A Novel B-Tree Index with Cascade Memory Nodes for Improving Sequential Write Performance on Flash Storage Devices

Bo-Kyeong Kim 1, Gun-Woo Kim 2 and Dong-Ho Lee 2,*

1 FW Nextgen Tech, SK Hynix, Icheon 467010, Korea; bkhacker@hanyang.ac.kr
2 Department of Computer Science and Engineering, Hanyang University, Ansan 15588, Korea; kgwhsy@hanyang.ac.kr
* Correspondence: dhlee72@hanyang.ac.kr; Tel.: +82-31-400-5236

Received: 20 December 2019; Accepted: 19 January 2020; Published: 21 January 2020

Abstract: Flash storage devices such as solid-state drives and multimedia cards have been widely used in various applications because of their fast access speed, low power consumption, and high reliability. They consist of NAND flash memories that perform slow block erasures before overwriting data on a prewritten page. This characteristic can lead to performance degradation when applying the original B-tree on the flash storage device without any changes. Although various B-trees have been proposed for flash memory, they still require many flash operations that degrade overall performance. To address the problem, we propose a novel B-tree index structure that reduces the number of write operations and improves the sequential writes by employing cascade memory nodes. The proposed B-tree index structure delays the updates for the modified B-tree nodes and later performs batch writes in a cascade manner. Also, when records with continuous key values are sequentially inserted, the proposed B-tree index structure does not split the leaf node so that it improves write throughput and page utilization. Through mathematical analysis and experimental results, we show that the proposed B-tree index structure always yields better performance than existing techniques.

Keywords: flash memories; indexes; tree data structures

1. Introduction

Flash storage devices have been widely used in diverse application areas from small embedded systems to large scale servers. There are various kinds of flash storage devices such as an embedded multimedia card (eMMC), a solid-state drive (SSD), and an all-flash array. Compared to the traditional magnetic disk drive, the flash storage device has many advantages like fast access speed, low power consumption, and high reliability [1–3].

The flash storage device inherits the distinctive characteristics of NAND flash memory because it is composed of a number of NAND flash arrays. A flash array consists of many blocks with a fixed number of pages each. Differently with the magnetic disk drive, the flash memory requires an erase operation before data is written in the same physical address because it cannot overwrite data in-place. Furthermore, the erase operation is performed in a block unit and so it is much slower than the read or write operations that are done in a page unit [4].

The flash memory cannot be directly deployed as a storage device in the conventional host system by itself due to the aforementioned physical characteristics. Therefore, many vendors of the flash storage device have adopted an intermediate software layer, which is called a flash translation layer (FTL), between the host systems and the flash memory. The key role of the FTL is to redirect each write request from the host to an empty flash page that has already been erased in advance, thus overcoming...
the limitation of in-place updates. In the internal architecture of the flash storage device, its controller performs FTL functionality that makes the flash memory work as a block device like the magnetic disk drive [5–7].

A B-tree [8,9] index structure is widely used in conventional file systems (e.g., ReiserFS [10], XFS [11], Btrfs [12]) and database systems (e.g., PostgreSQL [13], MySQL [14], SQLite [15]) because of its ease of construction and retrieval performance. The B-tree intensively overwrites data into the same node so as to keep the balance of the tree height. Therefore, severe performance degradation occurs when the B-tree index structure is deployed in the flash storage device although the FTL provides an efficient mapping algorithm.

In order to enhance the performance of the B-tree on flash devices, various B-tree index structures have been proposed for flash memory. Flash-aware B-tree index structures are classified into two categories. The first group has B-trees that employ the memory buffer to improve the write throughput. These buffer-based B-trees are much faster than another group, but they suffer from the high cost of maintaining the memory buffer and risk of data loss in the case of a sudden power failure. The second group’s B-trees are the variations that modify node structure to avoid in-place updates. These structure-modified B-trees have more reliable features and also need small memory resources than buffer-based B-trees. However, their write performance is poor because they invoke many write operations.

In this paper, a novel B-tree index structure is proposed for the flash storage device in order to improve the overall performance with small memory resources and page utilization. For reducing the number of write operations, it keeps the modified B-tree nodes by key insertions in the main memory until the batch writes are performed in a cascade manner. Additionally, the proposed B-tree index structure does not split leaf nodes so as to avoid additional write operations for the node split and store more keys in the leaf node when records with continuous key values are sequentially inserted. Through mathematical analysis and various experimental results, we show that the proposed B-tree index structure always yields better performance than existing works.

The rest of this paper is organized as follows. Section 2 reviews flash-aware index structures and discuss the drawbacks of the related works. Section 3 describes the proposed B-tree index structure. In Sections 4 and 5, we show the superiority of the proposed index structure through mathematical analysis and various experiments. Finally, we conclude in Section 6.

2. Background and Related Work

2.1. Flash Storage Devices

Figure 1 shows the internal architecture of the flash storage device. The flash storage device consists of NAND flash arrays and a controller that maintains them. A NAND flash array has a number of blocks each of which contains a fixed number of pages. Each page is composed of a sector area to store user data from the host interface and a spare area to store metadata for managing the page.

Figure 1. The architecture of the flash storage device.
NAND flash memory in the flash array has unique physical characteristics compared to the magnetic disk drive. First, the flash memory provides three basic operations: read, write, and erase. The read operation is much faster than the write operation and the write operation is faster than the erase operation. In addition, read and write operations are performed in a page unit, whereas the erase operation is done in a block unit. Therefore, these asymmetric I/O speeds and units should be considered when a new algorithm is designed for the flash memory.

Second, the flash memory requires an erase-before-write procedure because it does not allow an in-place update. For overwriting the data in the prewritten page, the block containing the page to be updated must be erased in advance. Due to this erase-before-write operation, the flash memory cannot be directly deployed as a block device by itself in the conventional host system. Therefore, the FTL, which hides the constraint of the in-place update and erase-before-write procedure, is required between the host system and the flash memory. The FTL functionalities are performed in the controller of the flash storage device.

The key role of the FTL is to process the logical-to-physical address mapping from the host system to the physical flash memory. The address mapping is largely classified into sector mapping and block mapping. The sector mapping [16–18] maps every logical sector from the host system to the corresponding physical page on the flash memory. In order to avoid the in-place update, every write request from the host system always assigns a new empty page on the flash memory. As a result, until there are no free pages of the flash memory, the sector mapping quickly performs all write requests without giving rise to erase operations. However, the size of mapping information significantly increases as the storage capacity increases because every logical sector has its own physical page address.

In contrast, the block mapping [19–22] handles address information in a block unit, not a page. The size of its mapping information is very small because its logical sectors can be accessible by calculating only the count of the physical blocks and pages. Although it uses only a few memory resources, the block mapping suffers from frequent overwrites that invokes many erase and write operations on the flash memory. Recently, there are some combined FTL algorithms with the block mapping and the sector mapping algorithm according to the memory resource [23–25].

In a different way, Demand-based FTL (DFTL) [26] stores all the page mapping information in the flash memory instead of the main memory and uses the stored mapping information for accessing the flash memory. However, a large number of read/write operations for getting address information are invoked as the capacity of the flash storage device significantly increases.

2.2. B-Tree on the Flash Storage Device

A B-tree index structure is widely used to quickly access the stored data in file systems and database management systems. However, it may be inefficiently built on the flash storage device due to its frequent updates for the same node.

Figure 2 shows the B-tree index structure that consists of root node A and three-leaf nodes (B, C, and D). Assume that every node is mapped on a flash page. If a record with key 7 is inserted, leaf node B will be updated with the new record. In the sector mapping, as shown in Figure 2a, updated leaf node B’ is simply written into an empty page on the flash block and the old page storing leaf node B is invalidated. Since the amount of invalid page increases as the number of insert operations increases, the flash storage device should perform garbage collection to reclaim the invalid pages. This garbage collection invokes many read/write operations in addition to erase operations.

