AS B-tree: A study of an efficient B$^+$-tree for SSDs

SUNGHO KIM[1], HONGCHAN ROH[1], DAEWOOK LEE[2] AND SANGHYUN PARK[1]
[1]*Department of Computer Science*
Yonsei University
Seoul 120-749, Korea
[2]*Department of Computer Science and Engineering*
Sogang University
Seoul 121-742, Korea

Recently, flash memory has been utilized as the primary storage device in mobile devices. SSDs have been gaining popularity as the primary storage device in laptop and desktop computers and even in enterprise-level server machines. SSDs have an array of NAND flash memory packages and are therefore able to achieve concurrent parallel access to one or more flash memory packages. In order to take advantage of the internal parallelism of an SSD, it is beneficial for DBMSs to request input/output (I/O) operations on sequential logical block addresses (LBAs). However, the B$^+$-Tree structure, which is a representative index scheme of current relational DBMSs, produces excessive I/O operations in random order when its node structures are updated. Therefore, the conventional B$^+$-Tree structure is unfavorable for use in SSDs. In this paper, we propose the Always Sequential (AS) B-tree which consists of the Legacy B$^+$-Tree structure, a Sequential Writer, a Write Buffer, an Address Mapping Table, and a Node Validation Manager. All of the modified nodes in the Legacy B$^+$-Tree are stored in the Write Buffer. If the Write Buffer is full, the Sequential Writer contiguously writes each node of the Write Buffer at the end of the file. To support this algorithm, the Address Mapping Table links NodeIDs of the Legacy B$^+$-Tree to the LBA of the corresponding node. Because AS B-tree writes the modified nodes on sequential LBAs in this same manner, it is able to take advantage of the internal parallelism of SSDs. In the experiments presented as part of this paper, AS B-tree enhanced the insertion performance of the conventional B$^+$-Tree by 21%. We also confirmed AS B-tree demonstrates better performance than other flash-aware indexes in search-oriented workloads.

Keywords: Flash memory, B$^+$-Tree, FTL, SSD, parallelism

1. INTRODUCTION

Recently, flash memory has been utilized as the primary storage device in mobile devices. SSDs have been gaining popularity as the primary storage device in laptop and desktop computers and even in enterprise-level server machines. Unlike HDDs in flash memory, an overwrite operation is not allowed in SSDs unless it is preceded by an erase operation on the same block. To address this, a Flash memory Translation Layer (FTL) is employed on the flash memory. Even though the modified data block is overwritten to the same LBA, the FTL writes the updated data block to a different physical address from the previous one, thus mapping the LBA to a new physical address. This enables the flash memory to avoid the high block-erase cost. SSDs have an array of NAND flash memory packages so that they are able to achieve concurrent parallel access to one or more flash memory packages. To take advantage of the internal parallelism of SSDs, it is beneficial for the DBMSs to request I/O operations on sequential LBAs. Since sequential I/Os can be merged and converted into an I/O of a large chunk, the large chunk I/O can span over multiple flash memory packages inside SSDs, thereby enabling parallel data transfers to the multiple flash memory packages at the same time. This is why sequential I/Os shows higher bandwidth than random I/Os on SSDs. Moreover sequential I/Os can enhance the cleaning efficiencyof FTL, thereby improving long-term performance of SSDs [6].

The B^+-Tree structure, which is a representative index scheme of the current relational DBMSs, produces excessive I/O operations in random order when its node structures are updated. Therefore, the conventional B^+-Tree structure is unfavorable to SSDs [13, 14].

In this paper, we propose the Always Sequential (AS) B-tree structure which contiguously writes the modified nodes at the end of its file at every update operation. The LBA of the node used in B^+-tree is replaced with the NodeID of AS B-tree. The updated node is always placed at a newly assigned LBA at the end of the file. Each LBA is linked to the NodeID by the Address Mapping Table stored in the main memory. In this way, AS B-tree writes updated nodes on the sequential LBAs in order to take advantage of the parallelism of the SSDs.

So far, several flash-aware (the indexes having flash-memory oriented designs) indexes have been proposed such as BFTL [7], FD-tree [19], and LA-tree [20]. These indexes share the basic strategy to reduce write operations to SSDs at the expense of degrading search performance compared to the B^+-Tree. On the contrary, AS B-tree focuses on exploiting benefits of sequential I/Os on SSDs by replacing random-writes into sequential-writes, with no degradation in search performance compared to the B^+-Tree.

This paper is organized as follows: Chapter 2 introduces the background and related work of this paper. Chapter 3 explains the architecture and operation of the proposed AS B-tree structure, Chapter 4 presents the experimental results and analysis using the proposed AS B-tree, B^+-tree and other flash-aware indexes, and Chapter 5 presents our research conclusions on the performance of our proposed AS B-tree structure.

2. BACKGROUND AND RELATED WORK

In this section, we first introduce background of this research, features of flash memory and SSDs, and B+-tree which is a basic structure of AS B-tree. Next, we present a brief summary of flash-aware indexes that have been proposed by previous research and compare them with AS B-tree in section 2.3.

2.1 Flash Memory

Flash memory exists as one of two types: NOR or NAND. NOR flash memory supports random bit access and is primarily used for code storage. NAND flash memory is designed for data storage with a denser capacity, only allowing access in a sector unit.

Most currently available SSDs are based on NAND flash memory [2], which is the focus of this section. A NAND flash memory chip consists of blocks which are composed of pages, e.g., 64-128 pages/block. When the data on a page is updated, the new data is typically written to free space in a different memory location, and the old copy of the data is considered invalid. The page is termed a "free page" if it is available to receive data, is considered a "live page" if it contains valid data, and is considered a "dead page" if its data is invalid. The update strategy is referred to as an out-place update. Because it is possible that the data location on a flash memory is changed over time due to out-place updates, LBA space is commonly adopted to efficiently address the updated data [1, 3, 8]. The out-place update is supported by the Flash Translation Layer (FTL), which enables some file systems or DBMSs, which are developed for a hard disk (HDD), to be used without modification or adaptation.

2.2 B^+-Tree

B^+-Tree is a representative index scheme of current relational DBMSs that efficiently manage a large amount of data. A node in B^+-Tree can contain a maximum of *d* key values (K_1, K_2, …, K_d)and d+1 pointers (P_1, P_2, …, P_{d+1}). Figure 1 illustrates an example of a second-order B^+-Tree.

