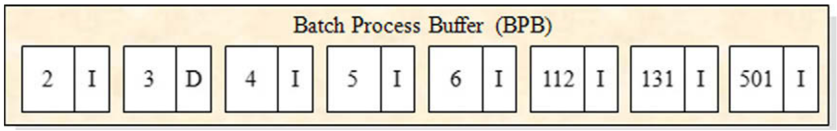

Fig. 3. The status of the BPB after inserting new TEs from Fig. 2.

- Entry (*E*): consists of a key-value and a pointer to a data record that has the key-value.
- Flag (*F*): represents the operation type. 'I' represents an insert operation, 'D' represents a delete operation, and update operations have no symbol because they can be denoted as a sequence of 'D' and 'I'.

In Fig. 2, for example, the first grid represents a TE with an *E* value of '2' and an *F* value of 'I'. This indicates that an insert operation with a key-value of 2 has been requested.

The explanation regarding the BPB in Section 4.1 was stated in order to simplify the concept of the BPB. When delete or update operations, in addition to insert operations, are requested to an MB-Tree, the MB-Tree processes these operations in the same way as insert operations because these operations most likely cause page-write operations. These operations are stored as TEs in the BPB and if the BPB is completely filled, then the MB-Tree simultaneously processes the operations.

4.2.2. BPB's input processing

Whenever users request an operation, the MB-Tree composes a TE and stores it in the BPB of the MB-Tree. As TEs are entered into the BPB, the MB-Tree examines them for eliminating redundancies in the following ways: If a TE is identical to an already existing TE, the BPB does not store it. Additionally, if a new TE with an *F* value of 'D' and the same *E* value as an already existing TE is entered, the BPB does not store the new TE and eliminates the existing TE.

After the rules written above are applied, TEs are sorted by their key-values first and creation times next in increasing order. Then they are stored in the BPB.

Insert, delete, or update operations to internal nodes are not created as TEs. The BPB maintains only TEs created by user requested operations (insert, delete, or update operations to leaf nodes).

Fig. 2 shows two new TEs about to be inserted when seven TEs are already stored in the BPB. Fig. 3 represents the BPB after the TEs have been processed according to the rules defined above.

4.2.3. BPB's output processing

If a BPB is completely filled, it selects a certain set of TEs and executes their operations. We define this as a flush operation, which consists of the following three phases:

First, the MB-Tree classifies all TEs in the BPB, based on the criterion of which leaf node the keys of the TEs belong to. Task Entry Set (TES), denotes a set of the classified TEs per leaf node.

Second, the MB-Tree chooses the TES that has the most elements as the Final Task Entry Set (FTES).

Third, the MB-Tree performs each operation of the FTES on the leaf node where the FTES belongs, with delete operations preceding insert operations.

Finally, the MB-Tree eliminates the TEs that belonged to the FTES, making room for new user requests.

In the above explanation, it is described that the BPB' flush operation obtains the largest TES as a FTES. This is, however, an ideal case. Locating the largest TES can cause additional overhead to the MB-Tree such as consuming excessive main memory space and another search process to locate boundary values of leaf nodes. The next section discusses and describes an efficient way to retrieve the FTES.

4.2.4. Process for obtaining FTES

To perform the first phase of the flush operation, the BPB needs to know where each TE belongs. In order to accomplish this, the BPB must know the lower and upper boundary values of each leaf node, which consist of the minimum and maximum key-value that can be inserted into the leaf node.

Two brute-force approaches can determine the boundary values. The first one searches all the internal nodes directly above leaf nodes and then extracts the boundary values of each leaf-node by using the entries of the corresponding internal node. This approach causes as much search cost as the cost of searching all paths from the root to the internal nodes directly above the leaf nodes, whenever a BPB performs the flush operation. The other approach is to maintain all the boundary values of all leaf nodes constantly. In this approach, the boundary values are updated and maintained on main memory whenever a new leaf node is created. This approach causes no search cost but main memory usage increases as the number of leaf nodes increases.

To address these problems, we designed an algorithm called Unique Path Search (UPS), which searches a single pass from the root to a certain leaf node, obtaining a sub-optimal FTES by using only main memory space that is necessary for a single path search. This search method is represented in Algorithm 1.

Algorithm 1. The UPS algorithm

```
UPS(task entries) searches a single pass to obtain a sub-optimal FTES. $Visited_i$, $Child_{i,j}$, $TES_{i,j}$, and $Largest_i$ denotes the visited
  node on level $i$, $j$th child node of the visited node, the TES of the child node, and the largest TES among the TESs of the
  child nodes, respectively. Each task entry is represented as a data structure $\alpha$, and its key-value is represented as $\alpha.key$.
  The number of TEs in a TES, tree height, and the lower and upper boundary values of a node are denoted as $|TES|$, $H$,
  $Lower$ and $Upper$, respectively.
1. (Initialize.) This search begins by visiting the root node. Then, the UPS selects the node that has the largest TES among
   the child nodes of the root node.
   (a) Set $Visited_0 \leftarrow$ the root node.
   (b) For each $Child_{0,j}$, which is a child node of the root node, set $TES_{0,j} \leftarrow \{\alpha|\alpha \in BPB$, and $Lower$ of $Child_{0,j} \leqslant \alpha.key \leqslant Upper$
       of $Child_{0,j}\}$
   (c) Locate the child node that has the maximum number of $\alpha$ in its TES.
   (d) Set $Visited_1 \leftarrow$ the child node, $Largest_0 \leftarrow$ its TES.
2. (Explore internal nodes, narrowing down the FTES.) While visiting a node level by level, the UPS retrieves the boundary
   values of the visited node's child nodes and obtains their TESs by using the boundary values. Next, the UPS chooses the
   child node that has the largest TES among the obtained TESs as the next node to be visited and proceeds to the child
   node. This process ends when the child node of the visited node becomes a leaf node.
   (a) Repeat the following from level 1 ($i = 1$) to level $H - 1$ ($i = H - 1$):
           For each $Child_{i,j}$, which is a child node of $Visited_i$,
               set $TES_{i,j} \leftarrow \{\alpha|\alpha \in Largest_{i-1}$, and $Lower$ of $Child_{i,j} \leqslant \alpha.key \leqslant Upper$ of $Child_{i,j}\}$
           Locate the child node that has the maximum number of $\alpha$ in its TES.
           Set $Visited_{i+1} \leftarrow$ the child node, $Largest_i \leftarrow$ its TES.
   (b) Set FTES $\leftarrow Largest_{H-1}$, $Leaf \leftarrow Visited_H$.
3. (Execute task operations and update BPB.) Now that this process has obtained the FTES and the target leaf node, it fol-
   lows phases 3 and 4 in Section 4.2.3.
```

The UPS is a greedy approach that obtains a sub-optimal FTES through repeated selecting of the best child node, which has the largest TES. Even though the UPS cannot always obtain the optimal FTES, it is significantly more efficient than the aforementioned approaches because it searches only one path without the need of extra memory space to store the boundary values.

Fig. 4 represents the process the UPS performs in the case of Fig. 3. In Fig. 4a, the MB-Tree obtains the boundary values of the child nodes of the root node from the entries of the root node and then divides the TEs in the BPB into two groups of TEs, one with the key-values smaller than 300, and the other with the key-values greater than 300. Among these groups, the left group is larger than the right group, and thus the UPS chooses and traverses internal node 1. Next, Fig. 4b shows that the UPS obtains boundary values of the child nodes of internal node 1 from the entries of internal node 1 and establishes the TESs of the leaf nodes using the boundary values. Since the left TES in the BPB is the largest among the TESs, the UPS finally retrieves the left TES, which is the TES of leaf node 1, as the FTES.

4.3. Flushing process of the insert and delete task-entries to leaf nodes

This subsection describes in more detail how to perform each of the operations extracted from the FTES on the leaf node where the FTES belongs.

The MB-Tree first finds the leaf page where the operations have to be performed and then reads the leaf page into main memory. For a delete operation, the LNH helps the MB-Tree easily find the leaf page where the operation should be performed. This process of locating the leaf page for a delete operation is the same as the search operation that will be explained in Section 4.6. Next, the MB-Tree performs the operations on the leaf page and rewrites the updated leaf page on flash memory. The following paragraphs describe this process in more detail.

First, FTES delete operations are classified, and then the MB-Tree eliminates the entries that have the same key-values as the delete operations' from the leaf pages containing the entries to be deleted, after loading the leaf pages to main memory.

Second, the MB-Tree performs the insert operations remaining in the FTES. Here, the MB-Tree first confirms whether there is enough space on the leaf node. If the space is insufficient, the MB-Tree executes a split operation (to be explained later in Section 4.5). Otherwise, the MB-Tree arranges the new entries to be inserted on the leaf node by means of the best-fit placement policy [9]. More specifically, the MB-Tree finds a leaf page that has the nearest number of empty slots to the number of the entries to be inserted and inserts the entries into the leaf page. However, if some of the new entries remain after filling all the empty slots of the chosen leaf page, the MB-Tree finds another leaf page that has the nearest number of empty slots to the number of the remaining entries and inserts the entries into the leaf page. This process is repeated until no new entry remains.

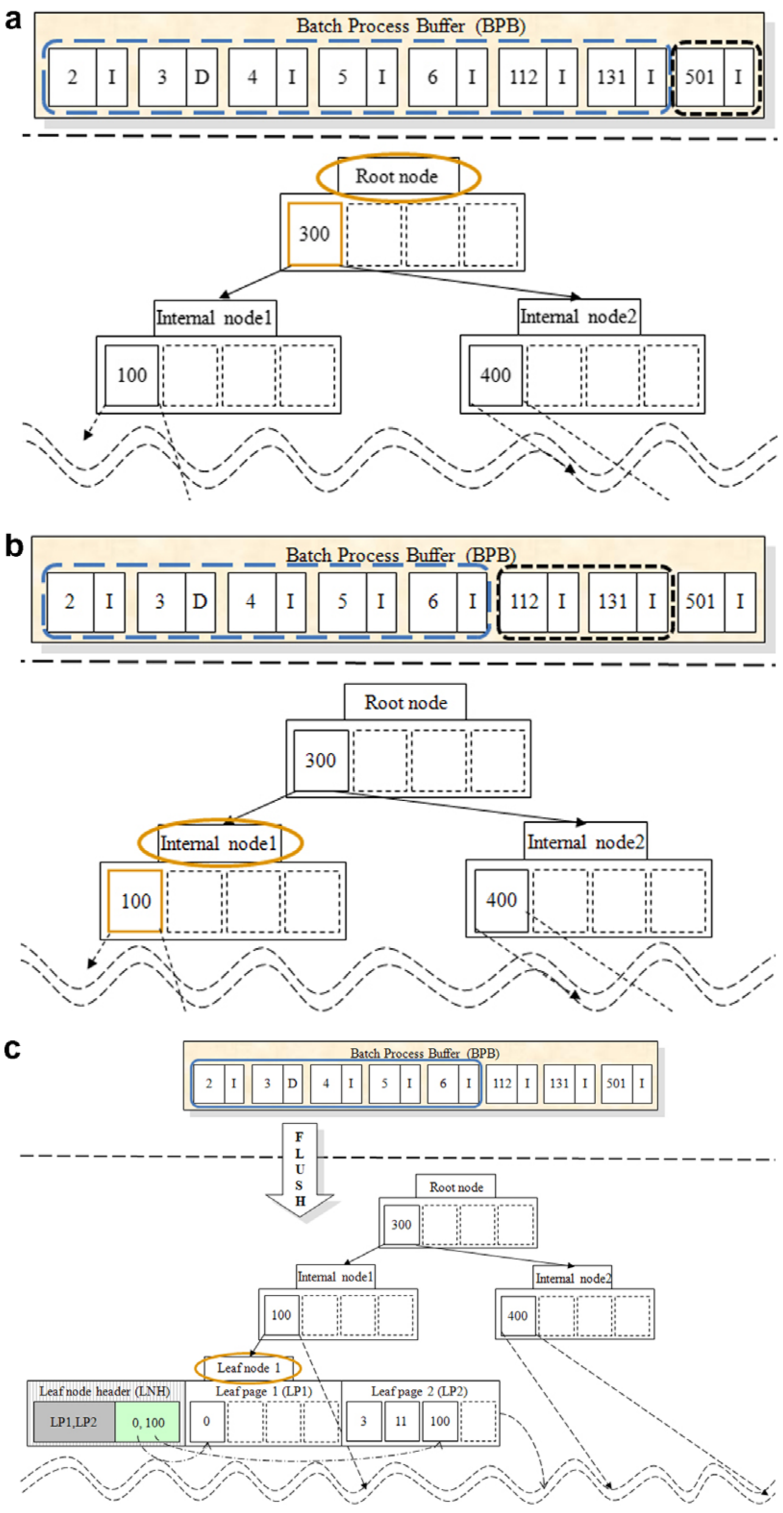

Fig. 4. (a) Step 1 of the BPB flush operation. (b) Step 2 of the BPB flush operation. (c) Step 3 of the BPB flush operation.

