International Journal of Internet, Broadcasting and Communication Vol.8 No.3 12-24 (2016)
http://dx.doi.org/10.7236/IJIBC.2016.8.3.12

IJIBC 16-3-2

A Flash-based B+-Tree using Sibling-Leaf Blocks for Efficient Node Updates and Range Searches

Seong-Chae Lim

Dept. of Computer Science Dongduk Women's University, Korea
sclim@dongduk.ac.kr

Abstract

Recently, as the price per bit is decreasing at a fast rate, flash memory is considered to be used as primary storage of large-scale database systems. Although flash memory shows off its high speeds of page reads, however, it has a problem of noticeable performance degradation in the presence of increasing update workloads. When updates are requested for pages with random page IDs, in particular, the shortcoming of flash tends to impair significantly the overall performance of a flash-based database system. Therefore, it is important to have a way to efficiently update the B+-tree, when it is stored in flash storage. This is because most of updates in the B+-tree arise at leaf nodes, whose page IDs are in random. In this light, we propose a new flash B+-tree that stores up-to-date versions of leaf nodes in sibling-leaf blocks (SLBs), while updating them. The use of SLBs improves the update performance of B-trees and provides the mechanism for fast key range searches. To verify the performance advantages of the proposed flash B+-tree, we developed a mathematical performance evaluation model that is suited for assessing B-tree operations. The performance comparisons from it show that the proposed flash B+-tree provides faster range searches and reduces more than 50% of update costs.

Keywords: *flash memory, B+-tree index, database, buffer pool*

1. Introduction

During the past decades, we have seen drastic advances in the technology of digital storage devices. Especially, the NAND-type flash memory has drawn much attention because of its performance advantages over the traditional hard disk drives (HDDs) [1, 4]. Since flash memory has many desirable features such as robust shock resistance and low power consumption, it has superseded HDDs in the case of low-end computing devices such as laptops and smart phones [2]. Recently, as the price per bit is decreasing at a fast rate, there is a growing demand for high-end flash-based DBMSs that capitalizes on very fast speeds of random reads in flash [1]. In this context, much research has been done to develop flash-aware B+-tree algorithms that are suitable for flash storage [3, 5, 7- 9, 11-16].

Manuscript Received: Jun. 5, 2016 / Revised: Jun. 15, 2016 / Accepted: Jul. 5, 2016
Corresponding Author: sclim@dongduk.ac.kr
Tel:+82-2-940-4589, Fax: +82-2-940-4170
Dept. of Computer Science Dongduk Women's University, Korea

Because the B-tree index was originally invented for its use in HDD storage, the B-tree index may suffer from performance degradations because of any unique I/O feature of flash memory, if it is used for flash storage [7, 11, 13]. As matter of fact, the absence of in-place-update in flash memory is a distinct I/O feature, which is not found in the traditional HDD storage. If bit cells in a page *P* are electronically charged for writing data on *P*, then they should be cleaned before page *P* is updated with new data. In flash storage, such cleaning (or erasing) operation is performed in the unit of a flash block, which is comprised of 32 or 64 pages. The different units of data writing and before-write cleaning significantly increase the I/O cost for in-place updating of pages [9, 13, 14]. For that reason, in-place-update of a page is not usual in flash storage. Instead, flash storage utilizes a special-purpose firmware, called the FTL (Flash Translation Layer).

The FTL is responsible for doing hidden actions for out-of-place updates and garbage collection to reclaim invalidated data pages [1, 11]. Although there have been many efforts to develop efficient FTL algorithms that work against frequent updates, it has been reported that updates on random pages inevitably hurt the I/O throughput of flash-based storage because of a number of the occurrences of *full merges* [11-14]. When it comes to the update patterns in a B+-tree, most of them seem to arise on leaf nodes. Note that other type nodes, i.e., internal nodes, are updated only when tree reconstruction is executed for handling the overflow or underflow of leaf nodes. Since a populated B+-tree has many leaf nodes scattered widely across storage space, update operations in the B+-tree usually entail random page updates. Therefore, a B+-tree with frequent update operations tends to impair storage performance because of I/O overheads of full merging in the FTL.

To avoid the excessive full-merge overheads, our B+-tree groups the leaf nodes according to their parent nodes and stores the same group of leaf nodes in a block, called the *sibling-leaf block* (SLB). If a leaf node *N* in SLB *S* needs updating, then *N*'s new version is saved in *S*, along with its original version. Since the page address of *N* is changed due to its update, we also need to update the parent of *N* to reflect *N*' new address in that parent node. That second update gives rise to the third another update on the grand parent of *N*. As a result, the updates recursively go upward to the root node, thereby increasing B+-tree update costs significantly [7, 12].