To find an index record with a key value K in B^+-Tree, the root node should be examined. First, a node search for a key value greater than *K* should be conducted. If such a key is found, the corresponding pointer will point to a child node. When the child node is a leaf node, it is determined whether the node contains a key that is equal to *K*. If such a key exists, the B^+-Tree returns the corresponding pointer if not, it returns a failure. To insert an index record with a key value *K*, the proper leaf node should be found, which is then inserted into the node. If the node is already full, a split occurs. To handle $d + 1$ keys in the node, the first $\lceil (d+1)/2 \rceil$ keys are inserted into the existing node, and the remaining keys are inserted into the newly created node. Subsequently, the lowest key of the new node is inserted into its parent node, thereby acting as a separator. If the split is propagated to the root node, a new root node is created, and the height of the B^+-Tree increases. The deletion operation for a key *K* is similar to the insertion operation; after finding the leaf node that contains the key, the key is removed. If B^+-Tree with a fanout of d contains n index records, the worst case cost of the search, insertion and deletion operations is proportional to $\log_{d/2} n$ [4, 5, 9].

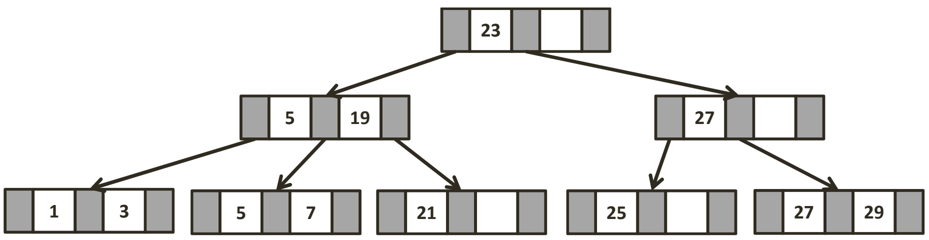


Fig. 1. Example of a second-order B^+-Tree

2.3 SSD

This paper uses the term SSD to indicate flash SSD. Figure 2 depicts a generalized block diagram for an SSD. Each SSD must contain host interface logic in order to enable the support of some form of physical host interface connection (USB, FiberChannel, PCI Express, or SATA) and logical disk emulation, such as a flash translation layer mechanism, which enables the SSDs to mimic a hard disk drive. The performance of an SSD is dependent on the bandwidth of the host interconnection. Therefore, the experiments discussed in this paper utilized an identical SATA interface. A buffer manager contains data which is requested to be transferred along the data path. A multiplexer (Flash Demux/Mux) transmits commands and handles the data transport along the serial connections to the flash packages. The multiplexer can include additional logic, e.g., in order to buffer commands and data. Additionally, a processing engine is required to manage the flow request and mappings from the disk's logical block address to the physical flash location [6, 10].

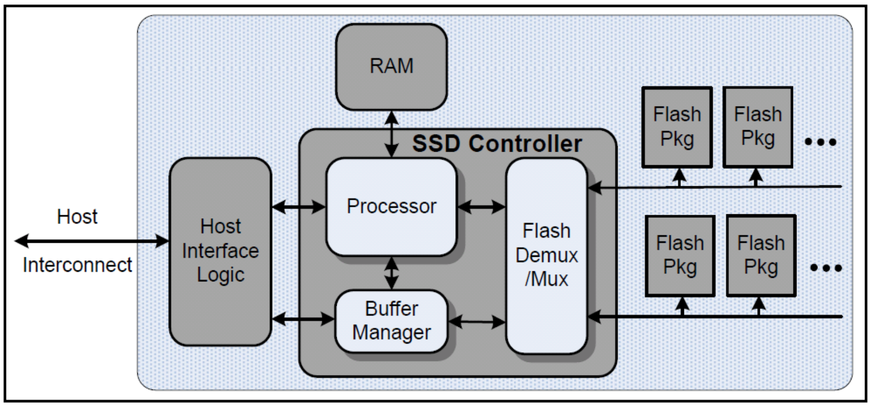


Fig. 2. SSD Logic Components [6]

SSDs can contain several channels, each of which is connected to several flash packages. An SSD controller is able to send request commands to simultaneously read or write data on different channels. In addition, the SSD controller can transfer data from its internal buffer to the flash packages or simultaneously from the flash packages to the buffer on different channels. This type of parallelism is termed Channel-level Parallelism. On the same channel, data writes can be interleaved into several flash packages, thereby allowing simultaneous data access on the same channel. This type of parallelism is termed Package-level Parallelism. By taking full advantage of these two types of parallelism, we were able to obtain the maximum performance from an SSD. However, optimized results were not always able to be obtained in our experiments due to some of the conditions restricting the parallelism of the SSD, including the access pattern for the data, the characteristics of the SSDs, and the unit of the read or write operation [11, 12]. Therefore, it is imperative that we understand the characteristics of SSDs and the conditions that make it possible to utilize the parallelism. Table 1 shows the experimental results which help in understanding the SSD characteristics.

The experiments were conducted using the IOMeter benchmark tool on a Linux machine with an 8 Core CPU and 16 GB of RAM. The parameter value for the outstanding I/O was fixed at 32 in order to examine the maximum performance of the SSD with heavy workloads. The Ext2 file system was configured. In Table 1, SR, SW, RR and RW indicate sequential read, sequential write, random read, and random write, respectively. Table 1 shows that the throughputs for SW and RW were 188 and 22 MB/s, respectively, with the I/O size of 8 KB. The throughput of SW is nine times faster than that of RW, thus indicating the focus for obtaining the maximum performance from an SSD. To efficiently utilize SSDs, data access patterns should be SW as opposed to RW.

I/O	SR	SW	RR	RW
4KB	193 MB/s	158 MB/s	149 MB/s	21 MB/s
8KB	206 MB/s	188 MB/s	192 MB/s	22 MB/s
16KB	206 MB/s	186 MB/s	227 MB/s	32 MB/s
32KB	206 MB/s	175 MB/s	256 MB/s	40 MB/s

Table 1. SSD performance (Intel x25-e SSD, IOMeter workload)

2.4 Related Work

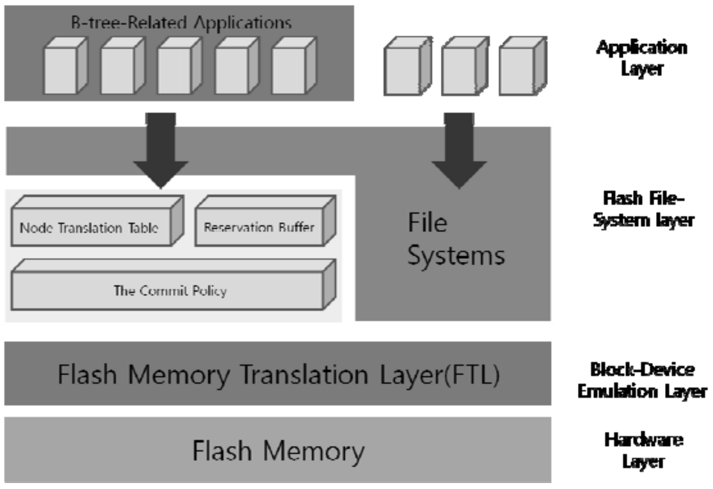


Fig. 3 (a). System architecture of BFTL [7]