Finally, the MB-Tree proceeds to rewrite the updated leaf pages to flash memory. At this time, within each of the updated leaf pages, the MB-Tree locally sorts the entries of the leaf page according to the key-values. This can reduce the cost of later local searches within the leaf page (i.e., a binary search can be used).

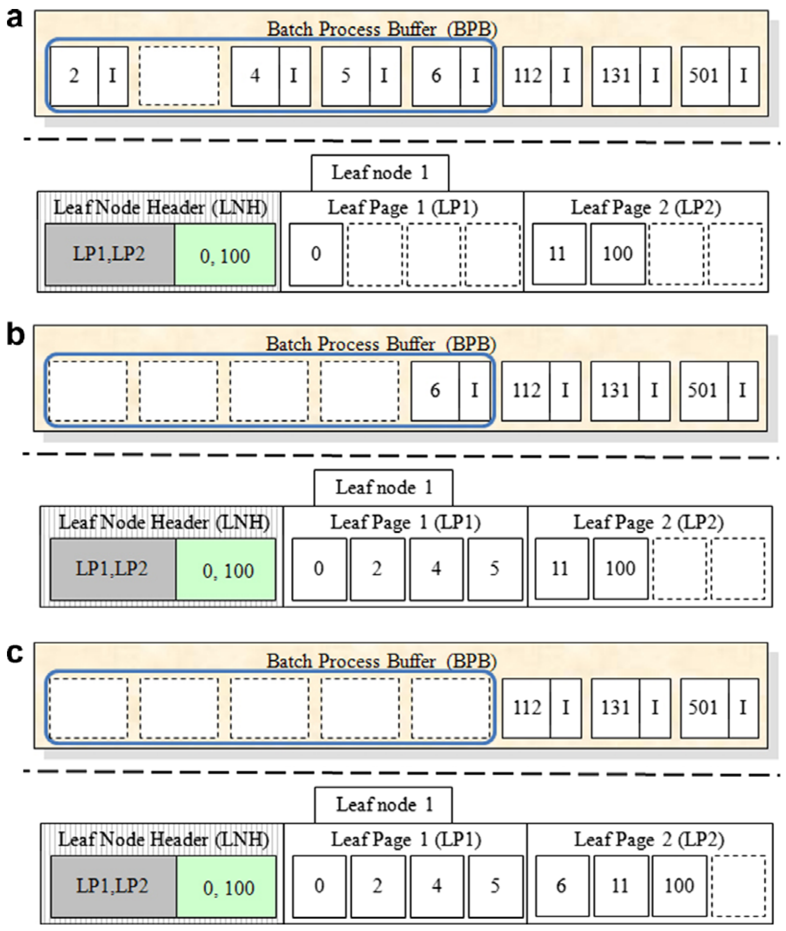

Fig. 5. (a) The process of a delete operation on a leaf node. (b) Step 1 of an insert operation. (c) Step 2 of an insert operation.

Fig. 5a explains the result when the MB-Tree executes the delete operation on the entry that had a key-value of 3 from the example in Fig. 4c. Fig. 5b and c represent the steps by which the MB-Tree performs the insert operations after the delete operation. In Fig. 5a, there are 4 entries to be inserted and 5 empty slots on leaf node 1, so the split operation is unnecessary. Since the number of empty slots in leaf page 1 (LP1) and leaf page 2 (LP2) are 3 and 2, respectively, and the number of new entries to be inserted is 4, LP1 is selected as the best fit. The new 3 entries are inserted into LP1. These results are shown in Fig. 5b. Similarly, to handle the remaining entries to be inserted, LP2 is selected, and then the new entry is inserted into LP2 as represented in Fig. 5c.

4.4. Structure and construction algorithm of an LNH

Once both the insert and delete operations are performed through the BPB flush operation, the LNH has to be reorganized. This subsection describes the structure and construction algorithm of an LNH.

4.4.1. Structure of an LNH

The LNH was designed to enhance search performance within a leaf node. Since the flushing process to leaf nodes writes entries into a leaf node regardless of the key-value order of already existing entries, in contrast with B-Trees, which maintain the sorted order of entries' key-values within each leaf node, the disorder between the entries' key-values makes a binary search impossible. Consequently, this degrades the search performance within each leaf node. To address this problem, an LNH maintains key-value order information between entries within a leaf node.

An LNH is frequently read from a leaf node because an LNH has to be read whenever a search operation is performed or a leaf node is updated. Consequently, unless the flash memory area allocated for storing an LNH is small, significant time will be devoted to reading LNHs. For this reason, we limited the flash memory area for storing an LNH to a single page.

In the LNH, each entry's order-information of the leaf node is represented as an ID number of a leaf page where each entry resides. Each entry's order-information ID (OID) is sorted by the key-value of each entry. To search faster, the key information value (KIV), the biggest key-value of a group of OIDs, is added to the LNH similarly to the internal node of a B-Tree. The

MB-Tree splits the OIDs into several intervals and records the biggest key-value of the entries in each interval as a KIV. As shown in Fig. 2, the structure of an LNH is separated into two parts, where the left side represents OIDs and the right side represents KIVs.

As seen on leaf node 2 in Fig. 2, the key-values of the entries on LP1 are greater than those on LP2. Therefore, in the OID region, the LP2 precedes LP1 to indicate that LP2's entries have smaller key-values than those of LP1. Unless there is an LNH for storing the order information of all the entries on each leaf node, the MB-Tree will waste search time finding an entry such as '101' because the MB-Tree needs to read both LP1 and LP2. In other words, a faster search algorithm cannot be applied to search the MB-Tree's leaf pages (LP1, LP2) when the entries on a leaf node are not sorted or an alternative type of order information for the sorting process does not exist.

4.4.2. Algorithms for constructing an LNH and searching within an LNH

Storing each OID and KIV compactly is important because it affects the performance of the MB-Tree. Since the MB-Tree limits the physical size of an LNH, the number of entries maintained by an LNH has to be limited as well. Every leaf node has to contain the same number of entries as the maximum number of OIDs that each LNH can contain. In this respect, if the MB-Tree stores more OIDs and KIVs, each leaf node can contain more entries. Having more entries improves the MB-Tree's performance, since it reduces the number of page-reads and page-writes caused by an insert operation or search operation, as will be shown in Section 4.9 (The number of entries within a leaf node is represented as *M* in Section 4.9).

Therefore, the ideas of compressing index structures are partially adaptively used in constructing an LNH (especially in making KIVs). Regarding research about compressing index structures, a recent study [8], which would reduce the space occupancy without significantly degrading the query performance, was proposed. This research can be regarded as a basic strategy in designing compressing algorithms of the indexes. However, we are concerned that if the compression is applied even to index records (the inserted leaf node entries), it may increase the search time because of the decompressing time. Therefore, we decided to adapt this index compressing technique only for handling the compaction for OIDs and KIVs.

The process for constructing an LNH with a compaction process is described in Algorithm 2. The MB-Tree brings all the entries within the leaf node after reading all the leaf pages of the corresponding leaf node to which the LNH belongs. Next, the MB-Tree executes Algorithm 2 by using the retrieved entries.

Algorithm 2. LNH construction algorithm

```
LNHMake(entries) writes OIDs and KIVs in the page allocated for the LNH, applying a compaction method. We denote a
  data structure (entry, leaf page ID) by $\beta$. The entry and leaf page ID within a $\beta$ is represented as $\beta.e$ and $\beta.id$, and the key
  and pointer value of the entry are denoted by $\beta.e.key$ and $\beta.e.ptr$, respectively. A page size in bytes, the number of leaf
  pages, key size of KIVs, and the largest and smallest key-value in the leaf node are denoted by PageSize, L, K, Large, and
  Small, respectively.
1. (Make an array of $\beta$s from the entries.) For each $entry_i$:
   (a) Set $\beta_i.e \leftarrow entry_i$, and $\beta_i.id \leftarrow$ ID of leaf page that contains $entry_i$.
2. (Sort the array.) The array of $\beta$s is sorted in the ascending order of $\beta.e.key$.
3. (Eliminate consecutive OIDs that are the same.) Consecutive OIDs that are the same are reduced to one, since writing
   only one for the OIDs sufficiently preserves the order information.
   (a) Set $OID_1 \leftarrow \beta_1$, and Set $Cnt \leftarrow 1$.
   (b) For each $\beta_i$, if $\beta_i.id$ is not equivalent to $\beta_{i-1}.id$, set $OID_{Cnt+1} \leftarrow \beta_i$, and increment Cnt by one.
4. (Write OIDs to the LNH.) Each OID is sequentially written as a bit-stream from the beginning area of the flash memory
   page allocated for the LNH.
   (a) The minimum number of bits needed for each OID is $\lceil \log_2 L \rceil$.
   (b) Sequentially write $OID_i.id$ from the beginning of the LNH, incrementing $i$ from 1 to Cnt.
5. (Decide the minimum key size of KIVs.) The key size of KIVs can be reduced using the locality of key-values. The keys
   are recoverable even if the difference-value between each key and the smallest key in the leaf node is written to the
   LNH with the smallest value written first.
   (a) Set $K \leftarrow \lceil \log_2(Large - Small) \rceil$ bits.
6. (Decide the OID interval for a KIV.) Since the KIVs should also be stored within the remaining space of the LNH, if the
   remaining space is insufficient for storing all the KIVs, then the MB-Tree increases the OID interval for a KIV until the
   space can contain both OIDs and KIVs.
   (a) The space needed for storing KIVs is $\lceil (Cnt \cdot K)/(IntervalSize \cdot 8) \rceil$ bytes.
   (b) The following constraint for the LNH to store OIDs and KIVs should be met.

       $\lceil Cnt \cdot \lceil \log_2 L \rceil / 8 \rceil + \lceil (Cnt \cdot K)/(IntervalSize \cdot 8) \rceil \leqslant PageSize$

   (c) Locate the minimum IntervalSize that satisfies the constraint.
7. (Write the KIVs within the remaining space.) Write the KIVs by using the determined interval.
   (a) Set $KIV_i \leftarrow OID_j.e.key$, where $j = IntervalSize \cdot i$, incrementing $i$ from 1 to $\lceil Cnt/IntervalSize \rceil$
   (b) Sequentially write each $KIV_i$ to the remaining space of the LNH
```

After LNH construction ends, the MB-Tree rewrites the new LNH to the flash memory.