In the block mapping, as shown in Figure 2b, the block including leaf node B is removed in advance for updating leaf node B. After erasing the block, updated leaf node B’ is stored and valid nodes (A, C, and D) are rewritten into the erased block. In this example, inserting a record with key 7 invokes many flash operations (e.g., one erase operation and four read/write operations). Compared to the sector mapping, an insert operation requires more read, write, and erase operations on the flash memory to update the leaf node.
To improve the performance, various B-tree variants have been proposed for flash memory. In general, they can be classified into two categories, buffer-based B-trees and structure-modified B-trees. In the following subsection, we review the features and problems of the previous B-tree index structures in more detail.

2.3. Buffer-Based B-Trees

Buffer-based B-trees employ the write buffer to reduce the number of write operations. Figure 3 shows a brief process of how buffer-based B-trees flush the inserted records from the buffer to the flash memory. In this example, the logical B-tree is built with four nodes A, B, C, and D by inserting records in the following key sequences: 1, 17, 9, 19, 13, 3, and 11. Assume that a page can store a maximum of four records in the flash memory. As shown in Figure 3a, Buffer-based FTL (BFTL) [27], the first buffer-based B-tree, temporarily stores the inserted records into the buffer. When the buffer overflows, BFTL flushes the data in a page unit to the flash memory. In this example, BFTL first flushes records with keys 1, 17, 9, and 19 to page #0 on the flash memory. Although BFTL reduces the number of write operations, its retrieval performance is very poor because it requires many read operations to find records that are scattered in the several pages.

Through these two examples, it is obvious that the performance degradation occurs in the flash storage device regardless of FTL mapping algorithms when constructing the B-tree index structure. To improve the performance, various B-tree variants have been proposed for flash memory. In general, they can be classified into two categories, buffer-based B-trees and structure-modified B-trees. In the following subsection, we review the features and problems of the previous B-tree index structures in more detail.

Figure 2. B-Tree on the flash storage device.

Figure 3. Buffer-based B-Trees for flash memory.
To store all the records that exist in the same logical node together, the recent buffer-based B-trees (e.g., IBSF [28], Lazy update B+-tree [29], and AS-tree [30]) reorder the inserted records in the buffer. Figure 3b shows an example of flushing nodes in IBSF. When the buffer overflows, IBSF finds all the victim records by simply referring to the key of the first inserted record in the buffer. Since leaf node B is the relevant victim node, all records associated with leaf node B (keys 1 and 3) are flushed from the buffer to page #0 on the flash memory. In contrast to IBSF, Lazy update B+-tree selects the victim node related to the least recently inserted record to improve the write throughput and the buffer hit ratio.

AS-tree sorts all records in the buffer according to the logical node for batch writing. When the buffer overflows, all sorted records are sequentially flushed in a page unit to the flash memory. For avoiding overwriting data, it assigns more pages than the other index structures and requires garbage collection to remove invalid nodes. Generally, the overall performance increases as the buffer size increases. However, as the buffer size significantly increases, the overall performance could decrease because IBSF, Lazy update B+-tree, and AS-tree reorganize all the records in the buffer. In addition, there are other buffer-based B-trees such as MB-tree [31], FD-tree [32], and AD-tree [33]. They quickly create the index structure but they consume more time to find data due to the flexible node size.

Most buffer-based B-trees guarantee fast performance without regard to FTL mapping algorithms. However, their performance is largely affected by the size of the high-cost memory buffer and the risk of data loss still remains in the case of a sudden power failure. Therefore, their solutions do not demonstrate the effective performance in the small embedded system that has limited memory resources and insecure power supply.

2.4. Structured-Modified B-Trees

Structure-modified B-Trees change the node structures to avoid in-place updates. They are designed to handle the physical address because the early embedded system equips the raw flash memory without the FTL. Figure 4 shows a brief process of how structure-modified B-trees store the inserted records to the flash memory. In this example, the logical B-tree is built with six leaf nodes and four parent nodes including root node A. Assume that a node of the B-tree is mapped on a page on the flash memory.

![Figure 4. Structure-modified B-Trees for flash memory.](image)

In flash file system JFFS3 [34], the wandering tree is the first B-tree index structure that considers the characteristics of the flash memory. In order to avoid the in-place update, the wandering tree stores all the updated nodes into empty pages on the flash memory when inserting a record. As shown in Figure 4a, lead node J, parent node D, and root node A are written into new empty pages, respectively,
when inserting a record into leaf node J. As a result, it does not perform many erase operations caused by in-place updates, but it still requires many read/write operations for updating parent nodes.

In order to reduce the additional operations for updating parent nodes, μ-tree [35] and μ*-tree [36] write all the updated nodes into a single page on the flash memory. As shown in Figure 4b, μ-tree writes updated leaf node J' and its parent nodes (nodes D and A) into an empty page on the flash memory when inserting a record into leaf node J. To do this, μ-tree divides a page into several partitions in fixed-size. μ*-tree dynamically assigns the size of the leaf node according to its page state for improving the page utilization. In μ-tree and μ*-tree, node splits frequently occur and the tree height increases rapidly because the sizes of leaf nodes are less than that of the original node for the B-tree. This feature causes severe performance degradation when building the index structure.

IPL B+-tree [37] defers updating the parent nodes by storing only the changed records as logs. As shown in Figure 4c, IPL B+-tree divides a flash block into two areas: a data area that stores tree nodes and a log area that stores inserted, deleted, and updated logs corresponding to the data area. The size of the log area is fixed based on IPL [38]. In contrast to the IPL B+-tree, dIPL B+-tree [39] dynamically assigns the number of log pages on the block in order to efficiently use the log pages. If a record related to leaf node J is inserted, the record will be temporarily stored into the log block. When the log area becomes full, the data area and the log area are merged. This merge operation invokes many read, write, and erase operations on the flash memory.

Similar to the IPL B+-tree, LA-tree [40], LSB-tree [41], and BbMVBT [42] also store all nodes to be updated into the temporary area on the flash memory for avoiding in-place updates. The retrieval performance of these index structures that employ log area is worse than that of the B-tree due to the traversal of the log areas.

Most structure-modified B-trees guarantee high reliability because they directly store all updated records into the flash memory. However, they require additional operations such as parent updates and merge operations according to their structures.

Table 1 shows the characteristics of the buffer-based B-tree and the structure-modified B-tree in terms of performance, reliability, and memory usage. It is necessary to develop a novel B-tree index structure that has the advantages of two B-trees such as fast performance, high reliability, and low memory usage.

| Item          | Buffer-Based B-Trees | Structure-Modified B-Trees |
|---------------|----------------------|-----------------------------|
| Performance   | High                 | Low                         |
| Reliability   | Low                  | High                        |
| Memory Usage  | High                 | Low                         |

3. CB-Tree: A B-Tree Employing Cascade Memory Nodes

As mentioned in Section 2.2, the performance degradation is inevitable when directly constructing the B-tree index on flash storage devices. To address this problem, buffer-based B-tree approach (Section 2.3) and structure-modified B-tree approach (Section 2.4) have been proposed. They have their own advantages and disadvantages as mentioned in Sections 2.3 and 2.4. The key idea of CB-tree employs only the advantages of each approach. That is, it yields good performance with small memory resources and also guarantees high reliability by storing all updated records into the flash memory.