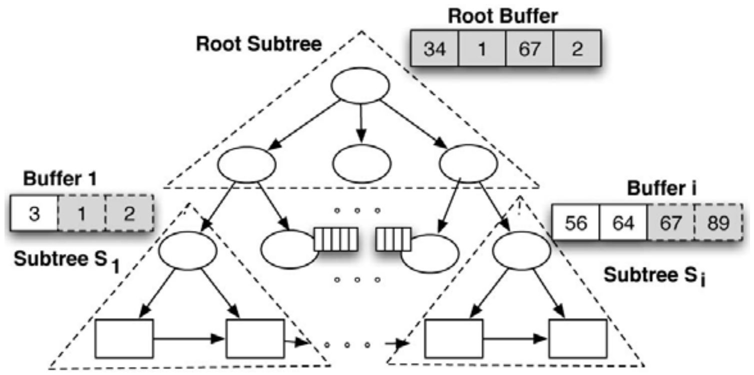


Fig. 3 (b). System architecture of LA-tree [20]

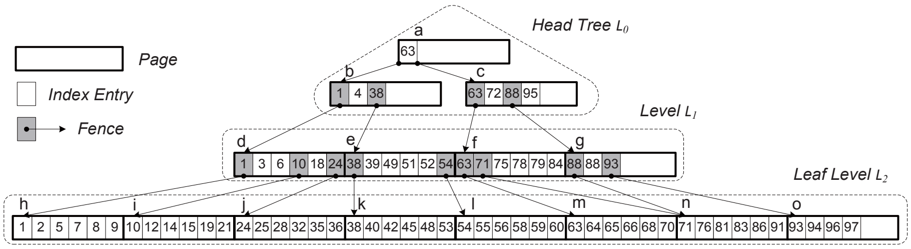


Fig. 3 (c). System architecture of FD-tree [19]

In this section, we introduce other flash-aware indexes such as BFTL [7], FD-tree [19], LA-tree [20].

Figure 3 (a) shows the architecture of BFTL, which consists of Index Units, the Commit Policy and a Node-Translation Table. When a B$^+$-Tree node is inserted, deleted or modified, a new index record is generated and temporarily retained in the Reservation Buffer. If the reservation buffer becomes full, all dirty index records are removed. When the index records in the Reservation Buffer are removed, the Index units are appended right after the sector where the index units were most recently appended. To construct the logical view of a B^+-Tree node, the relevant index units need to be integrated [7]. Therefore the Node-Translation Table maintains the information where the index units of a logical B^+-Tree node are scattered. Since the index units of a logical node can be scattered over multiple sectors, the search cost of the node can be increased by the factor of the number of sectors. The number of sectors in each logical node is limited by the compaction parameter *C*. If the number of sectors for each node becomes greater than *C*, then the BFTL performs the compaction operation that gathers and rewrites the index units together after reading all the sectors where the index units are scattered. Since multiple update operations can be buffered in the Reservation Buffer and can be later flushed into a sector at once, the update operations of the BFTL causes considerably less write operations on SSDs than B^+-Tree. For the search operations, however, the BFTL can read *C* sectors for a node, thus increasing the search cost by a factor of *C* in the worst case.

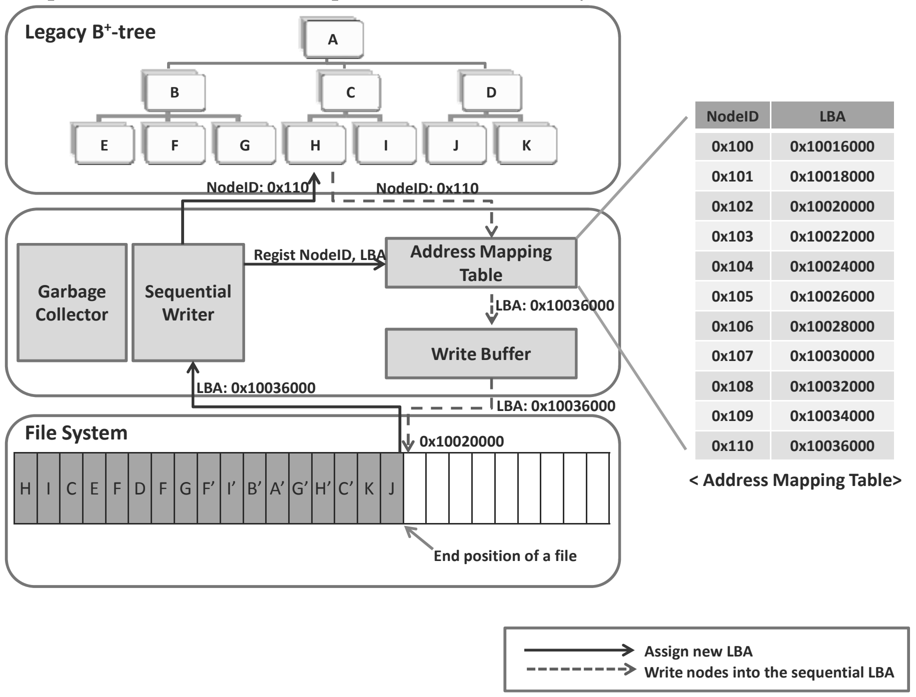
Figure 3 (b) presents an example of LA-tree, which has nearly the same structure as the B+-tree except the appended write buffer for each subtree. The basic strategy of deferring node-write operation is similar to that of the BFTL. However, the write buffer is not an in-memory structure like the Reservation Buffer of BFTL but it is stored on SSDs. Whereas the Reservation Buffer globally buffers all the update operations, LA-tree has a write buffer for each subtree that absorbs the update operations coming from the parent node of the subtree. The deferred update operations in the write buffer are dealt in the corresponding subtree, delivering them to the write buffers of the child subtrees, when the write buffer becomes full. Since the write buffers of LA-tree creates an additional overhead for search operations but the additional cost is less than the BFTL's since not every node of LA-tree has the write buffer.