When a search operation is given, the MB-Tree locates the entry to be searched within a leaf node by using the order and key information of an LNH as follows:

Algorithm 3. Search algorithm within a leaf node by using LNH

```
SearchWithinLeafNode(keyValue, IntervalSize) searches the leaf page that contains the entry whose key is equivalent to
  the given key-value. Each leaf node is represented as $LeafPage_i$ where $i$ denotes the ID number of the LeafPage.
 1. (Find an interval for the given key-value.) The interval is determined by comparing the given key-value with the key-
    values of the KIVs.
    (a) Set $IntervalFound \leftarrow i$, where $KIV_{i-1} < keyValue \leqslant KIV_i$ ($KIV_0$ is $-\infty$).
 2. (Read leaf pages and return an entry for this search.) The OIDs that belong to the found interval is first retrieved, and
    then leaf pages corresponding to the found OIDs are read. Finally it returns the entry having the same key-value as
    keyValue.
    (a) Set $Found \leftarrow \{OID_i | (IntervalFound-1) \cdot IntervalSize < i \leqslant IntervalFound \cdot IntervalSize\}$
    (b) For each $Found_i$, read $LeafPage_x$ where $x$ is $Found_i$, and if there is an entry having the same key-value as keyValue, in
        the leaf page, return the entry.
```

In conclusion, these techniques make the search faster because, in an OID interval for a KIV, there generally would be fewer leaf pages than all the leaf pages on a leaf node. In addition, this search algorithm still has room for improvement. For example, if a binary search can be applied when reading OIDs within the found interval because they are sorted in ascending order of their key-values, it can further reduce the number of pages to be read.

4.5. Split operation of leaf nodes

When no more entries can be inserted into a leaf node, the MB-Tree splits the leaf node into two leaf nodes (the original one and a newly created one), sorts the entries in increasing order of their key values, divides the entries evenly into two groups and stores each group on each leaf node, similarly to a B-Tree.

In contrast to a B-Tree, the MB-Tree automatically adapts each leaf node size (the number of leaf pages that composes a leaf node) according to what degree the OIDs and KIVs of each leaf node are stored compactly. Specifically, if an LNH is organized densely through Algorithm 2, the MB-Tree makes the leaf node larger, and the MB-Tree otherwise makes the leaf node smaller.

The leaf node size is determined by what degree OIDs and KIVs are compacted. In other words, if the same OID contiguously emerges in OIDs, OIDs can be compactly stored in the LNH and therefore more entries can be inserted into this leaf node, making the leaf node larger. Conversely, if the OIDs are not clustered to a large degree, then the MB-Tree decreases the leaf node size, since a small compaction rate is anticipated.

Another important criterion to control the leaf node size is that the interval size of the KIVs should be one. The MB-Tree controls the leaf node size in order to prevent search time from increasing. This is because if the MB-Tree increases the leaf node size, entries to be stored will increase as well. Consequently, the OIDs will occupy more storage area of the LNH and the LNH will lack the storage area to store KIVs. This causes a lengthened OID interval for a KIV. Therefore, the search cost within the leaf node will increase because more leaf pages need to be read. If the MB-Tree controls the leaf node size to allow the interval to be one, then the search process will read a single leaf page.

The MB-Tree does not control the leaf node size in real time but controls it when a leaf node is split in order to reduce the update cost of leaf nodes. The steps to control the leaf node size are as follows: First, before new entries are inserted, if the space usage of the leaf node is greater than 70% and the interval of KIVs is 1, then the MB-Tree increases the leaf node size by 1 (adds another leaf page) and creates a new leaf node of that size. Second, if the interval is larger than 1, the MB-Tree decreases the leaf node size by 1 regardless of the space usage rate and applies the increased leaf node size to the new leaf node created for this split process.

In addition, the reason why the boundary condition is set at 70% is to prevent the leaf node size from increasing excessively. Without such a limit, a leaf node can be split even at the rate of 50% and become larger when the FTES is large.

In this way, the MB-Tree can make each leaf node size converge into the appropriate size.

4.6. Overall algorithms of insert, delete, update, and search operations

Even though several partial algorithms including flushing the BPB, constructing an LNH, and search within a leaf node are given above, comprehensive algorithms have not been presented yet.

The comprehensive algorithm of an insert, delete, and update operation is described in Algorithm 4.

Algorithm 4. The comprehensive algorithm for an insert, delete, and update operation

```
OverallUpdate(TE) explains the comprehensive process from when a task entry is first inserted into the BPB to when the
  task entry is executed on a leaf node.
1. (Generate TEs.) Execute the following process on the basis of the type of the requested operation.
   (a) If an insert or delete operation is given, then a search with the key-value of the operation is conducted on the BPB
       and if a TE having the same key-value is located, then the redundancy-elimination process described in Section
       4.2.2 is executed.
   (b) If an update operation is given, the operation is divided into two operations, a delete operation and an insert oper-
       ation having the same key-value and record pointer. Next, the two operations generate the two TEs. The TE made by
       the delete operation is first inserted to the BPB, followed by the TE made by the insert operation. For each operation,
       the same search and redundancy-elimination process as stated above in step (a) is executed.
2. (Insert TEs into the BPB.) If the delete or insert operation is canceled by the redundancy-elimination process in phase 1,
   this process terminates. Otherwise, the TEs are inserted into the BPB.
   (a) If the BPB's capacity is enough to store the TE, TEs are stored and this process terminates.
   (b) Otherwise, Algorithm 1 is executed
3. (Execute each operation of FTES to the leaf node.) All the operations of the FTES including the given operation are exe-
   cuted to the chosen leaf node with Algorithm 1.
   (a) When the insert operations of the FTES are conducted, it places each entry following the best-fit placement policy
       described in Section 4.3.
4. (Reconstruct the leaf node's LNH.) After all the operations of the FTES are executed on the leaf node, then the leaf node's
   LNH is updated by Algorithm 2.
5. (Rewrite the leaf node on the flash memory.) The LNH and leaf pages that are updated through phases 1 to 4 are rewrit-
   ten to the flash memory allocated for them.
```

The comprehensive algorithm of a search operation is described in Algorithm 5.

Algorithm 5. The comprehensive search algorithm

```
OverallSearch(keyValue) explains the overall search algorithm from when a search operation is first requested to the MB-
  Tree to when an entry is finally retrieved from a leaf node.
1. (Search in the BPB.) The BPB is first checked for finding the requested entry.
   (a) If the entry having the same key-value as keyValue is found in the BPB, return the entry.
   (b) Otherwise, proceed to search further to the internal nodes.
2. (Traverse internal nodes.) Traverse the internal nodes to find a leaf node possibly containing the entry with the same
   key-value, following the B-Tree's internal-node traversing process.
3. (Search in the leaf node.) Algorithm 3 is executed.
```

4.7. A modified LRU buffer for enhancing search performance

We included a modified LRU buffer as the basic structure of an MB-Tree to enhance the search time, though it is not represented in the figures illustrated above. The classic LRU buffer causes page-writes even for search operations because of its replacement policy. Therefore, we propose a modified LRU buffer strategy suitable for flash memory.

The idea to utilize an LRU buffer for enhancing B-Trees' performance was first introduced by [6]. However, the traditional LRU policy can cause inefficient replacements such as replacing relatively high level nodes. Therefore, applying previous buffer management algorithms to B-Tree indexes should take into consideration B-Trees' features. To address this problem, a priority-based buffer replacement strategy [4], which considered several unique aspects of B-Trees, was proposed. The priority was determined by several factors such as the node level, number of node accesses, and time elapsed since the last access.

We propose a modified LRU buffer strategy that adopts the idea of the priority-based buffer replacement strategy, incorporating an idea that makes the buffer strategy efficiently perform on flash memory. The latter idea is to prevent buffer replacement when search operations are requested, which makes the buffer replacement occur only when insert operations are executed. This helps to eliminate the number of unnecessary page-writes caused by insert and search requests. When B-Trees operate without any LRU buffer, search requests cause no page-writes. However, if the classic LRU replacement is adopted, then the page-write operation will occur many times because of the dirty nodes to be replaced for the search requests. Therefore, it is inefficient for flash memory to insert nodes into the LRU buffer when search operations are requested.

The priority of the proposed LRU strategy is decided, first, by the node level, and, second, by the time elapsed since the last access. When the buffer becomes full by newly inserted nodes, nodes in the buffer are sorted by their node level, and the proposed LRU algorithm replaces the lowest level node (closest to the leaf node level). If the node levels between several nodes are the same, then it replaces the node with the longest elapsed time.

The proposed algorithm deals with insert and search requests as follows.

When insert operations are requested, the algorithm first checks whether the node to be updated is in the buffer or not. If the node is in the buffer, then the proposed algorithm updates the node and the node's last access time in the main memory without any page-write, and the process ends. Otherwise, it updates the node, writing the node on flash memory, and inserts the updated node into the LRU buffer. If the buffer is full, the replacement occurs as stated above. However, if the inserted node has the lower priority than the chosen node for the replacement, then nothing happens.

When search operations are requested, if the requested nodes are in the buffer, it updates only the last access time. Even if they are not in the buffer, it does not insert the nodes into the LRU buffer. In other words, the node accessed by the search operation is not maintained by the LRU buffer, even when there is enough space to hold the node in the buffer.

4.8. Simple performance analysis of insert and search operations

Although a B-Tree rewrites a leaf node through an insert operation, the MB-Tree gathers insert operations and then rewrites a leaf page performing FTES insert operations simultaneously. If the leaf page size of the MB-Tree is the same as the leaf node size of a B-Tree, the average insertion cost of the MB-Tree is approximately 1/|FTES| of the B-Tree's. This is because the B-Tree should rewrite a leaf node per insert operation but the MB-Tree can handle |FTES| operations by means of rewriting only one leaf page. To be more exact, if the FTES consists of only insert operations and |FTES| entries can be stored within a single leaf page, the MB-Tree will pay the cost of only two page-write time, one for a leaf page and one for the LNH of the leaf node. In the case of the B-Tree, even if a B-Tree's leaf node consists of only one page, it pays one page-write cost per insert operation. Since the MB-Tree pays a two page-write cost per |FTES| insert operations, the MB-Tree costs approximately 2/|FTES| times less than the B-Tree.

The cost of a search operation in the MB-Tree is one page-read more than in the B-Tree's since the search cost of phase 1 is negligible, phase 2's cost is the same as the B-Tree's, and phase 3's is one page-read more than the B-Tree's in Algorithm 5. The first phase of Algorithm 5 causes no page-read cost since it is a main memory-based search. The cost of the second phase of Algorithm 5 is the same as the B-Tree's cost, reading as many internal nodes as the height of the index tree minus one. In the third phase of Algorithm 5, the cost to read the leaf pages will be one page-read if the OID interval size for a KIV can be maintained at one by the split process as explained in Section 4.5.

In summary, the MB-Tree can reduce the overall page-write cost on flash memory by a factor of up to 2/|FTES| at the expense of one page-read search cost, compared to the B-Tree (A more detailed time-complexity analysis for this will be explained in the next subsection).

4.9. The time complexity of the insert and search operation of the MB-Tree

This time complexity analysis is presented for the MB-Tree without the LRU buffer. To simplify the formulas, this analysis considered when no delete or update operation is requested, and only insert operations are requested.