To prevent the cascading updates along a B+-tree path, we do not store physically the nodes at the leaf's parent level. Instead, that level's node is generated based on the index entries saved in its child (leaf) nodes. Since the leaf nodes with the same parent exist in the same SLB, the I/O cost for reading them is not expensive. Since the DBMS can cache the virtual nodes in the buffer pool, moreover, the actual overhead for the use of virtual nodes seems to be negligible. For that reason, our B+-tree can offset the overhead for generating virtual nodes by reducing I/O costs paid for updating leaf nodes. To verify the performance advantages of the proposed flash B+-tree, we performs performance evaluations based on a mathematical cost model. From the evaluation results, we can see that the proposed B+-tree provides fast range search speeds as well as low update costs.

This paper is organized as follows. In Section 2, we introduce the problem of garbage collection and address the earlier algorithms for the flash B+-tree. We propose a new flash B+-tree scheme in Section 3, and evaluate its performance in Section 4. Lastly, we conclude this paper in Section 5.

2. Backgrounds

2.1 Garbage Collection

Since a page cannot be overwritten in flash memory, the page should be electronically cleaned before it is updated in place. The smallest granularity of such cleaning is a block and the block is composed of 32 or 64

pages. Because of such H/W features of flash memory, the cost for an in-place-update of a page is prohibitively high [18-20]. For this reason, an update request in flash storage is handled through an out-of-place update. To hide the rather complex mechanism of out-of-place updates, a firmware of the FTL (Flash Translation Table) is employed [1, 3, 5].

When the out-of-place updates consume almost all of the clean pages in flash storage, the FTL initiates actions for garbage collection so as to reclaim clean pages. Since I/O overheads for garbage collection deeply impact the performance of flash storages, many research has been done for supporting efficient garbage collection in the FTL [5]. The FTL algorithms for garbage collection can be classified according to the ways how the FTL manages its mapping information between the logical addresses and physical addresses of data pages. That is, the FTL algorithms are classified into the block-level FTL, the page-level FTL, and the hybrid FTL.

Among them, the garbage collection algorithms of the hybrid FTL are popularly accepted for its efficiency using a small size of an SDRAM buffer [10, 12]. In those algorithms, a small portion of flash blocks are managed through a page-level mapping scheme and the pages in those blocks are consumed to accommodate out-of-place updates on data pages. We refer to such special-purpose blocks as the *log blocks*. When the log blocks are exhausted because of frequent updates, the garbage collection actions are executed to obtain the clean log blocks again.

During the garbage collection action, the FTL inspects the existence of any log block that can replace other dirty data block. If such a block matching is detected, then the log block and the matched data block are switched to each other. At the same time, the data block is cleaned as a new log block. This type of garbage collection is called the *switch merge*. On the other hand, if a log block to be cleaned contains many pages having their original versions in different data blocks, a *full merge* is executed over multiple flash blocks.

Since a single full merge requires cleaning of more than three blocks and a large number of page copies, its overhead is quite larger than the switch merge. For this reason, the primary goal of garbage collection algorithms is to reduce the amount of full-merges. However, in the presence of heavy workloads of random updates, the previous FTL algorithms cannot guarantee robust I/O performance because of garbage collection overhears. Since the updates in the B-tree are random ones in nature, there is a need to develop an efficient B-tree algorithm suited for flash storage.

2.2 Earlier Flash B+-trees

Since the B+-tree imposes a majority of updates on its leaf level and the leaf nodes are distributed across storage, a flash B+-tree seems to inevitably suffer from frequent full merges because of many random updates. To solve such an inborn problem of the B+-tree stored in flash, some ideas about the flash B+-tree are proposed [3, 7-9, 11-16]. The flash B+-trees in [7, 8, 12] take an approach that updates on leaf nodes are not reflected in real-time. Instead, log data describing an update operation is temporally recorded in other tree levels above the leaf level of the B+-tree. When the amount of such log data for update operations becomes large enough, the data are physically flushed into leaf nodes in storage. To explain in more detail, let us suppose that a key k was deleted from the B+-tree and a certain process P is searching for key k. Also assume that the deletion of key k has been logged on any internal node. In this case, P can find the log data expressing the deletion of k, which is recorded in the middle of its downward search path. The key search of P can be finished without going to the leaf node.

In [12], the update log is incrementally stored in a sub-tree that includes the tree root and upper internal nodes. Since the higher-level sub-tree is emptied at the initial time, it can record many of update operations

until it becomes full. When the nodes of the update-absorbing sub-tree is full because of key inserts and deletes, the update log stored in it are reflected by restructuring the whole leaf nodes in batch mode. During that time, the sub-tree is emptied again. Unlike [12], [7, 11, 13] does not initially allocate any separated sub-tree used for logging updates in the tree. Instead, they can dynamically adjust the sizes of logging areas based on a simple cost model. Those previous schemes using update log residing in internal nodes can replace many of out-of-place updates with writes to clean pages, although they have pay for the management cost of logging areas.