FD-tree is illustrated in Figure 3 (c). An FD-tree consists of the head tree and several levels of sorted runs. On the top levels, the head tree resides. The head tree is a B+-tree that contains the index records most recently updated. If the limited size of the head tree is exceeded by newly updated index records, then all the index records in the head-tree is merged to the next level sorted run. If the sorted run becomes full because of the migrated index records from the head tree, then all the index records in the sorted run are merged into the next level sorted run. FD-tree is similar to B^+-Tree in the perspective that both of them are logarithmic methods. However, the inserted index records build up the index from the bottom level to the top level in B^+-Tree but from the top level to the bottom level in FD-tree. Since the merge operations are performed with sequential I/Os, FD-tree demonstrates outstanding performance for update operations. On the contrary, each node of FD-tree has less fanout than a B+-tree node since FD-tree node maintains not only index records but also the additional data structures, called internal fence and external fence. Less fanout indicates greater height of the index tree. Therefore, the search performance of FD-tree is worse than that of B+-tree.

BFTL, LA-tree, and FD-tree indexes have a common strategy to enhance the update performance on SSDs. They enhance the update performance by reducing the number of write operations to flashSSDs, at the expense of degrading search performance. As will be explained in the following section, AS B-tree does not compromise search performance at all. The search operation of AS B-tree operates nearly the same as that of B^+-Tree. Moreover, AS B-tree enhances the update operation of B^+-Tree by gathering updated nodes in the write buffer of AS B-tree and writing them together sequentially on flashSSDs. AS B-tree and FD-tree has a common point that they both exploit the benefit of sequential I/Os in order to enhance the update performance, but FD-tree has the worse search performance than AS B-tree as explained above.

3. AS B-TREE

Figure 4 depicts the architecture of an AS B-tree that consists of a Legacy B^+-Tree, a Sequential Writer, a Write Buffer, an Address Mapping Table, and a Node Validation Manager. Because the Legacy B^+-Tree is identical to a B^+-Tree except for the write and read operations, no additional explanations are necessary.



NodeID	LBA
0x100	0x10016000
0x101	0x10018000
0x102	0x10020000
0x103	0x10022000
0x104	0x10024000
0x105	0x10026000
0x106	0x10028000
0x107	0x10030000
0x108	0x10032000
0x109	0x10034000
0x110	0x10036000

Fig. 4. The AS B-tree architecture

3.1 Sequential Writer

To effectively utilize the internal parallelism of SSDs, all of the modified nodes will be written on consecutive LBAs. The node modified by an insert/delete/update operation will be written at the end of the file instead of being overwritten. This storage management forces all of the modified nodes to be written sequentially. Additionally, organizing sequential writes still improves performance by reducing the fragmentation and wear-leveling overhead, especially when the update size is smaller than the flash page size. Algorithm 1 presents a detailed explanation of this process. When a node is modified, it is written in a new LBA (lines 4-6) rather than being written in the LBA in which it was previously placed. The new address is located at the end of the file. AS B-tree accesses each node according to its unique node ID number (e.g., 0x00, 0x01, 0x02, …) instead of its logical block address (e.g., 0x10000000, 0x10016000, 0x10036000, …). In Algorithm 1, NodeID and LBA denote the node ID number and logical block address, respectively. Even though the modified node is newly written at the end of the file, the original NodeID of the node is not changed. To access a node in Legacy B^+-Tree using its NodeID, the Address Mapping Table is utilized, which converts a NodeID to the corresponding LBA (lines 1, 2).

```
Procedure: Insert(K, NodeID)
Input: K(inserted key), NodeID(node ID number)
begin
1:    LBA ← MapTbl[NodeID]; // MapTbl[]:   table that maps NodeID to LBA;
2:    Node ← read from LBA; // Node: data of a node.
3:    InsertKey(K, Node);
4:    LBA ← get the end position of the file;
5:    MapTbl[NodeID] ← LBA;
6:    write Node to LBA;
7:    Function InsertKey(K, Node)
8:        //Node.KEYS: the array of the keys in the Node
9:        I ← search the location that should be inserted in the Node.KEYS;
10:       Node.KEYS[i] ← K;
end
```

Algorithm 1. Assignment of the sequential LBA

3.2 Write Buffer

When an index record is inserted into a B^+-Tree, node reads as many as the height of the B^+-Tree have to be performed in order to locate the leaf node that contains the index record. In the B^+-Tree whose height is three, the following steps are processed in order to insert an index record. First, a root node is read; second, an internal node is read; third, a leaf node is read; finally, the modified leaf node is written. This indicates that each insert operation produces a write operation for every three read operations (read-before-write operation) [15, 16]. If the database makes use of a buffer cache, some write and read operations may not occur. If the requested node is in the buffer cache, either a writing or reading step will be omitted. This read before write operation occurs, even when the buffer cache is used.

If the node is not in the buffer cache, these operations will occur, and the written or read nodes will then be stored into the buffer cache. Before the node is stored into the buffer cache, a node that is selected as a victim by the replacement policy of the buffer cache must be flushed to the disk.

According to the experimental results for SSD testing, a read-before-write operation degrades the performance of an SSD [5]. Therefore, even if the modified nodes are written on sequential LBAs, we are unable to benefit from the internal parallelism of SSDs because of the read-before-write operation. The Write Buffer is used to resolve this problem, as shown in Figure 5. First, the modified node is temporarily placed in the Write Buffer contained in the main memory, not directly in the SSDs. Second, if the Write Buffer is full, the nodes are moved to the respective LBA. When the Write Buffer is not utilized, the read operations between the write operations interrupt the sequential write operations, thus inhibiting performance improvements to the insert operation. When the Write Buffer is utilized, only the read operations are processed; the write operations are deferred by placing the modified nodes into the Write Buffer of the main memory. In this way, consecutive write operations can be processed after consecutive read operations; thus avoiding performance degradation of the SSDs.

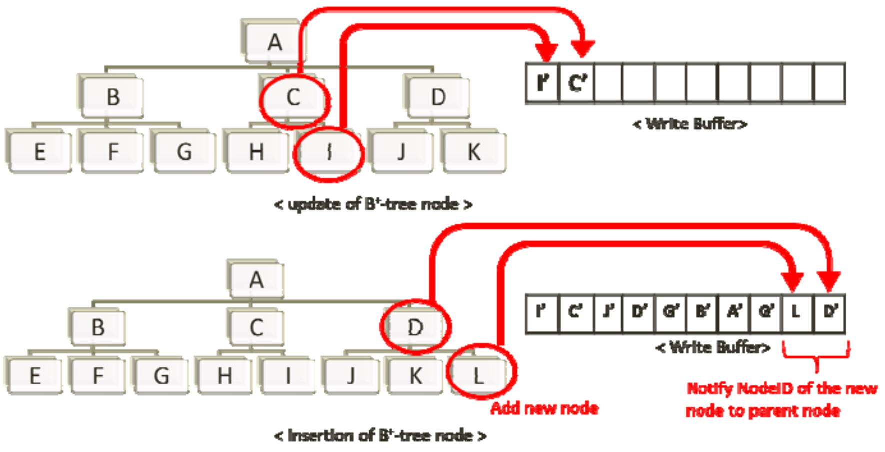

A
B
C
D
E
F
G
H
I
J
K
I'
C'
< Write Buffer>
< update of B+-tree node >
A
B
C
D
E
F
G
H
I
J
K
L
I'
C'
J'
D'
G'
B'
A'
G'
L
D'
< Write Buffer>
Add new node
Notify NodeID of the new node to parent node
< Insertion of B+-tree node >