In this section, we present a novel B-tree index structure, which is called CB-tree, improving sequential writes with cascade memory nodes. The design goals of CB-tree are as follows. The first goal is to quickly create the index structure with small memory resources. The second goal is to quickly find a record without visiting extra area irrelevant to the B-tree nodes. CB-tree improves the write throughput by employing the cascade memory node to keep inserted or deleted records in the main memory and later apply them into the flash memory in a batch process. Additionally, it reduces the
number of write operations by not splitting the leaf nodes when records are sequentially inserted in 
continuous key order.

3.1. Overview

CB-tree classifies the nodes into memory node for performance and flash node for reliability. The 
memory node is a node that stays in the main memory, and the flash node is a node to be stored in 
the flash memory. CB-tree employs only one memory node for each level of the B-tree to reduce the 
usage of memory resources and the risk of data loss. The more memory nodes there are, the better the 
performance. However, the usage of memory resources and the risk of data loss may also increase. 
Therefore, CB-tree maintained only one memory node for each level of the B-tree and consequently 
the number of memory nodes is equal to the height of the B-tree. In this paper, all the memory nodes 
from the leaf node to the root node are called cascade memory nodes.

When a record is inserted or deleted in the B-tree, several nodes of the B-tree are traversed 
from the root node to the leaf node for finding the target leaf node. The insertions and deletions are 
performed in the target leaf node after arriving at the leaf level. If the target leaf node overflows or 
derunderflows, the visited parent nodes during traversing will be updated. These parent updates invoke 
many internal read, write, and erase operations on the flash storage device. Since, in CB-tree, all the 
insert and deletion operations are completed in only the cascade memory nodes, the number of read, 
write, and erase operations on flash devices may be reduced. Only node switching invokes the read 
and write operations for the flash storage device. If a flash node is visited for inserting or deleting a 
record, node switching occurs between the current visiting flash memory and the memory node 
existing at the same level. That is, the content of the memory node at the same level is flushed to the 
flash memory and then the content of currently visited flash node is loaded into a new memory node. 
That is a basic process of node switching that will be explained in Section 3.2 for more details.

When a record is inserted, some nodes from the root node to the leaf node are visited to insert the 
record. If the types of all the visited nodes are all the memory nodes, the write operations on the flash 
memory will not be invoked because the insert operation is performed only in the main memory. If not 
so, node switching will happen.

Figure 5 illustrates the overview of CB-tree. There are six leaf nodes and four parent nodes 
including the root node. In this example, node B, C, E, F, G, H, and I are flash nodes and nodes A, D, 
and J are memory nodes. Therefore, only seven flash nodes are stored in flash memory. If a record 
insertion occurs in leaf node J, root node A, internal node C, and leaf node J will be visited. Since the 
visited nodes are all the memory nodes, the insert operation is performed in the cascade memory nodes 
without node switching. Therefore, this insertion for leaf node J does not invoke any operations 
on the flash memory.

Whereas, if a record is inserted into leaf node H, root node A, internal node C, and leaf node H 
will be visited. Since nodes C and H are flash nodes, two nodes are swapped for the current memory 
nodes D and J at the same levels, respectively. In other words, memory nodes D and J are flushed from 
the main memory to the flash memory and then flash nodes C and H are loaded as new memory nodes 
in the main memory.
If the CB-tree is built by inserting records with random key values, its performance will decrease because node switching frequently occurs. On the other hand, if records are sequentially inserted, node switching will rarely occur because many record insertions are performed in the memory node. In conventional file systems, they have about 80–90% sequential patterns among the write patterns \[43,44\]. The access pattern of multimedia systems also has a sequential write pattern \[45\]. Therefore, when the CB-tree is built with realistic data in the practical host system, its performance may not decrease because node switching is rarely invoked.

3.2. Insert Operation

The insert operation of CB-tree does not directly write the inserted record into the flash memory by storing it to the main memory. To defer the write operation on the flash memory, CB-tree employs the cascade memory nodes to insert a record and update the parent nodes. All the insert operations are carried out in the cascade memory nodes. To do this, node switching is performed in advance in the case that there is the flash node among the visited nodes. However, if the content of the memory node is not changed, the flash node will be simply loaded into a new memory node without node switching. This case is that the record is simply inserted or deleted in the leaf node. At this time, the previous memory node not to be changed is just removed from the main memory.

Figure 6 shows an example of the insert operation in the CB-tree where three memory nodes (A, B, and E) and two flash nodes (C and D) exist. When a record with key 35 is inserted, nodes A, B, and E are traversed for the insertion. Since all the visited nodes A, B, and E, are in the cascade memory nodes, the record with key 35 is inserted into the memory node without invoking any flash operations.

![Figure 6. Example of the insert operation.](image)

However, if there is at least one flash node among the visited nodes, node switching occurs between the visited flash node and the memory node at the same level. If a record with key 25 is also inserted after inserting the record with key 35, nodes A, B, and D are traversed for the insertion. Since leaf node D is not the memory node, node switching occurs between leaf node D and leaf node E at the same level. That is, memory node E is flushed from the main memory to the flash memory and flash node D is loaded into a new memory node D’. After node switching, the record with key 25 is inserted into leaf node D’ that is a new memory node. In this example, two record insertions with keys 35 and 25 are completed in one write operation.

Figure 7 shows the pseudo-code of the insert operation. Lines 2–12 describe the process of node switching during the tree traversal to find the target leaf node for the record insertion with a specific key (key). In lines 2–4, the insert algorithm identifies the node type of the current visiting node while traversing. If the current visiting node CurrentNode is a flash node, CurrentNode is loaded into the main memory as a new memory node. In order to reduce the number of write operations on the flash memory, the old memory node tempNode is written into the flash memory only if its content is changed in lines 7–8. In other words, every memory node does not have to be written into the flash memory if its content is same to the already stored node in the flash memory. Line 13 examines
whether the leaf node is full. In lines 14–21, if the leaf node is full, it checks the key sequences of the leaf node. If the key sequences are sequential, the leaf memory node is directly flushed from the main memory to the flash memory. An empty node is assigned for a memory node and then the record with a key is inserted into the newly allocated memory node. Otherwise, the leaf node is split into two leaf nodes. The leaf node for inserting the record with a key is assigned as a new memory node and the rest of the leaf node is written into the flash memory as a flash node. The parent memory nodes are also updated similar to the insertion of leaf nodes.

Algorithm 1. Insertion (key)

1. CurrentNode <= MemoryNode[1]
   // MemoryNode[1] is the root node
2. for i=2 to height
3. CurrentNode <= ChildNode
4. if CurrentNode is in flash memory
5. tempNode <= MemoryNode[i]
6. swap CurrentNode for MemoryNode[i]
7. if tempNode is dirty
8. write tempNode into flash memory
9. end if
10. end if
11. end for
12. LeafNode <= CurrentNode
13. if LeafNode is full
14. if LeafNode has serial key sequences
15. FlashNode <= LeafNode
16. assign a new empty MemoryNode[height]
17. else
18. split LeafNode into FlashNode and MemoryNode[height]
19. end if
20. store FlashNode into flash memory
21. end if
22. insert key into MemoryNode[height]
23. update the parent nodes

Figure 7. Pseudo code of the insert operation.

3.3. Insert Operation in the Case of Sequential Insertions

Figure 8 shows an example of record insertions with sequential key values in the general B-tree. There are leaf node C with continuous keys 5, 6, 7, and 8. If a record with key 9 is inserted into the B-tree, leaf node C will be split into two nodes due to the node overflow. In this case, three write operations on the flash memory are performed for storing leaf nodes C1, C2, and root node A'. Therefore, frequent node splits result in performance degradation in the flash storage device. In particular, space waste increases in split-leaf nodes (C1 and C2) with continuous keys. For example, leaf node C1 with keys 5 and 6 is flushed into the flash storage device even though there is empty space for more keys. This leads poor space utilization. To address these problems for splitting the leaf node, CB-tree does not split the leaf node when the leaf node with continuous key values overflows. Instead, it flushes only the leaf node fully filled with sequential key values into flash memory and maintains the rest inserted key values in a newly allocated leaf memory node. This approach improves the space utilization and reduces the write operations on the flash storage device.