Besides the log management cost, the schemes above have a problem regarding the concurrency control of the B+-tree. Under the traditional B+-tree concurrency control, the locking protocol called *lock-coupling* is applied to guarantee a consistent B+-tree state among concurrent B+-tree processes including searchers and updaters [19, 20]. Since the lock protocol assumes that all the keys are stored at the leaf level, it cannot correctly work for the B+-trees with update logs in internal nodes, without severe degradation of concurrency rate. Additionally, they cannot support fast range searches using the linkage constructed among leaf nodes. This is because the correct set of keys in a B+-tree is not visible without reading update log stored in internal nodes.

To avoid problems above, other algorithms [10, 17] adopt the approach the conventional redo mechanism. In these algorithms, when a page (i.e., a node) is updated, the redo log record for that update operation is saved within a log area, which is prepared in a flash block. The log area for saving redo records uses a portion of pages located at the tail of the block. The pages preceding the log area is used for saving the images of nodes stored in the block. To update a node *N* in a block *X*, an associated redo log record is written to the log area of *X*. Corresponding, to get the latest version of node *N* correctly, [10, 17] reads both a node saving the original image of *N* and the whole redo log records in block *X*. If there exists any redo record(s) of *N*, the latest version of *N* is made through an appropriate redo action(s) on the original image of *N*. Then, the latest version of *N* is cached in the memory buffer.

Although these schemes can completely remove the out-place-updates performed by FTL, they suffer from a waste of storage space for reserving a log area in every block. When a block happens to save internal nodes only, the log area in the block seems to be wasted because the B+-tree has infrequent updates at its internal levels. Moreover, since the memory buffer cannot cache the whole leaf nodes, the response time of key searches can get worse easily because of large I/O times for reading a significant size of redo log areas. To prevent the poor search time and storage utilization, we do not allocate such reserved log areas. We focus on how to update leaf nodes without the support of a FTL, without impairing the B+-tree performance of key searching.

3. Proposed Flash B+-Tree

3.1 Idea Sketch

When a new record having key *k* is inserted into a B+-tree, a downward search is first performed to find a leaf node where the index entry of (*k*, PID) is inserted. Here, the PID is the page ID of the inserted record. If the reached leaf node is a full one, then tree reconstruction is executed for node splitting; otherwise, if the leaf node is a safe node, it is updated to include the new index entry. Because the probability of tree reconstruction is not high, a majority of node updates occur within the leaf level of the B-tree. In particular, the possibility of recursive occurrences of tree reconstruction is very low [20]. This is also true for key deletion operations in the B+-tree.

In this light, our research is focused on how to update leaf nodes at cheap I/O costs. To this end, we

update leaf nodes without the help of FTL so that full merges can be evaded. In the case of internal nodes, meanwhile, they are updated via the FTL to retain the fast search speed. The proposed flash B+-tree achieves fast key search times by using the same structure of internal nodes as in the original B+-tree.

To explain this, we use Figure 1. Figure 1(a) depicts a part of an ordinary B+-tree, where nodes *A*, *B*, and *C* are leaf nodes with rightward linkages for fast range searches. In that figure, the *maximum-key fields* are represented by the underlined symbols. The maximum-key field in leaf node *N* is used to express the maximum value of keys existing in *N*. The maximum-key field in the leaf node is commonly used for range searches [18-20]. Figure 1(b) shows how to store the sub-tree of Figure 1(a) in flash storage according to our proposed scheme. In the proposed B+-tree structure, the sibling leaf nodes, i.e., leaf nodes with the same parent, are stored in the same block as in Figure 1(b). In this paper, we refer to the disk block used for saving sibling leaf nodes as the SLB (Sibling-Leaf Block).

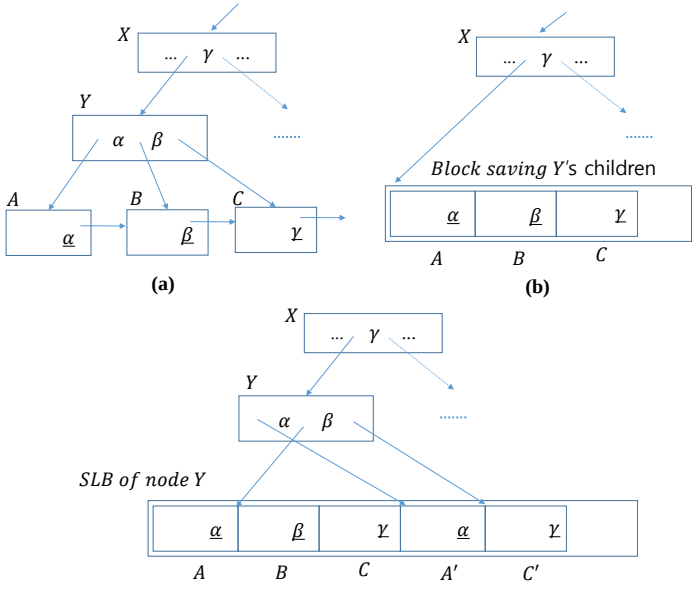

Figure 1. The structure of the SLB for storing node *Y* and its child leaf nodes.