Fig. 5. Write Buffer

Algorithm 2 describes the manner in which the modified node and a newly assigned node LBA are placed in the Write Buffer. The structure of the node stored in the Write Buffer consists of the Write Buffer Address (WBA) and the Write Buffer Data (WBD),where WBA is the NodeID of the node to be stored, and WBD is the data of the node to be stored. If the Write Buffer has the WBA of the node that is requested to be modified, the modified node needs to be replaced with the corresponding WBD (lines 3-9). In this case, the Write Buffer operates analogous to the buffer cache. If there is no other node with the same WBA, the Write Buffer assigns a buffer block for the node and places the WBA and WBD of the node into the buffer block (lines 11-14). If the Write Buffer is full, the WBD of each node in the Write Buffer is written to its associated WBA (lines 15-22).

```
Procedure: WriteNode(NodeID, Node)
Input: NodeID(node ID number), Node(data of the modified node)
begin
1:  bFound = FALSE;
2:  i=0;
3:  for i < WBN   do      // WBN: the number of nodes in Write Buffer
4:      if(WBA[i] = NodeID)   then      // WBA[]: NodeIDs of nodes in Write Buffer
5:        copy Node to WBD[i];      // WBD[]:data of nodes in Write Buffer
6:              bFound = TRUE;
7:      endif
8:      i++;
9:  endfor
10: if(bFound = FALSE)   then
11:     i ← get an index of the empty buffer in Write Buffer;
12:     WBA[i] ← NodeID;
13:     copy Node to WBD[i];
14:     WBN++;
15:     if (WBN = MS) then                  // MS: the maximum size of Write Buffer
16:         i=0, WBN=0;
17:         for i < MS   do
18:             LBA ← MapTbl[WBA[i]];
19:             write WBD[i] to LBA;
20:             i++;
21:         endfor
22:     endif
23: endif
end
```

Algorithm 2. Writing to a node with Write Buffer

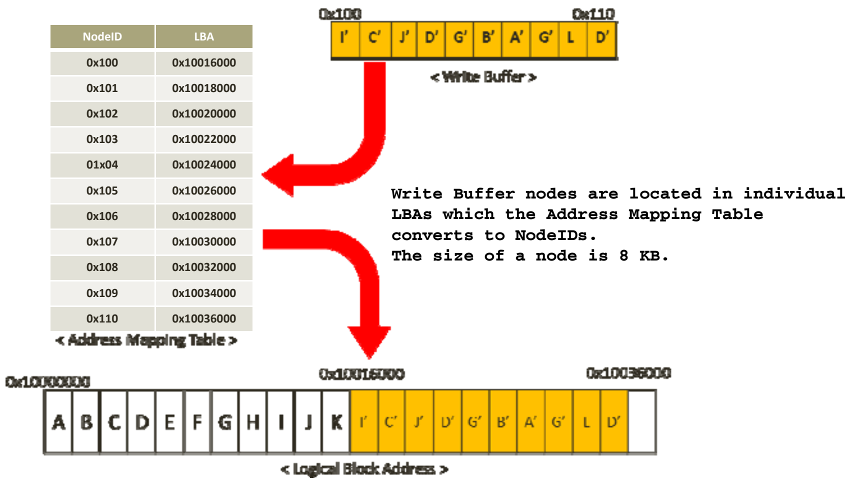
Algorithm 3 describes the manner in which read operations are processed with the Write Buffer. First, the Write Buffer is analyzed to determine if it contains the node requested to be read. Second, if the node exists, the node can be read directly from the Write Buffer in the main memory (lines 3-9). If the node does not exist in the write buffer, the corresponding LBA of the node is obtained from the Mappiing Table by using the given NodeID. With the obtained LBA, AS B-tree then reads the node from the flashSSD..

```
Procedure: ReadNode(NodeID)
Input: NodeID(node ID number)
Output: Node(data of the read node)
begin
1:  bFound = FALSE;
2:  i=0;
3:  for   i < WBN   do    // WBN: the number of nodes in Write Buffer
4:      if(WBA[i] = NodeID)   then    // WBA[]: NodeIDs of nodes in Write Buffer
5:          copy WBD[i] to Node;    // WBD[]:data of nodes in Write Buffer
6:          bFound = TRUE;
7:      endif
8:      i++;
9:  endfor
10: if(bFound = FALSE)   then
11:     LBA ← MapTbl[NodeID];
12:     Node ← read from LBA;
13: endif
14: return (Node);
End
```

Algorithm 3. Reading the node with Write Buffer

3.3 Address Mapping Table

To place the modified nodes on the sequential LBAs, we make use of not only the LBAs, but also the NodeIDs. Figure 6 shows the process used by the Address Mapping Table to convert a NodeID to the corresponding LBA. If the Write Buffer has no more available space for storing modified nodes, then each node in the Write Buffer is placed into its own LBA according to the Address Mapping Table. Because each node in the Write Buffer has a sequential LBA, the nodes as many as the size of the Write Buffer are written to sequential LBAs of a flashSSD. When B^+-Tree is read from SSDs to the main memory, an Address Mapping Table, with NodeIDs and LBAs of all the nodes, is created by reading all of the nodes from the SSDs.



NodeID	LBA
0x100	0x10016000
0x101	0x10018000
0x102	0x10020000
0x103	0x10022000
01x04	0x10024000
0x105	0x10026000
0x106	0x10028000
0x107	0x10030000
0x108	0x10032000
0x109	0x10034000
0x110	0x10036000

Write Buffer nodes are located in individual
LBAs which the Address Mapping Table
converts to NodeIDs.
The size of a node is 8 KB.