Figure 8. Sequential insertions of the B-Tree.
Figure 9 shows an example of sequential insertions. When records are sequentially inserted in following key sequences: 30, 31, 32, and 33, the records are inserted into memory node E. If a record with key 34 is inserted, leaf node E will overflow and be split in the B-tree but the leaf node will not be split in CB-tree. Since there are only sequential key values in memory node E, memory node E is directly flushed from the main memory to the flash memory before the record with key 34 is inserted. Then, the record with key 34 is inserted into new memory node F. In this example, five record insertions are performed in one write operation on the flash memory and flash node E is fully filled.

As a result, CB-tree improves write throughput and page utilization by slightly changing the property of the original B-tree where all leaf nodes have between \( \lceil (n - 1) / 2 \rceil \) and \( (n - 1) \) key values \( n \) is the degree of a B-tree.

The benefit in two aspects can be gained from direct flushing the leaf node without node splits. In terms of performance, the number of write operations on the flash memory decreases. Most files and records are sequentially written in the general file systems and the conventional database system [46]. If a large amount of records with sequential key values are inserted, our approach has more performance gains. Since the leaf memory node is almost empty after flushing the leaf node in the case of sequential insertions, large sequential insertions are performed in the leaf memory node without node splits and node switching. Consequently, the write performance increases because many write operations are deferred on flash memory. In terms of page usage, the page utilization of the leaf node increases because the entire space of the flash page is fully filled with records of the leaf node, whereas only half of the flash page is filled if splitting the leaf node.

3.4. Delete Operation

The delete operation in CB-tree is similar to the insert operation. All record deletions are performed in the memory node. First, CB-tree finds a target leaf node to delete a record by visiting the nodes from the root node to the leaf node. If there are flash nodes in visited nodes, CB-tree performs node switching. After swapping the nodes, the record deletion is performed in the leaf memory node and parent nodes are then also updated with the information of the changed leaf node. When the leaf memory node underflows after the record deletion, the leaf memory nodes retain its status in order to delay the write operation on the flash memory. That is, the underflowed leaf node will not be merged with its neighbor leaf node before node switching is performed. Similar to the B-tree, the underflow node is merged with its neighbor node after node switching.

As shown in the top of Figure 10, for example, when a record with key 25 is deleted, the record with key 25 in leaf node D’ is removed without any flash operation because the deleting record stays in the memory node. After the deletion, if records with keys 20 and 21 are deleted, leaf node D’ will underflow. Since leaf node D’ is the memory node, CB-tree does not perform the merge operation for leaf node D’ until node switching occurs in the leaf level. If node switching happens in the leaf level, the rest record with key 19 will be merged with leaf node C and the parent nodes are updated.
The search operation in CB-tree finds a record with a specific key by recursively visiting nodes from the top level to the bottom level regardless of the node type as does in the original B-tree. Its basic algorithm is similar to that of the original B-tree except that cascade memory nodes are loaded into the main memory in advance.

3.5. Memory Node Management

To minimize the overhead of memory resources, the number of entries in a memory node is dynamically allocated in the main memory by growing or shrinking the memory size. Even though the memory node can also have a maximum number of entries in a node, the size of the memory node is assigned to fit the number of key values if there are empty entries in the memory node. CB-tree does not assign the memory space for empty entries in advance to avoid the waste of the unnecessary memory space. When inserting a record, the CB-tree assigns an index entry to store the inserting record in the main memory and then links the newly allocated index entry to the end of the leaf memory node. After connecting the index entry, the contents of memory node and index entry are sorted.

Figure 10 shows the structure of the physical storage in a part of CB-tree where three memory nodes (A, B, and E) and two flash nodes (C and D) exist. If a record with key 25 is inserted, flash node D will be loaded as a new memory node (D’) and flash node D and the memory node E will be flushed from the main memory. Although node E, at the first, was the memory node, it become the flash node after being flushed. After node switching, the record with key 25 is inserted into D’ but there is no main memory space to insert the record. Therefore, CB-tree newly assigns an index entry with key 25 and then links it to the end of memory node D’ as depicted in the middle of Figure 10. Since the key value of the inserted records is the biggest in leaf node D’, it is unnecessary to sort the key values in leaf node D’. After inserting the record, the total size of the allocated memory becomes the size of eight entries by increasing its size.

To prevent the data loss in the case of a sudden power failure, all the cascade memory nodes of CB-tree should be flushed from the main memory to the flash memory periodically. Basically, on a large scale server system, which is rarely turned off and focused on high performance, all the memory nodes are flushed only if the size of the root node increases. As shown in Figure 11, for example, if inserting a record into leaf node E, the root node will increase. At this, all the memory nodes (nodes A, B, and E) are stored into the flash memory before increasing the root node.
In embedded systems that are frequently turned off, a more robust logging technique is required. CB-tree assigns log areas in the main memory and the flash memory to maintain the inserted and deleted records for the power-off recovery. To write the changed logs in one write operation on the flash memory, the maximum size of the log area cannot exceed the size of a page on the flash memory. Before flushing memory nodes, it always stores all the inserted and deleted records (including a key value, a record address, and an operation type) into log areas in the main memory and flash memory in FIFO order. In order to avoid reading the previous log data stored in the flash memory when writing the logs, CB-tree stores logs in both the main memory and flash memory. Therefore, the CB-tree only refers the log data in the main memory when recovering its index structure. When the log area becomes full, all the cascade memory nodes are flushed into flash memory and then all the logs stored in both areas are removed.

For recovering the index structure, CB-tree first finds the final flushed root node and the content of the log area stored in the flash memory. The found root node is used for a new root memory node and then all the records in the log area are inserted and deleted in stored order.

For example, assume that the sudden power failure occurs after flushing all the cascades nodes (nodes A, B, and E as depicted in Figure 11) and inserting three records (following key sequence: 7, 1, and 9). There are three records, which are stored in the inserted order, of the log area in the flash memory. CB-tree uses root node A that is finally flushed as the root memory node and then inserts three records by referring the inserted order (7, 1, and 9) of the log area into the index structure. Consequently, CB-tree is perfectly recovered after inserting all the records stored of the log area.

Compared to structure-modified B-trees, CB-tree has the overhead for managing cascade memory nodes. As mentioned in Section 3.1, the number of cascade memory nodes is equal to the height of CB-tree. Therefore, the maximum space overhead of CB-tree is ‘the size of a node x the height of CB-tree’. However, to minimize the space overhead, CB-tree does not assign the memory space in advance. The space for cascade memory nodes is dynamically allocated by the entry unit of a node as depicted in the middle of Figure 10.

4. System Analysis

To estimate the performance of CB-tree, we analyze the behaviors of CB-tree and the B-tree. For ease of analysis, we assume the following:

- There are enough free blocks to perform the insert, delete, and search operations without any garbage collection in the flash storage device because it is difficult to know how the flash storage device stores data internally.
- The cost of the insert, delete, or search operation is calculated without any operation cost that occurs in the main memory. That is, we only consider the cost of the operations that take place in the flash memory.

![Figure 11. Example of flushing memory nodes.](image-url)
Table 2 summarizes the notation used in our analysis.