The MB-Tree's worst case time complexity of a search operation can be represented as (1).

$$C_r \cdot \left(2 + \left\lceil \log_M \frac{2 \cdot N}{M \cdot L} \right\rceil \right) \tag{1}$$

$$\lceil \log_M (2 \cdot N / M \cdot L) \rceil \tag{2}$$

The total number of current leaf nodes can be denoted as $2 \cdot N/(M \cdot L)$ in the worst case, where N, M, and L denotes the number of total entries currently inserted, the maximum number of entries that can be included in a single page, and the number of leaf pages in each leaf node, respectively. The worst case is when the entries of each leaf node fill exactly half of the leaf node. Consequently, the tree height excluding the leaf node can be denoted as (2). The search operation reads as many internal nodes as the tree height excluding the leaf node. Since the internal node size is fixed at one page, the number of pages read is the same as (2). In the search process within the leaf node, the LNH is read first. As explained in Section 4.5, the OID interval size for a KIV can be maintained at one. Therefore, if we assume that the interval size is maintained at one, the cost to read the leaf pages will be one page-read. The total search cost within a leaf node becomes 2 page-reads, including the reading of the LNH (the LNH size is fixed at 1 page). The total search cost is calculated by multiplying the summation of the cost caused in the internal nodes and the cost caused in the leaf nodes by the worst-case time consumed for a page-read (C_r).

The MB-Tree's worst-case time complexity of an insert operation can be represented as (3).

$$\left\{ 3 \cdot C_w + C_r \cdot \left(L + \left\lceil \log_M \frac{2 \cdot N}{M \cdot L} \right\rceil \right) \right\} \Big/ |\mathrm{FTES}| \tag{3}$$

The BPB flushes |FTES| TEs by using Algorithm 1. The page-write cost caused by the flushing operation is less than three page-writes, one for the writing the LNH, and one or two for writing leaf pages. The one page-write is always performed since writing the LNH is mandatory for every flush operation, and the other one or two page-writes are caused by executing the insert operations of the FTES. Usually, one page-write is performed since the entries to be written by the insert operations of the FTES can be contiguously written in a chosen leaf-page, but two page-writes occurs when the chosen page's empty space is not enough to store all the entries. The page-read cost caused by the flushing operation can be divided into the internal node search cost and the cost of reading all the leaf pages in the leaf node. The internal node search cost can be calculated by multiplying the summation of (2) and the leaf node size L by C_r. By accumulating the page-read cost and page-write cost, the total cost of a flush operation can be represented as (4), where C_w represents the worst-case time consumed for a page-write.

$$3 \cdot C_w + C_r \cdot \left(L + \left\lceil \log_M \frac{2 \cdot N}{M \cdot L} \right\rceil \right) \tag{4}$$

This is not the cost of an insert operation but the total cost of the flush operation caused by |FTES| insert operations. Therefore, the cost of an insert operation can be calculated by amortizing the flush cost by |FTES| insert operations, which means dividing the flush cost by |FTES|. When the delete operation is included in the FTES, |FTES| is decreased by the number of the delete operations.

5. Discussion

5.1. The BPB's features and differences compared with the previous similar schemes

In this section, we summarize the features of the BPB and indicate the common features and differences of the BPB as compared with the Reservation Buffer of the BFTL and flashDB, the JFFS3's journal tree, and some main memory based caches.

First, the BPB is not a cache, since caches focus on enhancing read performance in general, whereas the BPB is concerned only with reducing the update performance. A more cache-like structure is adapted to the MB-Tree to improve the search performance. This structure is the modified LRU buffer introduced in Section 4.7. The BPB accumulates update requests to leaf nodes as a type of log. The most similar cache to the BPB is the disk cache. Generally speaking, disk caches, which are the main memory based caches residing in file systems, load the portion of the file to be updated or read as a unit of a main memory page to its allocated main memory. Later, if other file-related requests need to read or update a file, the disk-cache first checks whether or not the requested file-fragment is on the cached pages. If it is on the cached pages, then the read or update operations are performed in main memory without accessing storage devices. This disk-cache is a type of write-back (write-behind) cache[1] in the respect that disk-cache defers mirroring the updates of the cached page to the hard disk. The BPB postpones the update operations to the leaf nodes (excluding the updates on the internal node) until the BPB's main memory space is fulfilled by the deferred update operations. With respect to deferring updates to the storage devices, the BPB has a common feature with the disk cache. However, the BPB has significant differences even with disk caches. Disk-caches load pages from the hard disk to main memory, not only when a file is updated but also when a file is read. On the contrary, the BPB does not load the corresponding leaf node, where read or update requests are to be done, to main memory at all. Instead, the BPB accumulates the update logs as TEs in the allocated main memory space until the main memory space is fulfilled by the TEs.

Second, the BPB has very common features with the journal area of the JFFS3 (introduced in Section 2.1), in that both accumulate the update logs to the B-Trees' leaf nodes. Since both simultaneously update the accumulated logs for the corresponding leaf node, they can reduce many page-reads and page-writes on flash memory compared with writing the leaf node to the flash memory for every update.

However, between the BPB and journal, there is a clear difference, specifically, that the journal resides on flash memory whereas the BPB is maintained on main memory. JFFS3 also has a main memory based index structure called the journal tree, which summarizes the information of the file system objects in the journal area and indicates where to access for the given query between the journal and the JFFS3 tree area. The journal tree has nothing common with the BPB except that both are a main memory based structure. Moreover, the detailed algorithm to update leaf nodes with the accumulated logs in the journal is not specified in the paper [3]. On the contrary, the BPB is characterized by the UPS algorithm which maximizes the performance gain from the simultaneous updates of the accumulated logs by flushing approximately the largest set of entries belonging to a certain leaf node (mentioned above as the FTES).

Third, the BPB is similar to the RB (Reservation Buffer) of the BFTL and flashDB in that all of them store the logs of update requests until the buffer is filled. However, the BPB stores only the update logs to the leaf nodes, excluding the update logs to internal nodes. Moreover, the UPS algorithm differentiates the BPB from the RB. Emptying the buffer, the RB flushes all the logs at once, whereas the BPB flushes only the FTES and remains the other update logs.

[1] A type of cache in which updates to the cached pages are not immediately reflected to the backing store.

5.2. Adapting the MB-Tree's idea and structure to other previous B-Tree variants

It can be considered to adapt the MB-Tree's design principles or structures such as the BPB and LNH to other popular B-Tree variants. As covered in Section 2.2, there exist many other B-Tree variants each of which was designed for more specific purpose such as efficient handling of the range queries, ensuring concurrency and recovery, and maximizing storage utilization.

It is evident that the MB-Tree's idea and the structures, the BPB and LNH, can be applicable to the B+-Tree without significant modification. We have considered the B+-Tree a target index to which the idea of the MB-Tree is to be adapted since the early days when we began this research. A B^*-tree has also been considered an MB-Tree's adaptation target. However, we are concerned that the MB-Tree's search performance may be decreased. On average, B^*-Tree's search performance is expected to be better than other variants, since its height decreases because of the high rate of storage utilization. However, the worst case search performance is degraded by the distributed index records in the sibling nodes. Moreover, the splitting and automatic leaf-node size control algorithm of the MB-Tree ought to be revised, since the B^*-Tree conducts its own 2–3 split. In a future research, the MB-Tree should be adapted so that it can assure concurrency and recovery in order to be considered more practically. Therefore, previous research on concurrency and recovery is important. Lomet and Salzberg's research [24] can be considered as a reference that makes the MB-Tree assure concurrency. The idea of the research, lazy splitting of the parent, is also helpful for reducing page-writes on flash memory. Manolopoulos's research [25], which assures concurrency and recovery at the same time, is also noteworthy as a method of designing a more concrete recovery model.

5.3. Possible MB-Tree variants

5.3.1. Excluding KIVs

In order to reduce the insertion cost at the expense of increasing the search cost, this scheme eliminates KIVs from the original design of the MB-Tree and maintains only OIDs (Order Information IDs) on the LNHs. The compaction process of the OIDs is not applied in this scheme. Therefore, all the OIDs corresponding to the all the entries of the leaf node are present in the LNH.

This method decreases the insertion cost. Despite the disabled compaction process, the OIDs without KIVs will occupy less space than with KIVs since a KIV occupies more space than an OID in general (this can be inferred by referring to Algorithm 2). Therefore, configuring each LNH with OIDs without KIVs allows the LNH to maintain the OIDs with a greater number of entries, decreasing the number of leaf nodes for the given fixed entries inserted to the leaf nodes. The FTES size is increased because of the reduced number of leaf nodes. Finally, the increased FTES size reduces the insertion cost due to the fact that the increased FTES size means that more insert operations are simultaneously executed at a cost of rewriting a leaf page (these can be inferred by deferring to formula (3)).

This variant also effects a slight increase in the search cost since a binary search should be applied to all the OIDs without the KIVs. In the original scheme, KIVs made it possible to directly search by using this key information as a type of hash function, but without it, an MB-Tree should perform a pseudo-binary search using OIDs' order information linked to the corresponding entries of the leaf node. This is possible because the OID informs the search process of the sorted order of each entry. To make the semi-binary search possible, the compaction process of the OIDs should be forbidden, since the pseudo-binary search algorithm cannot tell which OID belongs to which entry if the compaction process makes the consecutive identical OIDs into a single one.

Due to the degradation in search performance, this variant might has worse search performance than the flashDB. It is a more write-optimized B-Tree than the original MB-Tree since the insertion cost can be considerably reduced.

Fig. 19 represents the LNH of leaf node 2 when this scheme is applied. Whereas only two OIDs existed by the compaction process in the original LNH of leaf node 2, all the four OIDs corresponding to the four entries of leaf node 2 are found in Fig. 19. There are 4 more empty spots for the OID region since each leaf node can afford to sustain 8 entries.

5.3.2. Appending FEI

In order to reduce the page-write cost caused by delete operations, this scheme incorporates more information about what entries are free or need to be freed to the LNH structure, preserving the KIVs and OIDs. We named the information Free Entry Indicator (FEI). Appending FEI causes the LNH to be occupied by another *LeafEntries* bits, where *LeafEntries* denotes the number of entries in a leaf node.

This variant reduces the number of updates on leaf pages triggered by each delete operation. Generally, a delete operation is more likely to be directly linked to a physical update of a leaf page, since it has to be executed on the exact same leaf page where the entry to be deleted is located. The delete operation cannot be clustered, in contrast with insert operations, whereas insert operations can be contiguously stored on any free space of a leaf page because the MB-Tree's flush operation permits entries to be stored regardless of the increasing order of key-values in contrast to the original B-Tree. However, if the FEI is incorporated into the LNH, this variant does not need to update a leaf page for every delete operation. Instead, just flipping the bits of the FEI on the LNH can indicate that the entries are to be deleted (or freed). By doing so, multiple delete operations can be executed at once. Since an LNH is updated once for each of the BPB's flush operations, regardless of whether or not FEI scheme is applied, no extra page-write is needed on the leaf node. When the flush operation occurs, multiple delete operations can be recorded on the FEI region by flipping the corresponding bits to the entries to be deleted.

Fig. 20 represents the configuration of the indicator. Basically, each bit represents whether the corresponding entry is free or not. In the FEI region of Fig. 20, if the leaf node has *LeafEntries* entries, then the kth unset bit ($1 \leqslant k \leqslant \mathit{LeafEntries}$) represents that the kth entry is free (non-used, already deleted) or is to be freed (deleted), and the Kth set bit indicates that the Kth entry is not empty.