< Address Mapping Table >
0x100
0x110
I' C' J' D' G' B' A' G' L D'
< Write Buffer >
0x10000000
0x10016000
0x10036000
A B C D E F G H I J K I' C' J' D' G' B' A' G' L D'
< Logical Block Address >

Fig. 6. Address Mapping Table converts the NodeID to the LBA

3.4 Node Validation Manager

Since a modified node is not placed into its previous LBA but into the end of the file, every modification increases the file size. The distribution of the valid nodes was examined by performing insert operations with keys in a uniform distribution. As shown in Figure 7, most valid nodes are placed at the end of the file. The number of valid nodes in old files, which have low file descriptor number(25~29), is low and that of new files, which have high file descriptor number(30~34), is high. This means that valid nodes rarely exist in the front or middle of the file. If the distribution of the values of the inserted index records is not uniform, then the valid nodes are scattered in a larger area of the SSDs. For space efficiency, the area where invalid nodes exist needs to be reorganized. An AS B-tree is stored in several files, each having 64 MB of storage space. The Address Mapping Table contains the LBAs of the valid nodes; therefore, we were able to determine if each LBA points to a valid node. By maintaining a count of the number of valid nodes in each file, it is possible to locate the file that contains a number of invalid nodes and thus needs to be recycled. If the number of valid nodes in a file is less than T percent of all of the nodes in the file, the recycling process is performed. First, in order to transmit the valid nodes to the new locations, the valid nodes in the file are read and re-written to the end of the newest file. In our experiments, T was set to 5%. After rewriting, the file is deleted, and the Address Mapping Table is updated with the new LBA of the transmitted nodes. By periodically performing this process, we were able to reduce the total file size of AS B-tree.

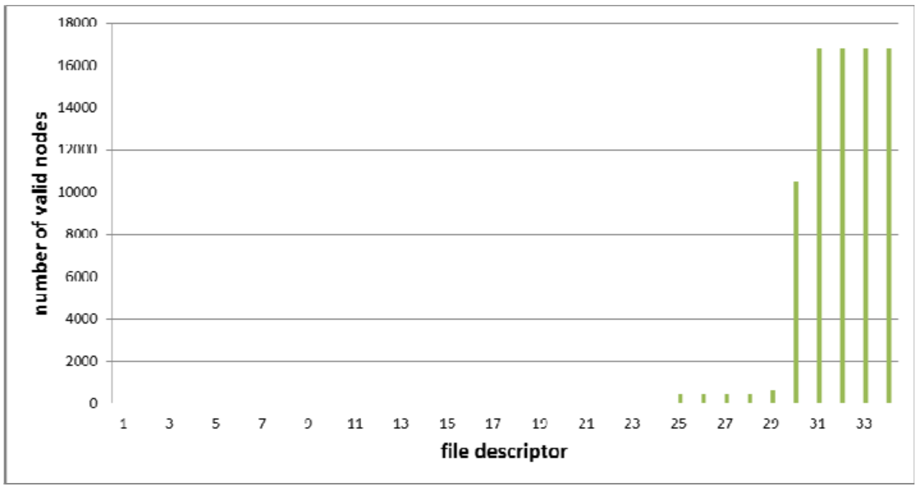
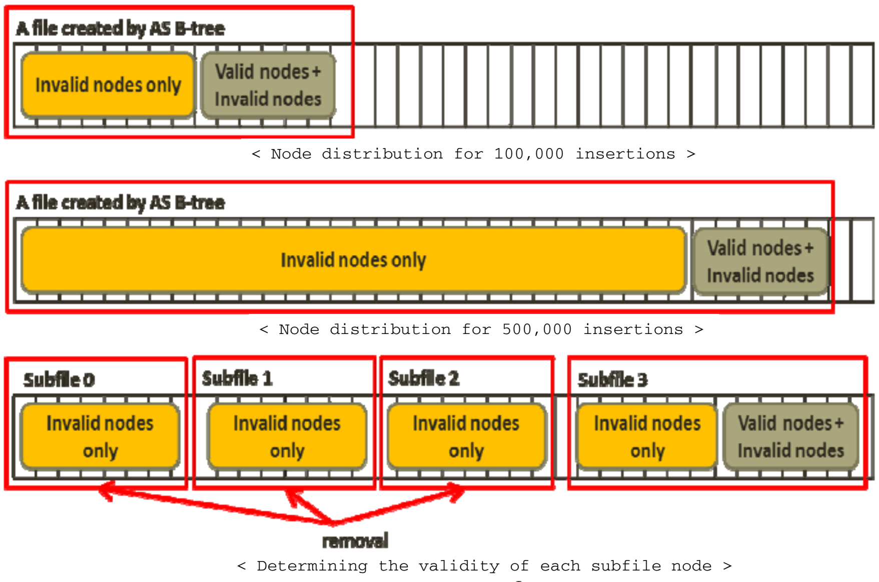


Fig. 7. Number of valid nodes in subfiles created by AS B-tree



A file created by AS B-tree
Invalid nodes only
Valid nodes + Invalid nodes
< Node distribution for 100,000 insertions >
A file created by AS B-tree
Invalid nodes only
Valid nodes + Invalid nodes
< Node distribution for 500,000 insertions >
Subfile 0
Invalid nodes only
Subfile 1
Invalid nodes only
Subfile 2
Invalid nodes only
Subfile 3
Invalid nodes only
Valid nodes + Invalid nodes
removal
< Determining the validity of each subfile node >

Fig. 8. Determining the validity of each subfile node

4. EXPERIMENTAL RESULTS

All experiments were conducted on an Intel 8 Core CPU machine with 16GB memory running the Linux OS. For the experiments, we used intel x25-e SSD [21] which is known to be popularly used as an element of a SSD array (RAID configurations) in Enterprise servers. A file system using Buffered I/O mode allows the written or read data blocks to be stored in the write/read caches in order to minimize the number of read and write operations. Therefore, we are unable to measure the exact improvements caused by the Write Buffer in the file system using the Buffered I/O mode. The Direct I/O mode, which is the feature of a file system that allows write or read operations to bypass the write/read caches of the operating system. This mode was used in all of our experiments.

We first evaluated each index operation (insert, search, delete) of AS B-tree by comparing its performance with that of the B+-tree and BFTL. The total performance of AS B-tree was also evaluated by using synthetic workloads having mixed operations. In the workloads, we compared AS B-tree performance with other flash-aware indexes such as BFTL [16], FD-tree [19], and LA-tree [20], varying we the insert/search operation ratio. Each operation in the workload was randomly chosen among the insert and search operation. For all the workloads, random integer values were used for the index keys with uniform distribution.

For implementation, we utilized the source codes [22, 23] of FD-tree and LA-tree published by the authors of the papers. Since BFTL has no published source codes, we implemented a BFTL index following the description in the paper [16]. The FD-tree source codes were minimally modified for the conversion from its original Window-style codes to Linux style codes in order to conduct the experiments in the same environment as other indexes. The published source codes of LA-tree does nothing but creates an output file containing I/O traces of the I/O operations which the LA-tree is supposed to produce in response to the given workloads (In order to obtain the real response time, one more step is needed, executing the corresponding I/O operations written in the output file to SSDs).