As shown in Figure 1(b), the node *Y* is not saved in flash storage. The proposed B+-tree algorithm generates the node *Y* by referring the maximum-key fields of its child nodes. Note that the maximum-key values of α and β are the same as the key values of *Y*. Since the SLB saves all the leaf nodes with the same parent, we can always generate the virtual parent node by reading its SLB [20].

By using the virtual node concept based on the SLB, we can efficiently perform out-of-place updates on leaf nodes. To see this, we give Figure 1(c). In the figure, we assume that node *A* and *C* were updated because of a key insert and a key delete on each node. To update nodes *A* and *C*, we have written their new version into clean pages located at the tail of the SLB. Correspondingly, the virtual node *Y*, which is existing only in the memory buffer, was modified to have new addresses of *A* and *C* in Figure 1(c). At this point, we can generate the node *Y*, if *Y* is evicted from the memory buffer pool. Since the new versions of *A* and *C* have the same maximum-key values as them of previous versions, respectively, the node *Y* can be generated to correctly point to new versions of *A* and *C*, that is, *A'* and *C'*. The ways to build a B+-tree from leaf nodes in bottom-up fashion have been already addressed in literature [20].

To understand the benefit of the usage of virtual nodes, we suppose that node *Y* is stored in storage and node *A* is updated in out-of-place manner. Then, the node *Y* has to be updated to save the new address of

node *A*. Then, the parent node of *Y* needs to be updated as well, and its update will entail the next update of the parent node of it. Because of this cascading updates of nodes, this naïve update mechanism cannot be used for the B-tree in flash storage. For this reason, the mechanism of the logging based updating was employed in earlier flash B-trees [3, 6, 7, 10-13, 14, 17]. Unlike such schemes, we only have to store the new version of leaf nodes without any update on their parent node. Then, the correct image of the virtual node is obtained from in-SLB data of leaf nodes.

To save the new versions of updated leaf nodes, every SLB keeps a part of pages as clean pages. Then, those new versions are saved in the clean page zone. When the clean page zone in an SLB becomes full because of update operations in it, the SLB is cleaned and all the up-to-date versions of leaf nodes are copied into the SLB. From this reclamation of the SLB, we can make the clean page zone again. This actions for garbage collection is very cheap, compared with that paid for full merges.

3.2 Tree Reconstructing Algorithm

For the purpose of the efficient concurrency control, many B+-tree algorithms do not provide the mechanism for node merging against node underflow. Instead, the underfollowed node is not reconstructed until it becomes empty. When the underfollowed node is emptied, that node is removed from the B+-tree [9]. Of course, in the case of node overflow, all the B+-tree algorithms performs the action of node splitting. We also follows such a general rule for node underflows and overflows in the B+-tree.

Firstly, we explain how to split an overflowed node by writing new nodes in the SLB. In Figure 2(a), the virtual node *Y* currently points to its child nodes *A* and *B*. At this time, nodes *A* and *B* contain two and three index entries, respectively. Here, we assume that the leaf node can contain up to three index entries. Let us suppose that a new index entry is inserted into node *B*. Then, for node splitting we add two nodes of B'_1 and B'_2 into the clean page zone as in Figure 2(b), where $\gamma < \beta$. To point to the new leaf nodes, the virtual node *Y* is also updated in the memory buffer. Since the maximum-key field of B'_2 is the same as that of *B*, the node *B* has been replaced with B'_2.

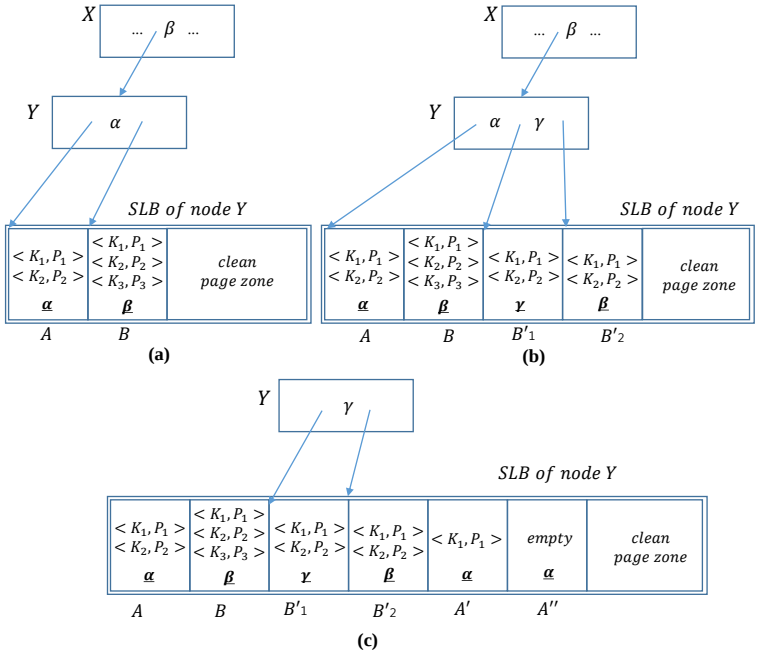

Figure 2. The ways for tree reconstruction using the SLB.