### Table 2. Notation summary.

| Symbols | Definition |
|---------|------------|
| $C_R$  | the consumed time to read a tree node in the flash memory |
| $C_W$  | the consumed time to write a tree node in the flash memory |
| $H$    | the height of the B-tree |
| $n$    | the number of inserted records |
| $m$    | the number of maximum entries per node |
| $R_B$  | the cost of the search operation for B-tree |
| $R_C$  | the cost of the search operation for CB-tree |
| $W_B$  | the cost of the insert operation for B-tree |
| $W_C$  | the cost of the insert operation for CB-tree |
| $T_B$  | the total cost of the insert operations for B-tree |
| $T_C$  | the total cost of the insert operations for CB-tree |

The search operation of the B-tree requires the same number of the read operations as the tree height $H$ because the target node with the desired record is always traversed from the root node to the leaf node. Therefore, we can obtain the cost of the search operation of the B-tree $R_B$ as follows:

$$R_B = C_R H$$  \hfill (1)

Compared to the B-tree, the node split causes updating for the parent node is less performed in CB-tree. As a result, the tree height of CB-tree is equal or less than the B-tree but we assume that the height of CB-tree is the same as the height of the B-tree $H$ for convenience of the analysis. Generally, the search operation of the CB-tree is also affected by the tree height because CB-tree traverses all paths from the root node to the leaf node for searching the desired record regardless of the node type. If the visited nodes are all the cascade memory nodes, there is no read operation on the flash memory. Otherwise, all the visited nodes except the root node are read on the flash memory. Therefore, in the worst case, we can obtain the cost of the search operation of the CB-tree $R_C$ as follows:

$$R_C = C_R (H - 1)$$  \hfill (2)

From (1) and (2), we determined that CB-tree can quickly find a record than B-tree. However, in the practice system, some nodes of the B-tree also stay in the main memory to improve the search performance. At this time, it is hard to estimate the performances of two B-trees because we do not know which cache strategies are applied in the system. If LRU is adopted, the search performance of B-tree can be similar to that of the CB-tree.

The insert operation of the B-tree requires a search operation to find a target leaf node and a write operation to insert a record. Therefore, we can obtain the cost of the insert operation of the B-tree $W_B$ as follows:

$$W_B = C_R H + C_W$$  \hfill (3)

The insert operation of CB-tree requires a search operation to find a target leaf node and write operations for node switching. If the visited nodes are all the cascade memory nodes, there is no operation on the flash memory. At this time, the cost of the insert operation of CB-tree is equal to zero. However, if the visited nodes are all the flash nodes, node switching occurs at every level except the root node on the flash memory. In this worst case, we can obtain the cost of the insert operation of CB-tree $W_C$ as follows:

$$W_C = (C_R + C_W) \cdot (H - 1)$$  \hfill (4)

From the above analysis, we knew that the cost of the insert operation of the B-tree is always uniform and the cost of the insert operation of CB-tree is variable according to the visited node type.
In the worst case, the cost of the insert operation of CB-tree is much higher than that of the B-tree. However, in the real system, since most of the write patterns (about 80–90%) are sequential and most of the visited nodes are memory nodes, node switching rarely occurs in CB-tree. Therefore, we expected that the insert operation is faster than that of B-tree in the real system.

In order to estimate the write performance for sequential data pattern, we suppose that n records are sequentially inserted in continuous key order and a node of the tree index structure can store maximum m entries. If n records are sequentially inserted in the B-tree, the total cost of the insert operations $T_B$ is obtained as follows:

$$T_B = n(C_RH + C_W)$$

Furthermore, the B-tree requires additional write operations for the node split when the leaf node overflows. At this time, at least two write operations are invoked for storing a new split-leaf node and updating the parent node. For easy estimation, we suppose that updating the parent node is always performed from the leaf node to the root node. That is, the cost of updating the parent nodes is $CWH$. Since the half of the leaf node is always filled after splitting the leaf node if records are sequentially inserted, the leaf node is split whenever $m/2$ records are inserted. Therefore, in the case of the sequential insertions, we can obtain the total cost of the insert operations $T_B$ as follows:

$$T_B = n(C_RH + CW) + 2n/m \cdot CWH$$

In CB-tree, when records are inserted in a sequential key order, the leaf node is not split. Also, the cost of the search operation to find a leaf node equals to zero because the visited nodes are all the cascade memory nodes. At this time, the leaf memory node is directly written into the flash memory whenever $m$ records are inserted. Since flushing the leaf node invokes one write operation (i.e., the cost is $CW$) on the flash memory, the total cost of the insert operations of CB-tree is obtained as follows:

$$T_C = n/m \cdot CW$$

Similar to the B-tree, CB-tree also requires additional write operations for updating parent nodes. For fair analysis, we assume that all the cascade memory nodes are also stored in the flash memory when flushing the leaf node in CB-tree. Additionally, log writing except the basic insert operation is always performed for data recovery. Log writing invokes only a write operation (i.e., the cost is $CW$) whenever inserting a record. Therefore, we can obtain the total cost of the insert operations $T_C$ including the cost for log writing when n records having sequential key values are inserted in CB-tree as follows:

$$T_C = n/m \cdot CWH + nCW$$

From (6) and (8), since the total cost of two B-trees always becomes $T_B > T_C$, we determined that CB-tree quickly creates the index structure than the B-tree does when records are inserted in sequential key order. However, in the real system, all records are not inserted sequentially but most of the write patterns are sequential. Additionally, since CB-tree less splits the leaf node compared to the B-tree, CB-tree updates the parent node less than the B-tree does. Therefore, we determined that the creation time of CB-tree is faster than that of the B-tree. In the following section, we show the practical results such as this analysis through various experiments with real workloads.

Table 3 summarizes the evaluation notation used in B-Tree and CB-Tree.
the total cost of the insert operations (node split pattern) $T_B = n(C_R H + C_W)$ $T_C = n/m \cdot C_W$

5. Performance Evaluation

To evaluate the performance of CB-tree, we implemented CB-tree and various flash-aware B-trees on Flash SSD environments. For measuring the effect of memory buffer, CB-tree was experimented with the original B-tree without any buffer as a base algorithm, Lazy update B+-tree (LU-tree), IBSF, and AS-tree using the same sized write buffer. Also, for assessing the performance of structure-modified B-trees, Wandering-tree as a base algorithm, μ*-tree, dIPL B+-tree, and LSB-tree were implemented. Every node in each tree consisted of 128 entries that contain a key to find a child node and a pointer to the child node. For the comparison, we measured the number of flash operations, creation time, and retrieval time.

The experiments for buffered B-trees are performed on SAMSUNG S470 64GB MLC SSD (Figure 12a) running in Linux kernel 2.6 with Intel Core i5-2550 CPU and 8GB DDR3 memory. The experiments for structure-modified B-trees are performed on OpenSSD platform (Figure 12b) [47] that can modify its internal mapping algorithm. OpenSSD platform contains the ARM7TDMI-S core, 64MB mobile SDRAM, and two 32GB SAMSUNG K9LCG08U1M MLC NAND modules.