4.1 Individual Index Operation Performance

We compared the performance of AS B-tree with those of the B^+-Tree and BFTL. After initializing the indices with 400,000 index records, we measured the elapsed time of 100,000 insert operations. Likewise, an equal number of search, update, and deletion operations were conducted. For the experimental performance evaluation of AS B-tree, we used two parameters: the maximum number of keys in a node, F, and the maximum number of nodes stored in the Write Buffer, i.e., the Write Buffer size. F was varied from 64 to 256 in increments of two, and the size of the Write Buffer was varied from 1 to 32. The configuration of BFTL used in this experiment is described as follows. The Reservation Buffer size was set at F, indicating that the maximum number of Index Units that could be contained in the Reservation Buffer was identical to F. Each node in the Node-Translation Table (NTT) could have a maximum of C sectors. In our experiment, C was set at 3, implying that the compaction process was performed when the length of the LBAs was greater than 3 [6].

Figure 9 shows the experimental results for insert operations. The performance of AS B-tree was better than those of the other structures, as shown in Figure 9. AS B-tree performance was enhanced as F and the Write Buffer sizes increased; the read cost decreased for an insert operation as a result of the reduced height in Legacy B^+-Tree due to an increase in F. As the size of the Write Buffer increased, more data can be simultaneously written, which enhances the performance of the insert operation by utilizing the internal parallelism of SSDs. SSDs have better performance with write-only operations compared to that of a system with a mingled workload of both read and write operations [1]. An AS B-tree with an F of 256 and a Write Buffer size of 32 demonstrated a 21% performance improvement compared to that of the B^+-Tree, and it was also 57% faster than BFTL.

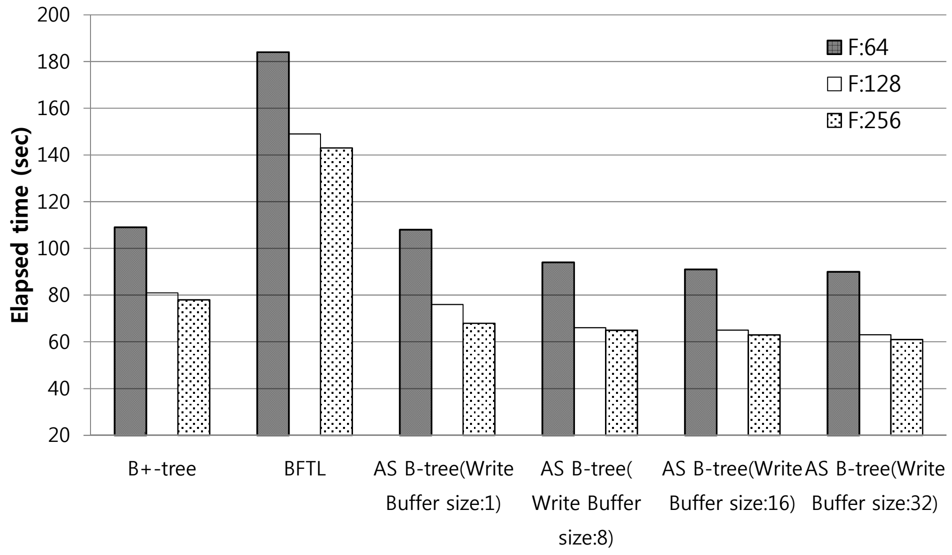


Fig. 9. Elapsed time of insert operations

Table 2 shows the elapsed time of the insert operations of AS B-tree with an F of 256 and the Write Buffer size of 32 compared to that of a B^+-Tree with a Write Buffer size of 32. The elapsed times of the read and write operations for an insert operation were measured separately. The write operation performance of AS B-tree was improved by 50% compared to that of B^+-Tree because the nodes that were stored in the Write Buffer were written to the sequential LBAs of SSDs. The Write Buffer in the main memory was able to operate analogous to a buffer cache. If a node that had already been cached in the Write Buffer was updated, it could be counted as a buffer hit. In this way, we measured the hit ratio of the updated nodes inside the Write Buffer: 1.1% for the read operations and 0.6% for the write operations. These small hit ratios suggests that the performance enhancement of AS B-tree did not originate from cache hits. The reason why the read operation time of AS B-tree was reduced by 17% compared to that of B^+-Tree was that the algorithm and implementation for determining where the modified node should be placed were simplified by placing the modified node at the end of the file.

	Read operation	Write operation	Total
AS B-tree	56 sec	5 sec	61 sec
B^+-tree	68 sec	10 sec	78 sec
Improvement of AS B-tree	17%	50%	21%

Table 2. Elapsed time of the insert operations of AS B-tree and B^+-Tree

Applying the same conditions as those for the insert operations, we measured the elapsed time of 100,000 search operations in AS B-tree, B+-Tree, and BFTL, as shown in Figure 10.

AS B-tree had nearly the same elapsed time as did the B+-Tree and a shorter time than BFTL. BFTL produced poor results due to its 37% more read operations compared to those of both AS B-tree and B+-Tree. BFTL tried to read an average of 1.49 sectors to produce a node in our experiments.

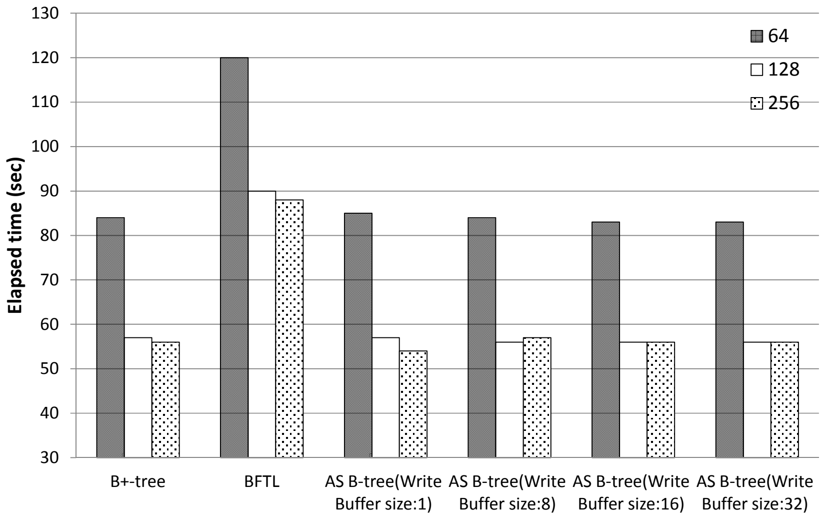


Fig. 10. Elapsed time of search

Applying the same conditions as those for the insert operations, we measured the elapsed time of 100,000 update operations in AS B-tree, B^+-Tree, and BFTL, as shown in Figure 11.