It also slightly increases the search cost since it consumes some LNH space to store the FEI. The space affects the OID interval size for a KIV, and consequently the interval size is increased because of the shortage of space in which the KIVs can be stored. Consequently, it affects the search cost, but not as much as the scheme that excludes KIVs, since this FEI scheme deprives KIV region of relatively little storage space because the FEI region size is fixed at *LeafEntries*/8 bytes. This size is considerably smaller than either the storage size needed for OIDs or KIVs.

This scheme is quite valuable for some applications having relatively many delete operations. Therefore, we conducted an essential experiment for this, not all the experiments. As it is described in Section 6.2 that the MB-Tree was evaluated in comparison with the other B-Trees the B-Tree, BFTL, and flashDB, this variant's performance was measured. This variant outperformed the other B-Trees as the MB-Tree did. Only difference between the performance of the MB-Tree and this variant was the slight increase in the search cost of this variant.

6. Experimental results

This section describes comparisons of the performance results of the MB-Tree, BFTL, flashDB, B-Tree. We implemented the MB-Tree on the base of the algorithms and structures mentioned above, and the BFTL, flashDB, and B-Tree were based on reference papers. For the BFTL and flashDB, the threshold for compaction (C) was fixed at 3, which was the same C value used in their papers [26,33]. As we mentioned in Section 2.3, the flashDB, and BFTL cannot operate without a certain amount of main memory space proportionate to the size of the NT (Node transition Table). In other words, having main memory space as much as the size of the NT is not merely an option to ensure that the BFTL and flashDB perform efficiently, but rather a pre-requisite to them. Therefore, in order to fairly conduct a performance comparison between the four indexes mentioned above, the same amount of main memory space used by the flashDB or BFTL was allowed to be used by the B-Tree and MB-Tree. Since the flashDB generally occupies less main memory space than BFTL (this will be confirmed in Section 6.2), it is fair for the B-Tree and MB-Tree to utilize the same main memory space as the flashDB utilizes. This is because if they are allowed to use the same main memory space as the BFTL, then they will unfairly have more advantages because they would use more main memory space than flashDB used. The main memory space was utilized as the modified LRU buffer, as stated above (from now on, the LRU buffer refers only to the modified LRU buffer), for both the B-Tree and the MB-Tree. However, the MB-Tree used a portion of the main memory space for the LRU buffer and the remainder for the BPB, whereas the B-Tree used the entire memory for the LRU buffer.

We performed several experiments examining insert and search requests for each index. In order to evaluate the performance of insert and search operations, the total number of page-read and page-write operations was measured when no greater than 1 million index records (inserted entries) were inserted on the SLC small block NAND flash memory specified in Table 1. Each inserted index record consisted of a four-byte integer key-value and four-byte pointer.

We used both real workloads and synthetic workloads. For the synthetic workloads, the integer key-value was randomly created ranging from 0 to 3 million. They were randomly inserted and searched in the experiments using synthetic workloads. We provided real workloads from the DNA sequence data of homo-sapiens chromosome 19.[2] Making an index that can efficiently search DNA sequence substrings with a certain length is a popular preprocessing method in DNA sequence alignment research [19]. Our real workload consists of four-byte integers, each of which corresponds to a 16-length substring of DNA sequence data. The workload contains 56 million integers. The integers were inserted according to the corresponding DNA sequences' order in the chromosome, up to 1 million times. Therefore, this workload has the characteristics of both random and sequential queries.

We categorized the experiments into five different experimental sets. In the first experimental set, the B-Tree's performance was measured under various conditions. What we aimed to do was to determine the best configuration of the B-Tree on flash memory. The insert and search performance was reported according to the node size and LRU buffer setting with the total number of the inserted entries fixed at 200,000 and the usable main memory space fixed at 20 kbytes by using real workloads. The second experimental set measured the performance of each index tree as the total number of inserted entries increased, by using both synthetic and real workloads. The flashDB was first tested, and the main memory space occupied by the flashDB was allocated to the MB-Tree and B-Tree. The BFTL's performance was also reported. In order to figure out how the main memory space allocated for the indexes affects their performance, the insert and search cost of the MB-Tree and B-Tree were measured in the third experimental set, as the main memory space increased with the total number of inserted entries fixed at 200,000, by using the real workloads. In the second and third experimental set, two versions of the B-Tree were examined. One was a B-Tree with the optimal node size that was proven in the first experimental set but does not have the proposed LRU buffer (denoted as B-Tree). The other is the more optimized B-Tree, which has both the optimal node size and the proposed LRU buffer (denoted as B-Tree (LRU)). The fourth experimental set assessed the necessity of the LNH (Leaf Node Header). The performance of the MB-Tree excluding the LNH was compared with the original MB-Tree including the

[2] The DNA sequence data was from http://www.ncbi.nlm.nih.gov/mapview/map_search.cgi?taxid=9606.

LNH with the total number of the inserted entries fixed at 200,000 and the usable main memory space fixed at 20 kbytes by using real workloads. The last experimental set measured the performance of the MB-Tree in relation to the rate of the main memory portion allocated for the LRU buffer with the total number of the inserted entries fixed at 200,000 and the usable main memory space fixed at 20 kbytes by using real workloads. This set was used to determine the optimal main memory usage percentage of the LRU buffer that would most enhance the performance of the MB-Tree.

6.1. The optimized B-Tree on flash memory

Varying the node size of the B-Tree, this experimental set measured the total number of page-reads and page-writes when insert and search operations are requested in order to determine the optimal B-Tree configuration on flash memory.

If the node size of the B-Tree is large, then it can be expected intuitively that the height of the B-Tree probably decreases. However, making nodes larger also results in increasing the search cost, since the larger the node, the more data needs to be read. Therefore, the problem of finding the optimal node size for B-Trees on flash memory requires an in-depth study.

As mentioned in [26], the optimal node size of a B-Tree can be determined by the utility-cost analysis proposed by [11]. The utility of increasing the node size is defined as the rate of decreasing the tree height as follows: $Utility = \log_2(\text{the fanout of the B-Tree node} - 1)$. The cost of increasing the node size is defined as the amortized access cost for a single node when a write operation is given as follows: $Cost = (Height \cdot ReadCost + WriteCost)/(Height + 1)$. *Height* is the B-Tree's height, and *ReadCost* and *WriteCost* are the average time spent reading and writing the node on flash memory, respectively. Therefore, the usefulness of increasing the node size is evaluated as *Utility*/*Cost*. By using this approach, [26] reported that one page is the optimal node size in the flash memory chip they used.

In the case of our research, the optimal B-Tree node size can be easily estimated by substituting the *ReadCost* and *WriteCost* with the page-read latency and page-write latency, respectively, of Table 1 in proportion to the node size. However, since this is just an estimation, we conducted an experiment to confirm the optimal node size with real workloads.

Fig. 6 illustrates the experimental results. In the experimental set, by increasing the node size by one page at a time from 1 page to 6 pages, the total number of page-reads and page-writes was measured after building a B-Tree index with 200,000 insert operations. The total number of page-reads was measured after executing 200,000 search operations, querying the same key-values of the previous insert operations. The performance of the B-Tree without the LRU buffer and the B-Tree with the LRU buffer is reported in the left side and right side of Fig. 6, respectively (20 kbytes main memory was allocated for the LRU buffer). The optimal node size of both B-Trees was one page, as represented in Fig. 6. The vertical axis represents the number of page-reads or page-writes, while the horizontal axis represents the type of requests (insert or search) and the type of cost (page-read or page-write). Every performance factor in both search and insert operations was increased as the page size increased. Additionally, the B-Tree with the modified LRU buffer showed a remarkably superior performance to that of the B-Tree without the LRU buffer, as represented in Fig. 6.

We concluded that the B-Tree with the node size of one page (512 bytes on SLC small block NAND flash memory) and the modified LRU buffer is optimal for flash memory through this experiment. Even though we did not report the experimental results conducted on different flash memory chips with different page size (SLC large block NAND, and MLC NAND), the results were the same as those of the SLC small block NAND flash, which means that one page is the optimal node size for B-Trees regardless of current flash memory types.

6.2. Performance comparison in relation to the number of input entries

This experiment measured the insertion and search cost of the MB-Tree, B-Tree, B-Tree (LRU), flashDB, and BFTL, increasing the total number of the inserted entries by 100,000 at a time up to 1,000,000. The insertion cost was measured by the total number of page-reads and page-writes that occurred while the indexes were built with the given number of insert operations. After building the indexes, the search cost was measured by the total number of page-reads that occurred while searching the same entries used when building the indexes. For the MB-Tree and B-Tree (LRU), the main memory space was allocated to the same extent as that of the flashDB. Forty percent of the given main memory space was allocated to the LRU buffer in the case of the MB-Tree, and the remainder was allocated for the BPB of the MB-Tree, whereas the entire given main memory space was allocated to the LRU buffer in the case of the B-Tree (LRU).

As presented in Fig. 7, the main memory space that had to be occupied by the BFTL and flashDB linearly increased as the total number of the inserted entries increased. The BFTL showed more dramatic increase than the flashDB. The BFTL occupied 1.75 times more main memory space on average than did the flashDB.

As mentioned above, we allocated the same amount of main memory space that the flashDB occupied to the MB-Tree and B-Tree (LRU) in each of the experiments with different inserted entries.

Figs. 9 and 10 represent the total number of the page-reads caused by insert operations for the synthetic and real workloads, respectively. For both real and synthetic workloads, the total number of page-reads was smallest in the case of the MB-Tree, followed by the flashDB, the B-Tree (LRU), the B-Tree, and lastly the BFTL, while all the indexes showed a linear increase. There was no significant difference between the experimental sets of synthetic and real workloads. For real workloads, the total number of page-reads caused by the MB-Tree was approximately 40%, 21%, 5%, and 2% of the page-read number caused by the flashDB, B-Tree (LRU), B-Tree, and BFTL on average, respectively.

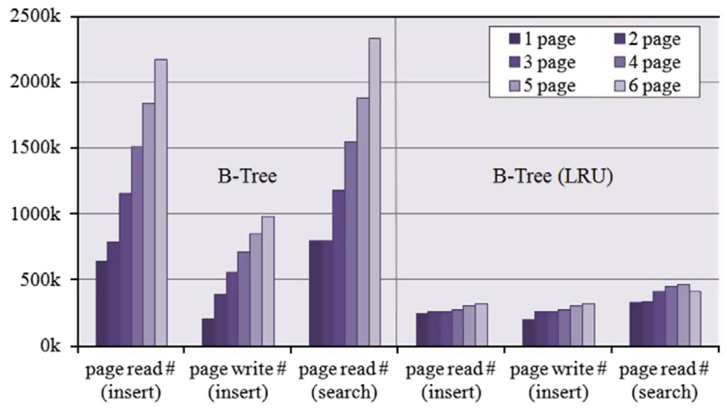

Fig. 6. The performance of the B-Tree without the proposed LRU buffer and B-Tree with the LRU buffer according to the node size with the total number of inserted entries fixed at 200,000 and the LRU buffer allocated 20 kbytes main memory (on the vertical axis, the performance is indicated by the number of page-reads and page-writes when insert and search operations were requested).