![Flash storage devices](image_url)

Table 3. Notation for evaluation of B-Tree and CB-Tree.

| Categories | B-Tree | CB-Tree |
|------------|--------|---------|
| the cost of the search operation | $R_B = C_R H$ | $R_C = C_R (H - 1)$ |
| the total cost of the insert operations (sequential data pattern) | $W_B = C_R H + C_W$ | $W_C = (C_R + C_W) \cdot (H - 1)$ |
| the total cost of the insert operations (node split pattern) | $T_B = n(C_R H + C_W)$ | $T_C = n/m \cdot C_W$ |

Table 4 shows I/O performance of the flash SSD environment used in our experiments. As depicted in this table, the performance of the SSD product is faster than that of the OpenSSD platform because most SSD products employ parallel writing and an internal buffer for the performance improvement. In contrast, OpenSSD platform simply offers to apply custom mapping, indexing, and caching algorithm for measuring the performance of the flash SSD. Therefore, we used MLC SSD for measuring the realistic performance of buffer-based B-trees. We also used OpenSSD platforms for measuring the performance of structure-modified B-trees because we can manipulate the physical address for implementing structure-modified B-trees.
Table 4. Performance of flash storage devices.

| I/O Type           | SSD Product | OpenSSD |
|--------------------|-------------|---------|
| Sequential Read    | 225 MB/s    | 66.5 MB/s |
| Sequential Write   | 66.5 MB/s   | 22 MB/s  |
| Random Read (4KB)  | 16.5 MB/s   | 10.5 MB/s |
| Random Write (4KB) | 20 MB/s     | 2.5 MB/s  |

5.1. Comparison with Buffer-Based B-Trees

We first evaluated the performance of buffer-based B-trees for large scale file systems and database systems. To estimate realistic values, we used traces collected by SNIA (Advancing Storage and Information Technology) [48]. Table 5 shows the number of I/O operations extracted in traces. MSR Cambridge data are the 1-week block I/O traces collected in the enterprise server of Microsoft Research in Cambridge. Exchange data are traces collected for an Exchange Server for a duration of 24 h. The TPC-C and TPC-E data are TPC-C and TPC-E benchmark traces collected at Microsoft. Trace patterns of MSR Cambridge and TPC-C benchmark show the number of write requests similar to the number of read requests. Exchange server invokes many write requests due to sending many e-mails whereas the TPC-E benchmark writes data once and reads data many times.

Table 5. The number of operation requests in each trace.

| Server       | Write Requests | Read Requests |
|--------------|----------------|---------------|
| MSR Cambridge| 1,877,535      | 1,625,346     |
| Exchange     | 3,921,312      | 1,789,471     |
| TPC-C        | 1,263,387      | 1,836,124     |
| TPC-E        | 123,562        | 1,313,596     |

To compare performances among buffer-based B-trees, we measured creation time and retrieval time by performing write requests and read requests from the above real traces on an MLC flash SSD. The original disk-based B-tree is used as a base algorithm without any memory buffer. For fair evaluation, the optimal buffer size is first obtained by increasing the page size. When the buffer size equals eight pages, the creation time shows the best performance. As a result, the buffer size is fixed to eight pages. Even though the buffer size of CB-tree dynamically increases as the tree height increases, its buffer size does not exceed the size of four pages because the heights of index structures created from the above traces become at most four.

Figure 13 shows the creation time when performing each write requests as shown in Table 5. Overall, IBSF and CB-tree are quickly built compared to the other index structure because they do not reorder the records in the memory buffer. In the cases of Figure 13a,c, CB-tree is built 27.6–36.4% faster than IBSF. Since their write requests show sequential patterns, the CB-tree reduces a large amount of node splits for the leaf node. Although the TPC-E benchmark has a more random pattern than the TPC-C benchmark does, the CB-tree is built 22.8% faster compared to IBSF because IBSF requires many seeks to find records to be inserted into the same logical node in the memory buffer.
Most of buffer-based B-trees including IBSF reduce the number of write operations for leaf node updates by simply employing a write buffer. However, since they do not consider flash operations for updating parent nodes and merging or splitting leaf nodes, the performance degradation may occur. In CB-tree, we exploit cascade memory nodes to avoid many updates for parent nodes and node splits for the leaf node in the case of sequential insertions. As a result, we confirmed that CB-tree quickly builds its index structure compared to the other buffer-based B-trees.

Figure 14 shows the total retrieval time when the trees perform each read request as shown in Table 5. The B-tree without any buffer performs the same number of read operations as its tree height in every read request. Since LU-tree, IBSF, and AS-tree store some records in the main memory, these records are also used in retrieval operations. Before traversing nodes to find a record, they first search the record in the buffer and then visit the nodes in the flash memory. CB-tree traverses nodes to find a record without regard to the node type (either a memory node or a flash node). Although Buffer-based B-trees except CB-tree similarly find records compared to the B-tree, the search operation of the CB-tree is much faster than the others because most accessed internal nodes are used for the memory nodes.

|                     | B-Tree | LU-Tree | IBSF  | AS-Tree | CB-Tree |
|---------------------|--------|---------|-------|---------|---------|
| MSR Cambridge       | 214,803| 16,172  | 12,494| 13,208  | 9796    |
| Exchange            | 513,568| 60,924  | 54,005| 55,350  | 39,054  |
| TPC-C Benchmark     | 154,263| 49,737  | 46,064| 48,758  | 29,288  |
| TPC-E Benchmark     | 14,241 | 953     | 768   | 812     | 592     |

Figure 13. Creation time in buffer-based B-trees (total elapsed time).
Most of buffer-based B-trees including IBSF reduce the number of write operations for leaf node updates by simply employing a write buffer. However, since they do not consider flash operations for updating parent nodes and merging or splitting leaf nodes, the performance degradation may occur. In CB-tree, we exploit cascade memory nodes to avoid many updates for parent nodes and node splits for the leaf node in the case of sequential insertions. As a result, we confirmed that CB-tree quickly builds its index structure compared to the other buffer-based B-trees.

Figure 14 shows the total retrieval time when the trees perform each read request as shown in Table 5. The B-tree without any buffer performs the same number of read operations as its tree height in every read request. Since LU-tree, IBSF, and AS-tree store some records in the main memory, these records are also used in retrieval operations. Before traversing nodes to find a record, they first search the record in the buffer and then visit the nodes in the flash memory. CB-tree traverses nodes to find a record without regard to the node type (either a memory node or a flash node). Although Buffer-based B-trees except CB-tree similarly find records compared to the B-tree, the search operation of the CB-tree is much faster than the others because most accessed internal nodes are used for the memory nodes.

As shown in Figure 14d, in TPC-E where the number of the read requests is larger than the number of write requests, CB-tree finds data 64.8% faster than the B-tree does. In every case, CB-tree finds data about 50% faster compared to the B-tree. Even though Buffer-based B-trees uses more memory resources than CB-tree does, CB-tree efficiently reduces the number of the read operations than the others because they simply employ main memory as the write buffer and CB-tree uses main memory to delay updating parent nodes in a cascade manner.

Figure 15 shows the counts of flash operations in buffer-based B-Trees. In every case for the read and write requests, the number of all the operations of CB-Tree is decreased than other index structures. Therefore, CB-Tree shows outperform than other buffer-based B-Trees.
For evaluating the performances of structure-modified B-trees, we used OpenSSD platform for adopting the custom mapping algorithm (i.e., FTL) because structure-modified B-trees need direct mapping to a physical address. In addition, OpenSSD can count the internal flash operations (read/write/erase operation). In order to compare the creation time of structure-modified B-trees, we measured the number of the read/write/erase operations and the total elapsed time by inserting 50,000 records with various key sequence ratio. If the ratio of key sequences is equal to 0%, the records with keys that are sorted in ascending order are sequentially inserted. In contrast, if the ratio is equal to 100%, the records with the key that are randomly generated are inserted. In every case, CB-tree finds data 64.8% faster than the B-tree does. In every case, CB-tree finds data about 50% faster compared to the B-tree. Even though Buffer-based B-trees uses more number of write requests, CB-tree uses main memory resources than CB-tree does, CB-tree efficiently reduces the number of the read operations than the others because they simply employ main memory as the write buffer and CB-tree uses main memory to delay updating parent nodes in a cascade manner.