Secondly, we explore the way for removing a node when it is emptied because of key deletes. Let us assume that an index entry is deleted from node *A* of Figure 2(b). Then, the new version of *A* is written in *Y*'s SLB. In Figure 2(c), the node *A'* is used for saving the new version of *A*. At this time, the nodes *A* and *A'* have the same maximum-key value. In the case of node *Y*, *Y* is updated to point to node A' instead of *A*. Also suppose that a key delete arises on *A'* and empties this node. Then, the emptied node is stored as node *A''* of Figure 2(c). At the same time, the pointer to *A'* is deleted from *Y*. Finally, the SLB is made to contain six nodes and two nodes are valid out of them as in Figure 2(c).

3.3 Search Algorithms

As mentioned before, the proposed flash B+-tree is accessed via the memory buffer. For this, the buffer manager caches B+-tree nodes within its memory capacity and manipulates the cached nodes according to its buffer replacement policy. The algorithm for the single-key search is given in Figure 3. The algorithm accepts the key value to be searched for and the root node of the B+-tree. The search algorithm is not different from the original search algorithm that is used for disk-based B+-trees. The only difference is found in lines 3-5, where the virtual node is accessed. If the virtual node has not been cached at the search time, then it is generated from its SLB. The way for the virtual node generation is already addressed in Section 3.1.

Algorithm 1: *SingleKeySearch(Key key, Node Root)*

input : key = key value being requested for searching;
$Root$ = root node of the B+-tree;

```
1  addr_slb ← MoveToVirtual(Root, key); // move to the virtual node
2  if Buffer.IsCached(addr_slb) = false then // check if it is existing in
   buffer memory
3  |  Buffer.ReadSLB(addr_slb, SLB); // load the SLB from storage
4  |  Virtual ← CreateVirtualNode(SLB); // generate the virtual node
5  |  Buffer.CacheNode(Virtual, addr_slb); // cache this virtual node
6  else
7  |  Virtual ← Buffer.Copy(addr_slb); // read from buffer memory
8  addr_l ← MoveToNext(Virtual, key); // move to the child node
9  if Buffer.IsCached(addr_l) = true then
10 |  Leaf ← Buffer.Copy(addr_l); // read from buffer memory
11 else
12 |  Leaf ← Buffer.ReadNode(addr_l); // read the leaf node from storage
13 |  Buffer.CacheNode(Leaf, addr_l); // cache this leaf node
14 return Leaf;
```

Figure 3. The proposed algorithm for single-key searching.

Figure 4 is the algorithm that is used for a range key search, which is requested to search for the keys *k* of $k\_left \le k \le k\_right$. In the figure, the algorithm first moves down to the LSB where the index entry with key *k_left* can be found. Then, other keys greater than *k_left* are retrieved by following the rightward links made on SLBs. This is different from the original range search algorithm that moves rightwards by following links made among leaf nodes. Since the I/O speed for reading a block is 5X-7X faster than that for reading the same size of pages, the proposed range search can provide better response times.

Algorithm 2: *RangeKeySearch(Key k_left, Key k_right, Node Root)*

input : k_left = minimum key value of the range search;
k_right = maximum key value of the range search;
$Root$ = root node of the B+-tree;

```
1 addr_slb ← MoveToVirtual(Root, key); // move to the virtual node
2 Buffer.ReadSLB(addr_slb, SLB); // load the SLB from storage
3 ResultSet.AddRangeResult(SLB, k_left, k_right); // add keys found in SLB
4 while k_right > SLB.maximum-key do // following SLB's links
5 |   addr_slb ← SLB.next-slb; // move to the next right SLB
6 |   Buffer.ReadSLB(addr_slb, SLB); // load the SLB from storage
7 |   ResultSet.AddRangeResult(SLB, k_left, k_right); // add keys
8 return ResultSet;
```

Figure 4. The proposed algorithm for range key searches.

4. Performance Evaluations

In this section, we analyze the performance advantages of the proposed flash B+-tree algorithms, compared with the original B+-tree. For this end, we develop a mathematical performance model for estimating the average time to process search queries and B+-tree update operations.

4.1 Performance Model

To develop a mathematical performance model to be used, we take into account the cost factors of Table 1. In the table, the symbol P_i is used as the buffer hit rate of a node that are accessed at the tree level i during downward searching. Here, the root node has the tree level of 1, and the tree level increases downwards to the leaf level. If a node being accessed is already cached in the memory buffer, the time for reading it is denoted by T_m. Otherwise, if that node has to be read from flash storage for its caching, the time for accessing it is denoted by T_r. The symbol T_w represents the time for writing a node into a clean page, and T_f is the time for updating a node via the FTL. By considering the full-merge overhead, we let the time of T_f be triple the time of T_w. The time for reading a SLB is denoted by T_b.