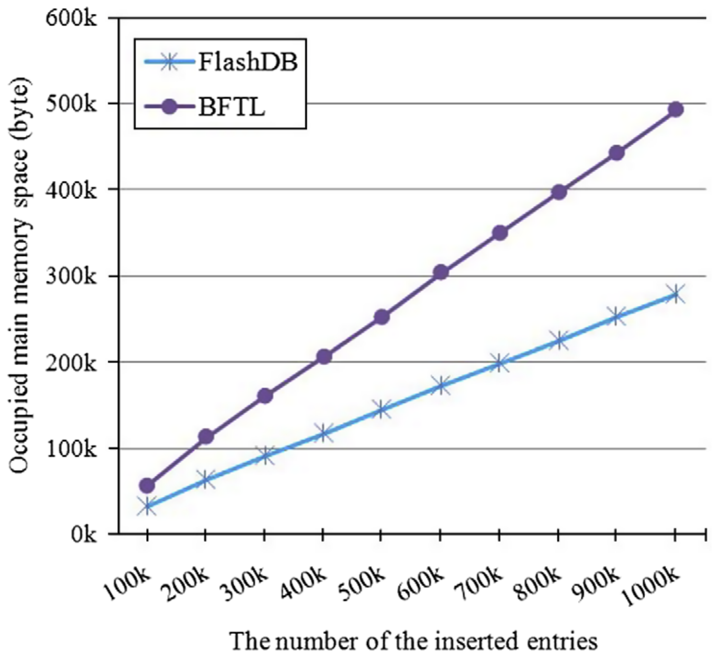

Fig. 7. The occupied main memory space by the flashDB and BFTL according to the total number of the inserted entries (real workloads).

In Figs. 11 and 12, the total number of the page-writes caused by insert operations is illustrated with synthetic and real workloads, respectively. For both real and synthetic workloads, the total number of page-writes was smallest in the case of the MB-Tree, followed by the flashDB, the BFTL, the B-Tree (LRU), and lastly the B-Tree, while all the indexes showed a linear increase. There were slight differences between the experimental sets of the synthetic and real workloads. In the experimental set of the real workloads, the page-writes of the MB-Tree, flashDB, B-Tree (LRU), and BFTL slightly decreased compared with that of the synthetic workloads. This is due to the sequential queries of the real workloads, which enhance the write performance of the buffers that operates similarly to write-back caches (all the indexes except the B-Tree have buffers operating similarly to write-back caches). With real workloads, the total number of page-writes caused by the MB-Tree was approximately 16%, 9%, 7%, and 14% of the number of page-writes caused by the flashDB, B-Tree (LRU), B-Tree, and BFTL on average, respectively.

This noteworthy result from the MB-Tree occurred because the MB-Tree's BPB efficiently saves page-reads and page-writes by flushing the FTES at once. The BFTL and flashDB also have similar insert-delaying buffers and flushing processes to those of the MB-Tree, but the gains from these are eclipsed by their log-structured entries and frequently executed

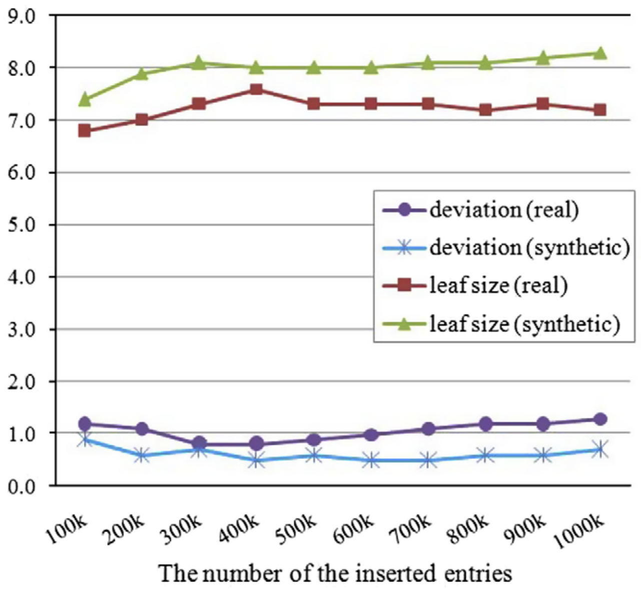

Fig. 8. The average leaf node size (flash-memory pages) and its standard deviation values of MB-Tree according to the total number of the inserted entries (both synthetic and real workloads).

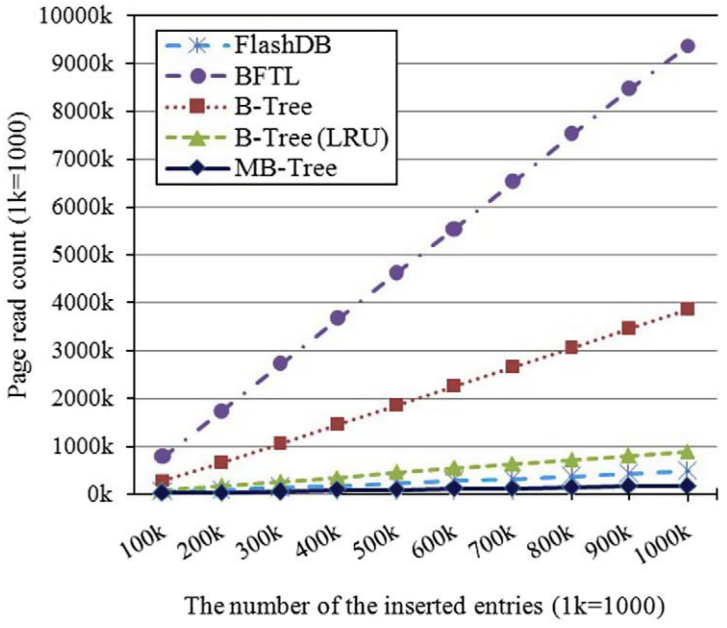

Fig. 9. The total number of page-reads caused by the insert operations according to the total number of the inserted entries (synthetic workloads).

compaction operations. The log-structured entries increase the tree height increasing page-read operations, and the compaction process causes extra page-writes.

Figs. 13 and 14 report the total number of page-reads that resulted from the search operations for synthetic and real workloads, respectively. There is no significant difference between the results of synthetic and real workload experiments. The search cost of the MB-Tree was approximately 94%, 191%, 48%, and 20% of the search costs caused by the flashDB, B-Tree (LRU), B-Tree, and BFTL on average, respectively.

None of B-Tree variants proposed for reducing page-writes on flash memory showed superior performance to the B-Tree (LRU) with respect to search operations. This occurred because the B-Tree variants have to sacrifice search performance to enhance the excessive page-writes of the original B-Tree. However, among the B-Tree variants, the MB-Tree demonstrated

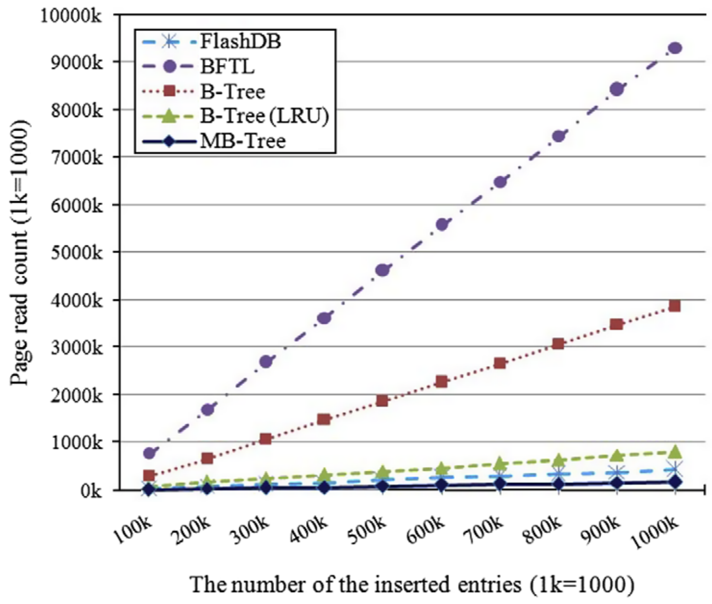

Fig. 10. The total number of page-reads caused by the insert operations according to the total number of the inserted entries (real workloads).

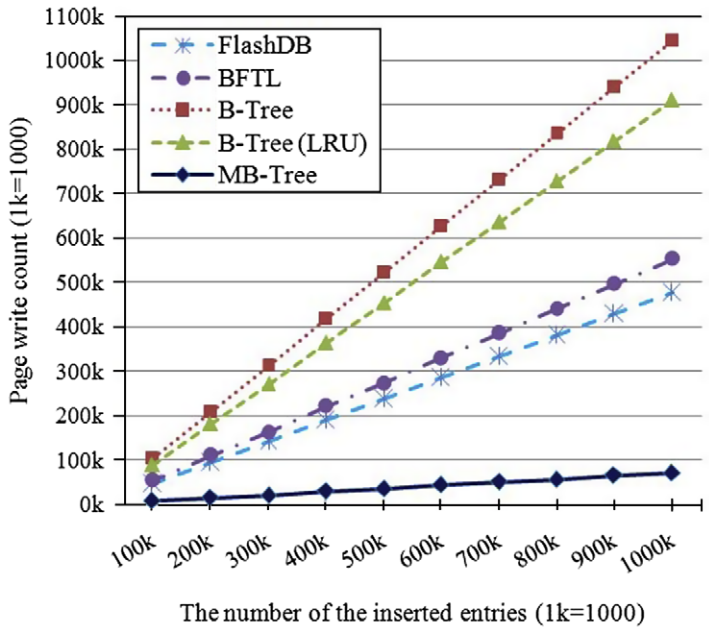

Fig. 11. The total number of page-writes caused by the insert operations according to the total number of the inserted entries (synthetic workloads).

the best performance in all aspects. This means that the MB-Tree accomplished one of its goals, a more efficient trade-off between insert and search performance than the previous B-Tree variants for flash memory.

In Fig. 8, the average leaf node size and its standard deviation are presented according to the number of the inserted entries. This experiment aimed to assess the efficiency of the automatic leaf-node size control. The average leaf node size and deviation converged on 8.01 and 0.62 in the synthetic workload experiment, respectively. In the case of real workloads, the average leaf node size and deviation were around 7.23 and 1.06, respectively. The small standard deviation value indicates that the automatic leaf-node size control algorithm makes the leaf node size converge without significant oscillation. Neither the leaf node size nor the standard deviation was significantly affected by the total number of the inserted entries, but they

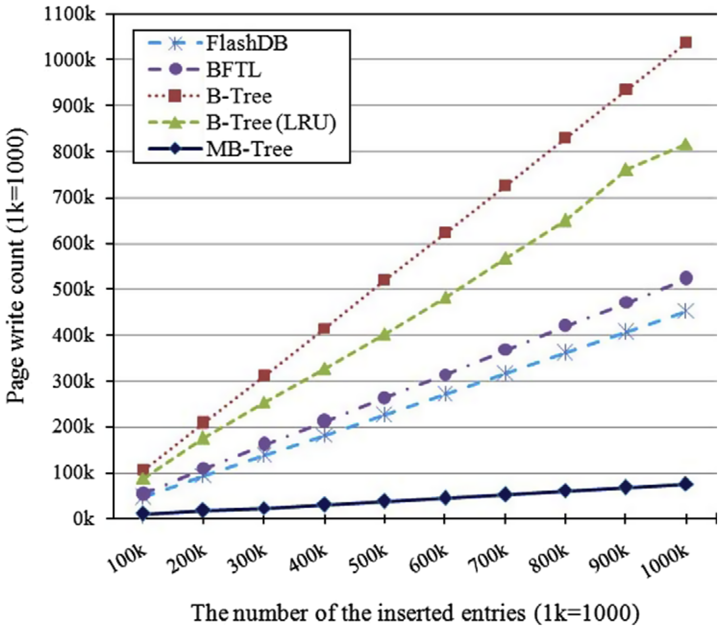

Fig. 12. The total number of page-writes caused by the insert operations according to the total number of the inserted entries (real workloads).