Figure 16 shows the counts of flash operations when records are inserted. The wandering tree is used as a basic B-tree for the flash memory. In these figures, B-tree, dIPL B, and CB(log) mean Wandering-tree, dIPL B+-tree, and CB-tree with log writes, respectively. Except for CB-tree, the other trees require a similar number of the read operations in every case because finding the leaf node to insert data affected by the tree height. However, since sometimes read operations are not performed to find the leaf node for the insertion according to the ratio of key sequences, CB-tree needs less read operations than the other trees.
As depicted in Figure 16a, the search operations on the flash memory are not invoked in CB-tree because the visited nodes for inserting records are all memory nodes. Erase operations on the flash memory are not also invoked because fewer pages are used to store only the memory nodes compared to the other trees. Also, fewer write operations are performed because the nodes in the CB-tree are written in the batch process only if the leaf memory node becomes full and the other trees always write a record in the flash memory whenever the record is inserted. Therefore, in the case of ratio 0%, the CB-tree more quickly creates the index structure than any other tree.

Figure 16b shows the operation counts in the case of the 50% ratio. The operation counts in Wandering tree are similar to the case of 0% ratio. The operation counts in dIPL B+, μ*-tree, LSB-tree, and CB-tree trees increases compared to the case of ratio 0%. Interestingly, in μ*-tree, the number of the read operations increases but the number of write operations and the number of erase operations decrease because fewer node splits are invoked by employing many new pages for inserting records. In the dIPL B+-tree and LSB-tree, since more complicated merges between the log area and the data area occur, more flash operations are performed. CB-tree also needs more flash operations to swap the nodes between the memory node and the flash node. Even though more flash operations are invoked in the CB-tree, the number of flash operations in the CB-tree is much less than the other trees.

Similarly, as the ratio goes to 100%, the flash operation counts increase in every tree except the μ*-tree as depicted in Figure 16c. Since the μ*-tree uses more pages to insert records without node splits, its operation counts are reduced. From these experiments, we confirmed that CB-tree efficiently reduces the number of write operations by keeping the updated records in the memory node.
For evaluating the overhead of the log writing, we additionally measured the operation counts with the log writing. Through the results of Figure 16a–c, the number of its write operations increases as the number of inserted records increases since additional write operations for the logs are performed in the logging process. As the ratio goes to 100%, the number of flash operations of CB-tree and CB(log) becomes similar due to node switching. Although the number of write operations increases, the overall performance is better than the other B-tree index structures because the number of the read operations in CB-tree is much smaller than the other index structure. This experimental result is shown in the following Figure 17.

![Figure 17. Creation time in structure-modified B-Trees.](image)

Figure 17 shows the creation time in structure-modified B-trees. Their patterns of results are similar to the number of flash operations in Figure 16. As the ratio goes to 100% (in Figure 17c), Wandering tree quickly builds the index structure because less node splits are performed than the case of the 0% ratio (in Figure 17a). However, regardless of the ratio, Wandering tree is much slower than the
other trees because it stores all the updated parent nodes when inserting a record. As you can see the results in Figure 16, μ*-tree performs more flash operations than diPL B+-tree and LSB-tree. In μ*-tree, the number of read operations is much more than the number of write operations. Unusually, the total creation times of these three trees in various ratios are almost similar because the read operation is faster than the other flash operations.

As depicted in Figure 17a, CB-tree is quickly built compared to the other B-trees because no read and erase operations are invoked on the flash memory. Although CB-tree is slowly built because much node switching occurs between the memory node and the flash node in CB-tree as the ratio increases, its creation performance is always faster than any other tree. Additionally, even though CB-tree performs log writing, its performance is still 22.6–41.8% better than the other structure-modified B-trees because of only the number of write operations for the logs increases. From the results of Figure 17a–c, we confirm that CB-tree quickly builds the index structure compared to the other structure-modified B-trees.

Figure 18 shows the average search time when finding a record in the index structure already built in Figure 15. In all ratios, Wandering tree finds a record in a similar time because its search operation is affected by only the tree height. As shown in Figure 18a, dIPL B+-tree and LSB-tree find a record similar to Wandering tree because records rarely exist in the log area in the case of the 0% ratio. However, as depicted in Figure 18c, the search performance decreases as the ratio goes to 100% because searching is invoked in two places the log area and the data area. Although the search operation of the μ*-tree is very slow because of the frequent leaf node splits causing a rapid increment of the tree height, its search performance is uniform in all ratios. CB-tree quickly finds a record compared to the other B-trees because some internal nodes stayed in the memory nodes. Through the experiment, we confirmed that CB-tree quickly finds a record compared to the other structure-modified B-trees.

![Figure 18. Average search time in structure-modified B-trees.](image-url)
As shown in Table 6, we measured the number of allocated pages on the flash memory when the index structures are built. Based on Wandering tree, dIPL B+-tree and µ*-tree allocates many pages on the flash memory. The reason is that dIPL B+-tree needs the log area to write the changes of the leaf node and the µ*-tree needs many pages due to its unique page layout. As the ratio goes to 0%, LSB-tree and CB-tree need fewer pages than any other B-trees because they do not perform the node split in the case of sequential insertions. Through the results, we found that CB-tree requires fewer pages to create the index structure compared to the other structure-modified B-trees.

Table 6. The number of allocated pages on the flash memory

| Ration | Wandering Tree | dIPL B+-Tree | µ*-Tree | LSB-Tree | CB-Tree |
|--------|----------------|--------------|---------|----------|---------|
| 0%     | 794            | 1574         | 1665    | 400      | 398     |
| 50%    | 691            | 1720         | 2054    | 469      | 493     |
| 100%   | 563            | 1834         | 2219    | 563      | 566     |

6. Conclusions

It may lead performance degradation to apply the original B-tree on the flash storage device without any changes because NAND flash memory has the characteristic that it has to perform slow block erasures before overwriting data on a prewritten page. Therefore, various techniques have been proposed for improving the performance of B-tree on flash memory. Generally, these flash-aware B-tree index structures are classified into two groups.

The key idea of the first group is to employ the memory buffer to improve the write throughput. The main advantage of this approach is much faster than the other group for read and write performance. However, they suffer from the high cost of maintaining the memory buffer and the risk of data loss in case of sudden power failure. On the other hand, the second group has B-tree variants that modify B-tree’s node structure to avoid in-place updates. The main advantage of this approach is more reliable than B-trees in the first group. They also have additional advantage that they just use small memory resources. However, the main disadvantage of B-trees in the second group is that their write performance is generally much lower than that of the first group.

The design goal of CB-tree is to improve the sequential write performance and also maintain reliability as do B-trees in the second group. As shown in various experiments, CB-tree achieves the goal by employing cascade memory nodes. CB-tree improves the write throughput by delaying the write of updated nodes into the flash memory. In particular, when records are sequentially inserted in continuous key order, it enhanced the page utilization of the leaf node by using the entire space of the page as a leaf node and reduced additional write operations by not splitting leaf nodes. Through mathematical analysis as well as various experiments, we have also shown that CB-tree always yields better performance compared to the related.

To sum up, in the creation time and search time with real traces, CB-tree outperforms the buffer-based B-trees by up to about 35%. Also, CB-tree performing the log writing creates the index structure 22.6–41.8% faster than the structure-modified B-trees.

CB-tree also has the space overhead because it employs additional memory nodes that are so-called cascade memory nodes for improving the performance. However, in order to minimize the space overhead of cascade memory nodes, CB-tree does not assign the memory space in advance but dynamically allocates it by the entry unit for cascade memory nodes.