Table 1. Cost Factors

Symbols	Descriptions
H	Height of the B+-tree to be considered
P_i	Buffer hit rate of a node at tree-level *i*
T_m	Time for reading a node being cached in memory buffer
T_r	Time for reading a node from a flash page
T_w	Time for writing a node into a flash page
T_f	Time for updating a node via the FTL
T_b	Time for reading a flash block
K_m	Maximum number of index entries in the node

Based on the cost factors of Table 1 to be considered, we derive the performance model of the original B+-tree. The average time for searching for a key is given in Figure 5(a), and the average time for inserting or deleting a key is given in Figure 5(b). In Figure 5(a), the value of $(1 - P_i)$ is the probability that a node

located at the tree-level i is not found in the memory buffer at the key search time, That is, that value corresponds to the buffer miss rate of that node. Since the number of index entries in a node ranges from $K_m/2$ to K_m, the probability of the occurrence of tree reconstruction is the same as $2/K_m$. Here, we assume that the actual number of index entries existing in a node is uniformly distributed in the range of $K_m/2$ to K_m.

$$T_{search} = \sum_{i=1}^{H} (P_i \times T_m + T_r \times (1 - P_i))$$

(a) average time for a key search

$$T_{update} = T_{search} + T_f + \sum_{i=1}^{H} ((\frac{2}{K_m})^i \times 3T_f)$$

(b) average time for a key insert or delete

Figure 5. Query processing time of the original disk-based B+-tree.

Figure 6 shows the average search time and the average update time in the case of the proposed flash B+-tree. In Figure 6(a), the value of $(1 - P_{H-1})$ is the buffer miss rate of the virtual node being accessed. If a virtual node is missed in the buffer, then its SLB is read for generating that node. When the proposed B+-tree is updated due to a key insert or delete, the new version of the leaf node is written in T_w. If tree reconstruction is required on that leaf node, it can be completed with three times of node writes. When tree reconstruction is propagated above the virtual node, its ancestor nodes are updated via the FTL. Therefore, that time is computed by $\sum_{i=2}^{H}((\frac{2}{K_m})^i \times 3T_f)$ as in Figure 6(b).

$$T_{update} = T_{search} + \sum_{i=1}^{H-2} \big(P_i \times T_m + T_r \times (1 - P_i)\big) + (1 - P_{H-1}) \times (T_b + 2T_m) + P_{H-1} \times \big(T_m + (P_H \times T_m + T_r \times (1 - P_i)\big)$$

(a) average time for a key search.

$$T_{update} = T_{search} + T_w + \frac{2}{K_m} \times 1.5T_w + \sum_{i=2}^{H} ((\frac{2}{K_m})^i \times 3T_f)$$

(b) average time for a key insert or delete

Figure 6. Query processing time of the proposed flash B+-tree.

4.2 Evaluation Results

To evaluate the performance advantages of the proposed B+-tree, we use the performance model in Figures 5 and 6. The hardware specification of the used flash memory is given in Table 2. The sequential read in flash can provide faster I/O times than the random reads. Its I/O advantage is reported to provide 5X-10X faster read speed, when a whole block is read, compared with the overall time for reading an equal size of scattered pages [12]. Based on the I/O feature reported, we chose a conservative level for the block

read, that is, the level of 5X. From this, we can get two different I/O times of a page read and a block read, respectively, as in Table 2.

Table 2. Flash Memory Specs.

Read Speed		Write Speed	Block Erase
25 μs (2KB of a page)	320 μs (128KB of a block)	200 μs (2KB of a page)	1,500 μs (128KB of a block)

In Figure 7, we compare the processing time of key searches in the proposed flash B+-tree with that in the original B+-tree. Here, we compute the average processing time of key searches with respect to varying buffer miss rates. In the figure, the notation of *Original* (H=*k*) represents the original B+-tree with tree height *k*, while *Proposed* (H=*k*) denotes the proposed B+-tree with tree height *k*. The buffer miss rate is probability that virtual parent nodes are not found in the memory buffer during the search times. This rate is computed as the number of SLB reads over the total number of nodes read for key searches. The higher buffer miss rate entails the longer search times because of many of SLB reads.

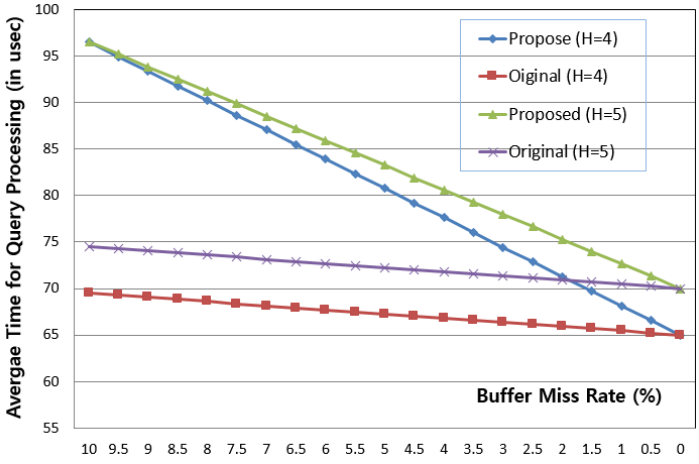

Figure 7. The average processing time for key searches with respect to varing buffer miss rates.