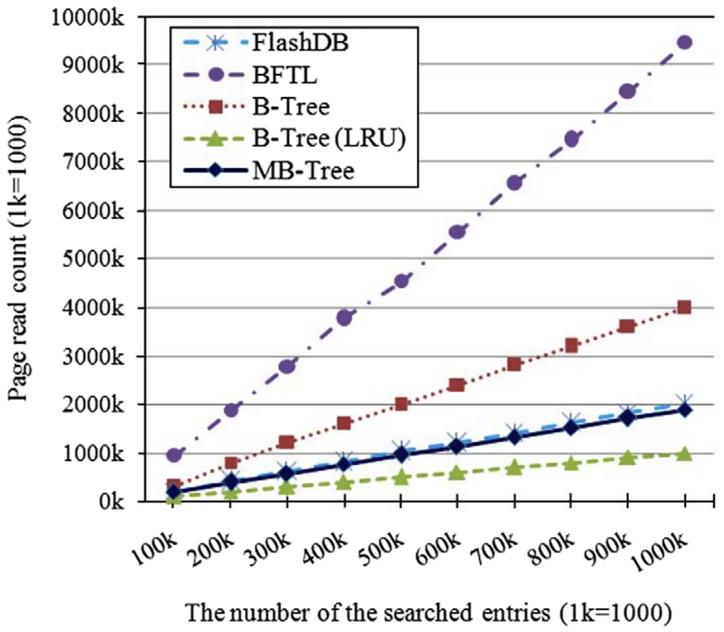

Fig. 13. The total number of page-reads caused by the search operations according to the total number of the searched entries (synthetic workloads).

were affected by their workload type. This is why in real workloads, the optimal leaf-node size was determined to be approximately eight pages which is one page smaller than the synthetic workloads, and why real workloads contain integers out of the range that synthetic workloads' integers have. The larger the range of the integers, the smaller the leaf node size. This is because the integers with larger ranges makes the LNH allocate more bits to its KIVs, and this makes the LNH contain fewer KIVs. This result confirms that the leaf-nose size automatic controlling algorithm successfully controls the leaf node size by taking into consideration the features of its own input data, making the node size converge on the most suitable size for the input data.

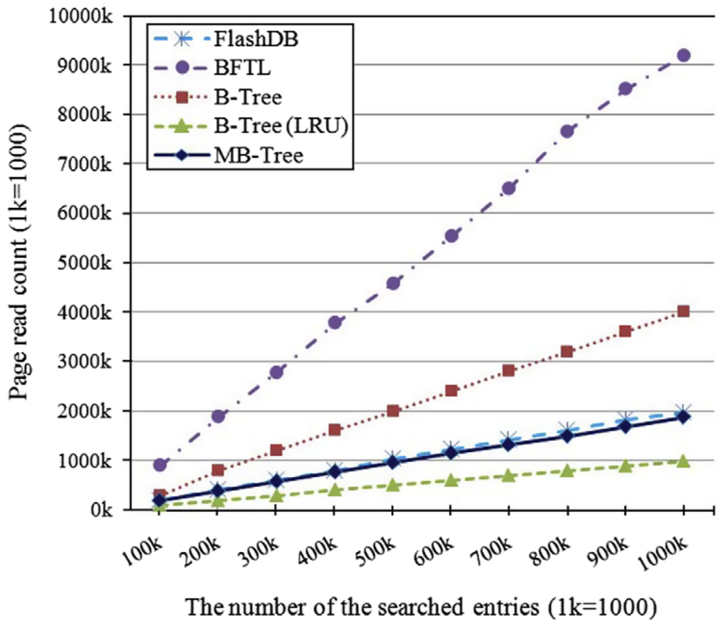

Fig. 14. The total number of page-reads caused by the search operations according to the total number of the searched entries (real workloads).

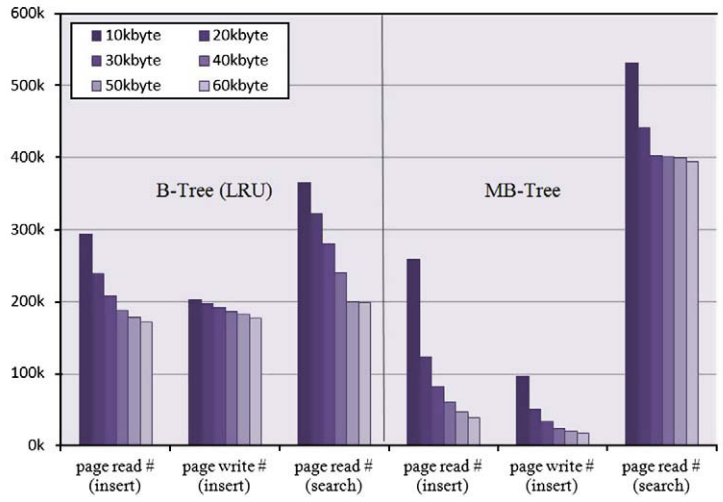

Fig. 15. The performance of the B-Tree with the proposed LRU buffer and MB-Tree according to the main memory space allocated from 10 kbytes to 60 kbytes with the total number of the inserted entries fixed at 200,000 (on the vertical axis, the total numbers of page-reads and page-writes are presented when insert and search operations were requested).

6.3. Performance comparison according to main memory space usage

In this experiment, we measured the total number of page-reads and page-writes, increasing the main memory space allocated for the MB-Tree and B-Tree (LRU) from 10 kbytes to 60 kbytes with the total number of the inserted entries fixed at 200,000. The flashDB occupied the maximum main memory space 60 kbytes when the total number of the inserted entries was 200,000, as shown in Fig. 7. Forty percent of the given main memory space for the MB-Tree was allocated for the MB-Tree's LRU buffer, and the remainder for the BPB.

As presented in Fig. 15, the page-reads and page-writes caused by insert operations more sharply decreased in the case of the MB-Tree as compared with the B-Tree (LRU). However, the page-reads caused by search operations decreased faster for the B-Tree (LRU) than the MB-Tree. The number of page-reads caused by the MB-Tree's insert operations started at 88% of the page-reads of the B-Tree (LRU)'s for the given 10 kbyte main memory and dramatically decreased to 22% of the page-reads of the B-Tree (LRU)'s for the given 60 kbyte main memory. The page-writes caused by the MB-Tree's insert operations started from 48% of the page-reads of the B-Tree (LRU)'s for the given 10 kbyte main memory, and sharply decreased to 9% of the page-reads of the B-Tree (LRU)'s for the given 60 kbyte main memory. This significant enhancement is due to the improved batch processing capability of the BPB made possible by the increased main memory space. On the contrary, the page-reads caused by the MB-Tree's search operation started at 145% of the page-reads of the B-Tree (LRU)'s for the given 10 kbyte main memory and increased to 198% of the page-reads of the B-Tree (LRU)'s for the given 60 kbyte main memory. This degradation is caused by the fact that the B-Tree (LRU) fully utilizes the given main memory space as the LRU buffer, but the MB-Tree has to use only a portion of the main memory space because of the BPB.

Because the MB-Tree's capability can efficiently utilize the main memory space, the MB-Tree dominated in the performance the other B-Tree variants for flash memory (the BFTL and flashDB) only with the 60 kbyte main memory space as demonstrated in Figs. 9–12, while the other B-Tree variants necessarily occupy more than 60 kbyte main memory. The 60 kbyte main memory amounts to 3.8% of the hard disk space spent storing 200,000 entries.

6.4. Performance of the MB-Tree with and without the LNH

This experiment measured the total number of page-reads and page-writes caused by the insert and search operations of the two MB-Tree versions by using real workloads. One version was the original MB-Tree that included both the BPB and LNH as its basic structure. The other version is the MB-Tree variant including only the BPB without the LNH. The total number of the inserted entries was fixed at 200,000, and 20 kbyte main memory space was allocated and a 40% portion of the given main memory space was used as the LRU buffer for both MB-Trees. This experimental set assessed how much performance gain the MB-Tree can obtain by using the LNH.

In Fig. 16, the performance of the MB-Tree without the LNH (which is denoted by MB-Tree (no LNH)) was measured, increasing the leaf node size from 1 page to 10 pages. The reason for varying leaf node size of the MB-Tree (no LNH) is that its leaf-node size cannot be automatically determined because the leaf-node size automatic control algorithm cannot work without the LNH. As shown in Fig. 16, the result indicates that a larger leaf node size slightly reduces the page-reads caused by the insert operations but significantly increases the total number of page-reads caused by search operations. On average, an insert operation updates flash memory space as much as half the leaf node size without the LNH, whereas the larger leaf node also reduces the insert cost by creating bigger FTESs. Therefore, these two effects cancel each other, resulting in a slight decrease of page-reads caused by insert operations. For search operations, the larger leaf node causes a linear increase of page-reads, since it should conduct a binary search without the LNH.

The performance gain from using the LNH in the MB-Tree is presented in Fig. 17. The gain values were calculated by subtracting the number of page-reads and page-writes caused by the original MB-Tree (reported in Figs. 10, 12, and 14) from the number of page-reads and page-writes caused by the MB-Tree (no LNH). Except when the leaf node size of the MB-Tree (no LNH) is 1 or 2 pages, the original MB-Tree dominated the MB-Tree (no LNH). Furthermore, even when the leaf node size is 1 or 2 pages, the benefit from page-reads and page-writes caused by the insert operations is significantly bigger than the loss from page-reads caused by the search operations. Even when giving no weight to a page-write in spite of its generally longer latency than a page-read, the gain was 2.9 and 3.8 times the loss in the 1-page and 2-page cases, respectively. This is because

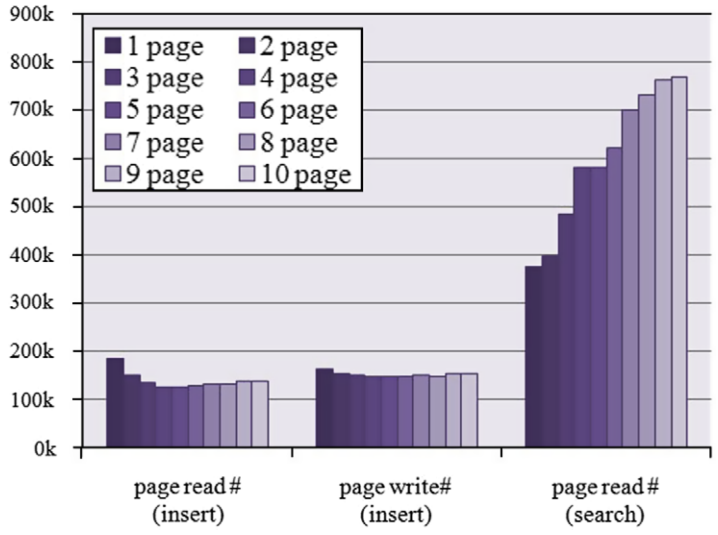

Fig. 16. The performance of the MB-Tree without the LNH according to the leaf node size from 1 page to 10 pages with the total number of the inserted entries fixed at 200,000 and the main memory space fixed at 20 kbytes and the main memory portion allocated for the LRU buffer fixed at 40% (on the vertical axis, the total numbers of page-reads and page-writes are presented when insert and search operations were requested).