The current version of CB-tree employs a rather simple recovery mechanism for sudden power failure as mentioned in Section 3.5. However, to maintain a high reliability on CB-tree, it needs a more elaborate algorithm for power-off recovery at various granularity levels. Therefore, as our future work, we are currently studying to efficiently manage logs into the flash memory at various granularity levels for power-off recovery.
Author Contributions: Conceptualization, D.H.L. and B.-K.K.; methodology, B.-K.K.; software, B.-K.K.; validation, B.-K.K., G-W.K., and D.H.L.; formal analysis, B.-K.K. and D.H.L.; investigation, G.-W.K.; resources, D.H.L.; data curation, B.-K.K.; writing—original draft preparation, B.-K.K.; writing—review and editing, G.-W.K. and D.H.L.; visualization, B.-K.K.; supervision, D.H.L.; project administration, D.H.L.; funding acquisition, D.H.L. All authors have read and agreed to the published version of the manuscript.

Acknowledgments: This research was funded by Basic Science Research Program through the National Research Foundation of Korea (NRF), grant number NRF-2016R1D1A1A09918271.

Conflicts of Interest: The authors declare no conflict of interest.

References
1. Grupp, L.M.; Davis, J.D.; Swanson, S. The Bleak Future of NAND Flash Memory. In Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST), San Jose, CA, USA, 15–17 February 2012; p. 2.
2. Harari, E. Flash Memory—The Great Disruptor! In Proceedings of the 2012 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, USA, 19–23 February 2012; pp. 10–15.
3. Li, Y.; Quader, K.N. NAND Flash Memory: Challenges and Opportunities. Computer 2013, 46, 23–29. [CrossRef]
4. Samsung Electronics. May 2010. Available online: https://www.alldatasheet.com/ (accessed on 12 December 2019).
5. Gal, E.; Toledo, S. Algorithms and Data Structures for Flash Memories. ACM Comput. Surv. (CSUR) 2005, 37, 138–163. [CrossRef]
6. Chung, T.-S.; Park, D.-J.; Park, S.; Lee, D.-H.; Lee, S.-W.; Song, H.-J. A Survey of Flash Translation Layer. J. Syst. Archit. 2009, 55, 332–343. [CrossRef]
7. Ma, D.; Feng, J.; Li, G. A Survey of Address Translation Technologies for Flash Memories. ACM Comput. Surv. (CSUR) 2014, 46, 36. [CrossRef]
8. Comer, D. Ubiquitous B-Tree. ACM Comput. Surv. (CSUR) 1979, 11, 121–137. [CrossRef]
9. Batory, D.S. B+ Trees and Indexed Sequential Files: A Performance Comparison. In Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD ’81), Ann Arbor, MI, USA, 29 April–1 May 1981; pp. 30–39.
10. Reiserfs. May 2019. Available online: http://reiser4.wiki.kernel.org/ (accessed on 12 December 2019).
11. XFS. May 2019. Available online: http://xfs.org/ (accessed on 13 December 2019).
12. Btrfs. May 2019. Available online: http://btrfs.wiki.kernel.org/ (accessed on 17 December 2019).
13. Postgresql. May 2019. Available online: http://www.postgresql.org/ (accessed on 27 October 2019).
14. Mysql. May 2019. Available online: http://www.mysql.com/ (accessed on 11 August 2019).
15. Sqlite. May 2019. Available online: http://www.sqlite.org/ (accessed on 11 August 2019).
16. Ban, A. Flash File System. U.S. Patent No. 5,404,485, 4 April 1995.
17. Kim, G.Y.; Urgaonkar, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings; Department of Computer Science and Engineering, The Penn-sylvania State University: State College, PA, USA, 2008.
18. Shin, I. Light Weight Sector Mapping Scheme for NAND-Based Block Devices. IEEE Trans. Consum. Electron. 2010, 56, 651–656. [CrossRef]
19. Ma, D.; Feng, J.; Li, G. LazyFTL: A Page-Level Flash Translation Layer Optimized for NAND Flash Memory. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; pp. 1–12.
20. Ban, A. Flash File System Optimized for Page-Mode Flash Technologies. U.S. Patent No. 5,937,425, 10 August 1999.
21. Shinohara, T. Flash Memory Card with Block Memory Address Arrangement. U.S. Patent No. 5,905,993, 18 May 1999.
22. Estakhri, P.; Ganjuei, A.R.; Iman, B. Moving Sectors within a Block of Information in a Flash Memory Mass Storage Architecture. U.S. Patent No. 6,145,051, 7 November 2000.
23. Estakhri, P. Block Management for Mass Storage. U.S. Patent No. 6,567,307, 20 May 2003.
24. Kim, J.; Kim, J.M.; Noh, S.H.; Min, S.L.; Cho, Y. A Space-Efficient Flash Translation Layer for Compactflash Systems. IEEE Trans. Consum. Electron. 2002, 48, 366–375.
25. Lee, S.-W.; Park, D.-J.; Chung, T.-S.; Lee, D.-H.; Park, S.; Song, H.-J. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *ACM Trans. Embed. Comput. Syst.* 2007, 6, 18. [CrossRef]

26. Lee, H.-S.; Yun, H.-S.; Lee, D.-H. HFTL: Hybrid Flash Translation Layer Based on Hot Data Identification for Flash Memory. *IEEE Trans. Consum. Electron.* 2009, 55, 2005–2011. [CrossRef]

27. Wu, C.-H.; Kuo, T.-W.; Chang, L.-P. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. *ACM Trans. Embed. Comput. Syst.* 2007, 6, 19. [CrossRef]

28. On, S.T.; Hu, H.; Li, Y.; Xu, J. Flash-Optimized B+-Tree. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT ’07)*; New York, NY, USA: ACM, 2007; pp. 144–153.

29. Na, G.; Moon, B.; Lee, S.-W. IPL B-Tree for Flash Memory Database Systems. *J. Inf. Sci. Eng.* 2011, 27, 111–127.

30. Na, G.-J.; Lee, S.-W.; Moon, B. Dynamic In-Page Logging for B+-Tree Index. *IEEE Trans. Knowl. Data Eng.* 2012, 24, 1231–1243. [CrossRef]

31. Agrawal, D.; Ganesan, D.; Sitaraman, R.; Diao, Y.; Singh, S. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *Proc. VLDB Endow.* 2009, 2, 361–372. [CrossRef]

32. Wang, J.; Lam, K.-Y.; Chang, Y.-H.; Hsieh, J.-W.; Huang, P.-C. Block-Based Multi-Version B-Tree for Flash-Based Embedded Database Systems. *IEEE Trans. Comput.* 2015, 64, 925–940. [CrossRef]

33. Roselli, D.S.; Lorch, J.R.; Anderson, T.E. A Comparison of File System Workloads. In Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, USA, 18–23 June 2000; pp. 41–54.

34. Leung, A.W.; Pasupathy, S.; Goodson, G.R.; Miller, E.L. Measurement and Analysis of Large-Scale Network File System Workloads. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 22–27 June 2008; Volume 1, p. 5-2.

35. Kim, B.-k.; Lee, D.-H. LSF: A New Buffer Replacement Scheme for Flash Memory-Based Portable Media Players. *IEEE Trans. Consum. Electron.* 2013, 59, 130–135. [CrossRef]

36. Lee, S.-W.; Moon, B.; Park, C.; Kim, J.-M.; Kim, S.-W. A Case for Flash Memory SSD in Enterprise Database Applications. In Proceedings of the 2008 ACM SIG-MOD International Conference on Management of Data, Vancouver, BC, Canada, 10–12 June 2008; pp. 1075–1086.
47. The OPENSSD Project. May 2016. Available online: http://www.openssd-project.org/ (accessed on 27 October 2019).

48. SNIA. Advancing Storage and Information Technology. May 2016. Available online: http://www.snia.org/ (accessed on 27 October 2019).