As shown in Figure 7, in the range of high buffer miss rates the proposed B+-tree shows worse performance, compared with original one. However, such performance gaps are fast decreasing with the decreasing values of the buffer misses. With the buffer miss rate of 1%, as a result, the performance gap does not exceed 4%. Since the internal nodes of a B+-tree are usually cached in the buffer during run-time, it is the case that the buffer miss of the virtual nodes is less than 1%. Note that most of buffer misses arise when a request for reading a leaf node is processed. Therefore, we can say that the proposed B+-tree scheme is nearly equivalent with the original B+-tree in terms of key search speeds.

In Figure 8, we show the performance evaluations that have taken into account the factors of update operations. That is, to make performance evaluation more realistic, we consider a situation where update operations are executed together with key searches. In Figure 8, the average processing time on the vertical axis is obtained by averaging the processing times of search queries and update queries. The ratio of update queries is the portion of update queries with respect to the whole queries. The comparisons on the query processing times are done using two different values of buffer miss rates, that is, 20% and 5%.

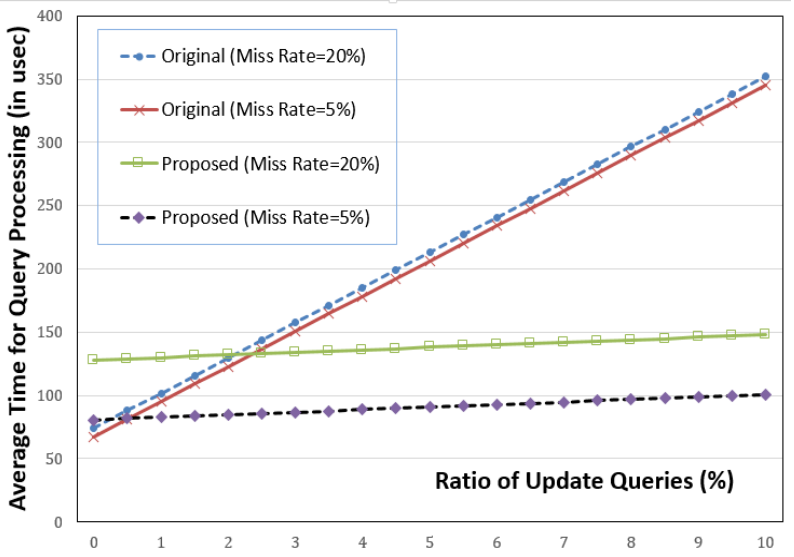

Figure 8. The average processing time with respect to varing portions of update queries.

As shown in Figure 8, the slopes of the increasing times for query processing are very steep in the case of *Original*. As a result, if the buffer miss rate is 5%, then *Proposed* provides a better processing time for the queries containing about 1% of update queries. Recall that such buffer miss rate is not low in a real processing environment. Therefore, we can say that the proposed B+-tree can provide better performance than the original B+-tree in most cases.

To see the performance advantages of *Proposed* more specially, we performed additional evaluations. In Figure 9, we use a B+-tree with the tree height of 4 and the buffer miss rate of 5%. As known from Figure 9, the query processing time of *Original* increases at a faster rate. At the ratio of updaters of 20%, about 300% of the performance improvement is provide by *Proposed*. In the case of the ratio of updaters of 10%, the proposed B+-tree can reduce more than 50% of the query processing time, compared with the original B+-tree.

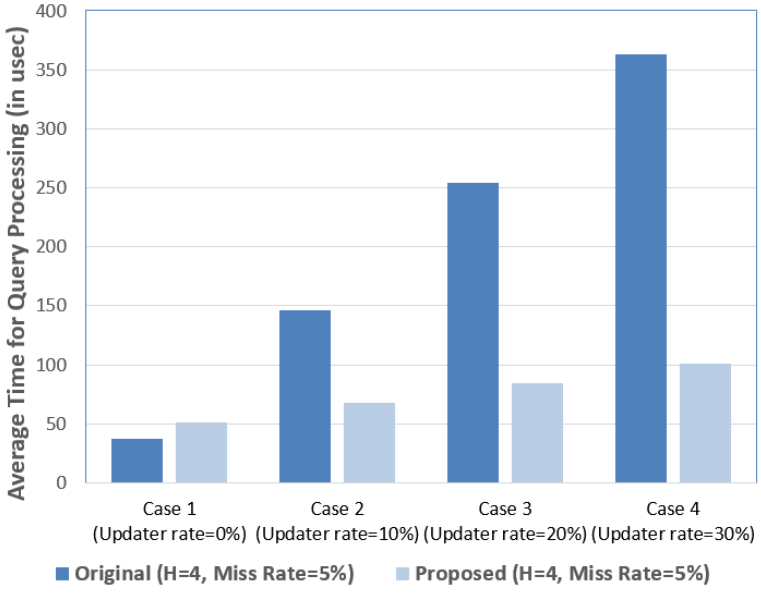

Figure 9. The average processing time for some typical cases with four different ratios of updaters.