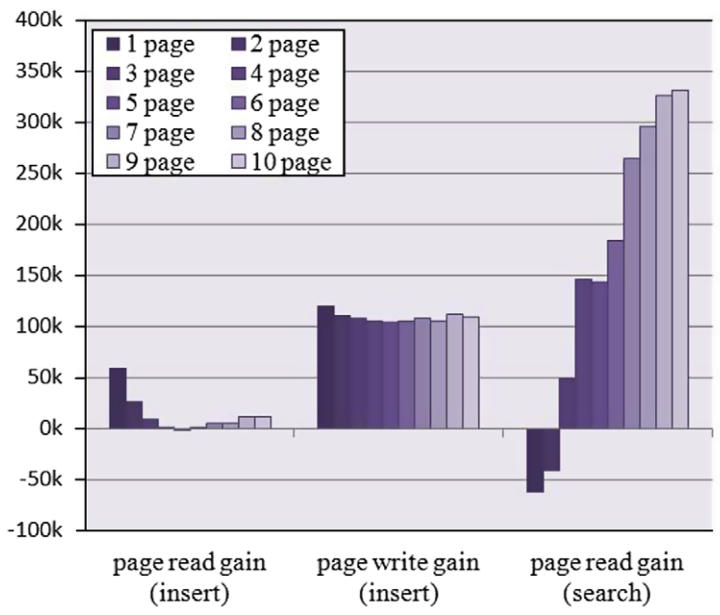

Fig. 17. The performance gain of the MB-Tree by using the LNH according to the leaf node size of the MB-Tree without the LNH from 1 page to 10 pages with the total number of inserted entries fixed at 200,000, the main memory space fixed at 20 kbytes, and the main memory portion allocated for the LRU buffer fixed at 40% (the vertical axis presents the difference in the number of page-reads or page-writes between the two MB-Trees with or without the LNH).

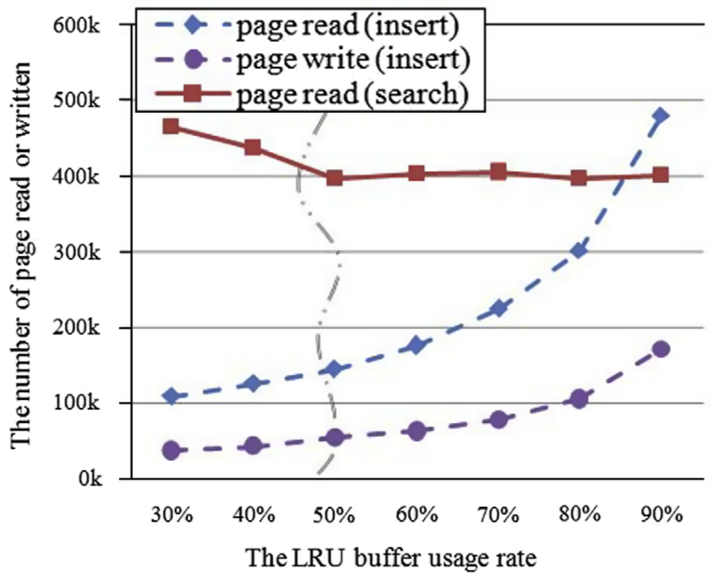

Fig. 18. The total number of page-reads and page-writes caused by the insert and search operations of the MB-Tree increasing the main memory portion allocated for the LRU buffer from 30% to 90% of the given main memory space (20 kbytes) with the total number of the inserted entries fixed at 200,000.

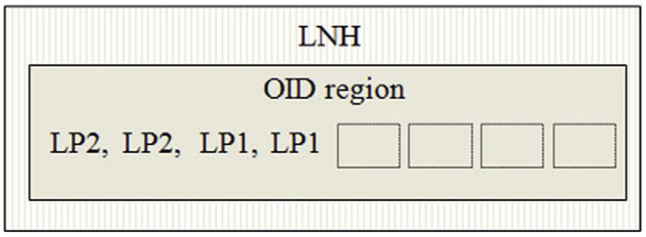

Fig. 19. The LNH configuration of a MB-Tree variant excluding KIVs.

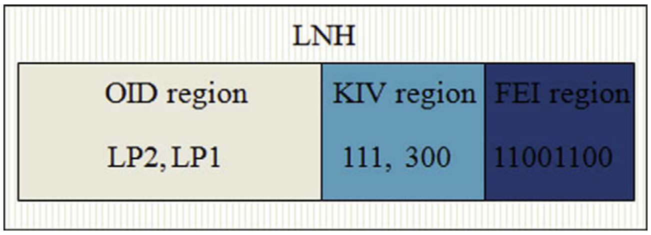

Fig. 20. The LNH configuration of a MB-Tree variant appending FEI.

the LNH makes an insert operation write only two or three pages in general, and it helps that only two page-reads occur within the leaf node when a search operation is requested. However, in the case of the MB-Tree (no LNH), an insert operation causes the flash memory space to be updated to as much half the size of the leaf node on average, and a search operation causes a binary search on the pages allocated for the leaf node.

6.5. Performance according to the main memory allocation rate for use in the LRU buffer

This experiment measured the total number of page-reads and page-writes caused by the insert and search operations of the MB-Tree, increasing the main memory portion allocated for the LRU buffer from 30% to 90% of the given main memory space (the entire usable main memory space was given as 20 kbytes).

As represented in Fig. 18, the total number of page-reads and page-writes caused by the insert operations of the MB-Tree sharply increased, whereas the total number of page-reads caused by the search operations of the MB-Tree steadily decreased as the LRU portion to the given main memory space became larger. The reason why the insertion cost sharply increased was that the allocation rate for the BPB decreased as the allocation rate for the LRU buffer increased, and therefore a small BPB made the FTES much smaller.

In conclusion, the allocation from 40% to 50% of the given main memory space for the LRU buffer is appropriate to ensure an efficient trade-off between the insertion and search cost. This is because at approximately 50% allocation, the total number of page-reads caused by the search operations decreased more steadily, and the total number of page-reads and page-writes caused by the insert operations increased more dramatically.

7. Conclusion

This paper proposed an MB-Tree index structure that reduces the overall page-write count and search time on flash memory to enhance the response time and life cycle of flash memory. First, the MB-Tree simultaneously handles a set of insert and delete requests by using a Batch Process Buffer and a unique path search algorithm. Next, to quickly perform search process operations, the addition of a Leaf Node Header on each leaf node made each search process faster and further enhanced the search time by automatically controlling each leaf node size according to the input data characteristics. Finally, the MB-Tree was designed to perform satisfactorily with limited main memory space, by storing both entries and the information of logical node organization on flash memory. Furthermore, we proposed not only the MB-Tree but also the optimal B-Tree configuration and a modified LRU buffer suitable for flash memory.

In order to evaluate the MB-Tree's performance, the page-write count and page-read count were measured with fewer than one million insert operations, and the page-read count was measured with fewer than one million search operations. Experimental results demonstrated that the MB-Tree significantly reduced the page-write count compared with the original B-Tree and other B-Tree variants proposed to provide better performance on flash memory. Furthermore, the MB-Tree searched faster than the other B-Tree variants, while the MB-Tree search performance was not significantly worse than the original B-Tree.

References

[1] Aleph One Company, Yet Another Flash Filing System.
[2] R. Bayer, E.M. McCreight, Organization and maintenance of large ordered indices, Acta Informatica 1 (1972) 173–189.
[3] A.B. Bityutskiy, JFFS3 Design Issues, <http://www.linux-mtd.infradead.org>.
[4] C.Y. Chan, B.C. Ooi, H. Lu, Extendible buffer management of indexes, in: Proceedings of the 18th VLDB Conference, 1992.
[5] L.P. Chang, T.W. Kuo, A real-time garbage collection mechanism for flash memory storage system in embedded systems, in: The Eighth International Conference on Real-Time Computing Systems and Applications, 2002.
[6] D. Comer, The ubiquitous B-Tree, ACM Computing Surveys 11 (1979) 121–137.
[7] Compact Flash Association, CompactFlashTM 1.4 Specification, 1998.
[8] P. Ferragina, G. Manzini, An experimental study of a compressed index, Information Sciences 135 (2001) 13–28.
[9] M.J. Folk, B. Zoellick, G. Riccardi, File Structure an Object-Oriented Approach with C++, Addison-Wesley, 1998.
[10] E. Gal, S. Toledo, Algorithms and data structures for flash memories, ACM Computing Surveys 37 (2005) 138–163.
[11] J. Gray, G. Graefe, The five-minute rule ten years later, and other computer storage rules of thumb, in: Proceedings of ACM SIGMOD Record, 1997, pp. 63–68.

[12] A. Guttman, R-Tree: a dynamic index structure for spatial searching, in: Proceedings of ACM SIGMOD International Symposium on the Management of Data, 1984, pp. 45–57.
[13] Intel Corporation, LFS File Manager Software: LFM.
[14] Intel Corporation, Understanding the Flash Translation Layer (FTL) Specification.
[15] D. Kang, D. Jung, J. Kang, J. Kim, μ-Tree: an ordered index structure for NAND flash memory, in: Proceedings of the Seventh ACM and IEEE International Conference on Embedded Software.
[16] A. Kawaguchi, S. Nishioka, H. Motoda, A flash-memory based file system, in: Advanced Computing Systems, USENIX Technical Conference, 1995, pp. 155–164.
[17] A.M. Keller, G. Wiederhold, Concurrent use of B-Trees with variable length entries, ACM SIGMOD Record 17 (1988) 89–90.
[18] H. Kim, S. Lee, A new flash memory management for flash storage system, in: Proceedings of the Computer Software and Applications Conference, 1999, p. 284.
[19] W. Kim, S. Park, J. Won, S. Kim, J. Yoon, An efficient DNA sequence searching method using position specific weighting scheme, Journal of Information Science 32 (2006) 176–190.
[20] D. Knuth, The Art of Computer Programming, vol. 3, Addison-Wesley, 1973.
[21] S. Lee, D. Park, T. Chung, D. Lee, S. Park, H. Song, A log buffer-based flash translation layer using fully-associative sector translation, ACM Transactions on Embedded Computing Systems 6 (2007).
[22] S. Lim, K. Park, An efficient NAND flash file system for flash memory storage, IEEE Transactions on Computers 55 (2006).
[23] S. Lim, M.H. Kim, Restructuring the concurrent B+-Tree with non-blocked search operations, Information Sciences 147 (2002) 123–142.
[24] D.B. Lomet, B. Salzberg, Concurrency and recovery for index trees, The VLDB Journal 6 (1997) 224–240.
[25] Y. Manolopoulos, B-Trees with lazy parent split, Information Sciences 79 (1994) 73–88.
[26] S. Nath, A. Kansal, FlashDB: dynamic self-tuning database for NAND flash, in: Proceedings of the Sixth International Conference on Information Processing in Sensor Networks, 2007, pp. 410–419.
[27] A.L. Rosenberg, Time-and space-optimality in B-Trees, ACM Transactions on Database Systems 6 (1981) 174–193.
[28] A.L. Rosenberg, L. Snyder, Compact B-Trees, ACM Transactions on Database Systems (1979).
[29] M. Rosenblum, J.K. Ousterhout, The design and implementation of a log-structured file system, ACM Transactions on Computer Systems 10 (1992) 26–52.
[30] SSFDC Forum, SmartMediaTM Specification, 1999.
[31] D. Woodhouse, Red Hat Inc., JFFS: The Journalling Flash File System.
[32] C.H. Wu, L.P. Chang, T. Kuo, An efficient B-Tree layer for flash-memory storage systems, in: Proceedings of the Ninth International Conference on Real-Time and Embedded Computing Systems and Applications, 2003, pp. 409–430.
[33] C.H. Wu, L.P. Chang, T. Kuo, An efficient B-Tree layer implementation for flash-memory storage systems, ACM Transactions on Embedded Computing Systems 6 (2007).
[34] C.H. Wu, L.P. Chang, T. Kuo, An efficient R-Tree implementation over flash memory storage systems, in: Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems, 2003, pp. 17–24.