5. Conclusions

During the past decades, the B+-tree has been most widely used as an index file structure for disk-resident databases. For the disk based B+-tree, a node update can be cheaply done by modifying its associated disk page in place. However, once the B+-tree is stored on flash memory, the traditional algorithms of the B+-tree seem to be inefficient due to the high cost of in-place updates on flash memory. For this reason, the earlier schemes for flash memory B+-trees usually take an approach that saves data of real-time updates into extra temporary storages. To keep tree consistency, at the same time, search processes are made to refer to that data during their top-down searches. Although that approach can easily evade frequent in-place updates in the B+-tree, it can suffer from a waste of storage space and prolonged search times. Particularly, it is not possible to process range searches by using a link connection chaining leaf nodes. To solve such problems, we devise a new scheme in which the leaf nodes and their parent node are stored together in a single flash block, called the sibling leaf block. Then, we also design some algorithms that are used to perform the node update in the unit of a sibling leaf block. From this, our scheme can improve the storage utilization and the efficiency of key search operations.

Acknowledgement

This work was supported by the Dongduk Women's University grant in 2014.

References

[1] Stepahan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler, "Flashing Databases: Expectations and Limitations", *In Proc. of Data Management on New Hardware*, pp. 9-18, 2010.

[2] Yongkun Wang, Kazuo Goda, and Masaru Kitsuregawa, "Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems", *In Proc. of DEXA*, pp. 777-791, 2009.

[3] Gap-Joo Na, Sang-Won Lee, and Bongki Moon, "Dynamic In-Page Logging for B+-tree Index", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 24(7), pp. 1231-1243, 2012

[4] Sungup Moon, Sang-Phil Lim, Dong-Joo Park, and Sang-Won Lee, "Crash Recovery in FAST FTL", LNCS Vol. 6399, pp. 13-22, 20111.

[5] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang, "An Efficient B-tree Layer Implementation for Flash-memory Storage Systems", *ACM Transactions on Embedded Computing Systems*, Vol. 6(3), 2007 (BFTL)

[6] Sang-Won Lee and Bongki Moon,"Design of Flash-based DBMS: An In-page Logging Approach," *In Proc. of ACM SIGMOD*, pp.55-66, 2007.

[7] Chang Xu, Lidan Show, Gang Chen, Cheng Yan, and Tianlei Hu, "Update Migration: An Efficient B+-tree for Flash Storage", *In Proc. of DASFAA*, pp.276-290, 2010.

[8] Hua-Wei Fang, Mi-Yen Yeh, Pei-Lun Suei, and Tei-Wei Kuo, "An Adaptive Endurance-aware B+-tree for Flash Memory Storage Systems", *IEEE Transactions on Computers*, 2013.

[9] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee, "B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives", *In Proc. of VLDB*, pp.286-297, 2011.

[10] Sang-Won Lee, and Bongki Moon, "Transactional In-Page Logging for Multiversion Read Consistency and Recovery," *In Proc. of ICDE*, pp.876-887, 2011.

[11] Sangwon Park, Ha-Joo Song, and Dong-Ho Lee, "An Efficient Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory", *In Proc. of ICESS*, 2007.

[12] D. Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi, "Tree Indexing on Solid State Drives", *In Proc. of the VLDB*, 2010

[13] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, Shashi Singh, "Lazy-Adaptive Tree: an Optimized Index Structure for Flash Devices", *In Proc. of VLDB*, pp. 361-372, August 2009.
[14] Chang Xu, Lidan Show, Gang Chen, Cheng Yan, and Tianlei Hu, "Update Migration: An Efficient B+ Tree for Flash Storage", *In Proc. of DASFAA*, pp. 276-290, 2010.
[15] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu, "Flash-Optimized B+-Tree", *Journal of Computer Science and Technology*, Vol. 25(3), 2010.
[16] Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, Peng Wei, "A Reliable B-Tree Implementation over Flash Memory", *ACM SAC*, 2008.
[17] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)", *Acta Informatica*, Vol. 33(1), pp. 351-385, 1996.
[18] Marcel Kornacker, C. Mohan, and Joseph Hellerstein, "Concurrency and Recovery in Generalized Search Trees", *In Proc. of SIGMOD,* 1997.
[19] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-ahead Logging", *In Proc. of SIGMOD*, 1992.
[20] Sungchae Lim, Myoung-Ho Kim, "Restructuring the concurrent B+-tree with non-blocked search operations", *Information Sciences*, 147(1-4), 2002.