AS B-tree had the shortest elapsed time, and BFTL had the longest elapsed time. AS B-tree with an F of 256 and a Write Buffer size of 32 demonstrated a 21% better performance compared with that of the B^+-Tree with a Write Buffer size of 32, and it also demonstrated a 32% faster performance compared to that of BFTL. AS B-tree performance was enhanced as F and Write Buffer size increased. Although the number of inserted index records into AS B-tree did not increase as it did for an insert operation, the algorithm of the update operation is similar to that of an insert operation. Therefore, the insert operations graph in Figure 11 has a similar pattern to that of the update operations in Figure 9.

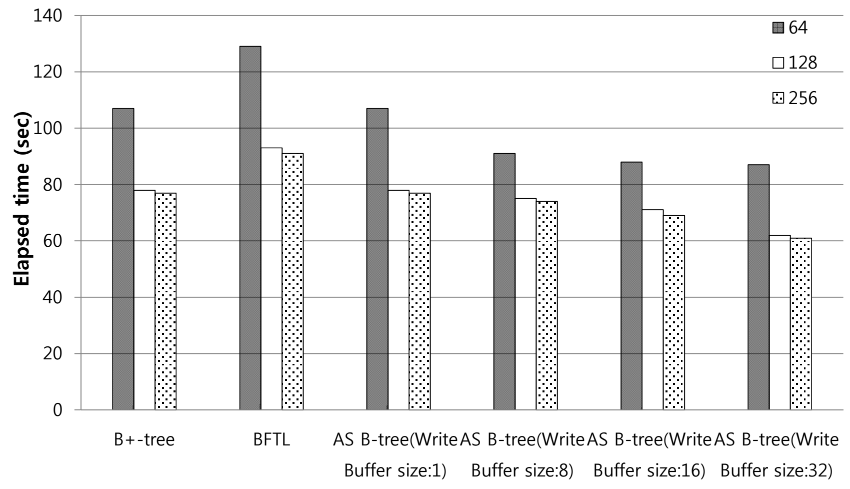


Fig. 11. Elapsed time of update operations

Applying the same conditions as those for the insert operations, we measured the elapsed time of 100,000 delete operations in AS B-tree, B^+-Tree and BFTL, as shown in Figure 12.

The delete operations had similar results to those for the insert operations. AS B-tree had the shortest elapsed time, and BFTL had the longest elapsed time. AS B-tree with an F of 256 and a Write Buffer size of 32 demonstrated a 21% better performance than did the B^+-Tree with a Write Buffer size of 32, and it demonstrated a 57% faster performance than that of BFTL. The experimental results for the delete operations are nearly identical to those for the insert operations.

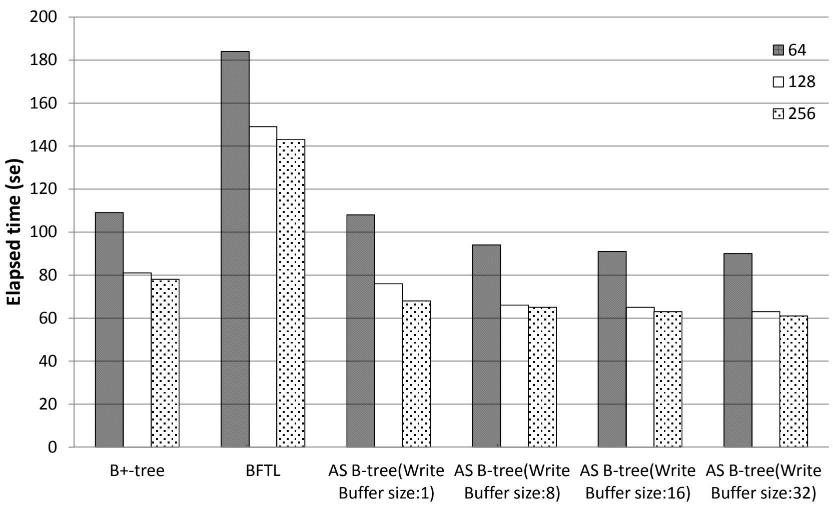


Fig. 12. Elapsed time of delete operations

We attempted an experimental comparison of the insert and search operations with a buffer cache that is similar with that in the database. We used Least Recently Used (LRU) as the replacement policy for the buffer cache, and the cache size was expressed as the number of obtainable nodes in the buffer cache. Figure 13 shows that AS B-tree, B$^+$-Tree and BFTL were improved at a similar rate relative to the hit ratio of the buffer cache. As the size of the cache increased, each index structure achieved improved performance. In order to maintain a similar performance with the same hit ratio, the cache size needed to increase when the number of inserted index records increased. The Write Buffer of AS B-tree guarantees a similar performance to that of a fixed Write Buffer size in the main memory, even if the number of inserted index records increases. Figure 14 shows that the search operations of each indexing structure were also improved at a rate similar to that for insert operations.

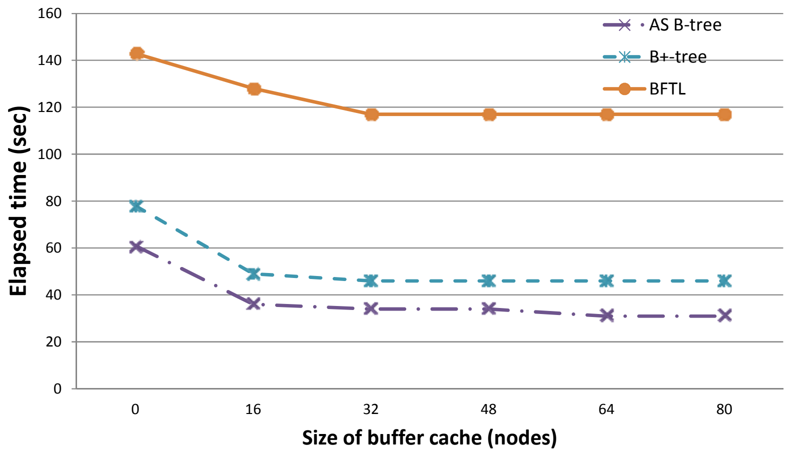


Fig. 13. Elapsed time of insert operations with a buffer cache (cache replacement policy: LRU, F: 256, size of node: 8KB, the number of insert operations: 100,000)

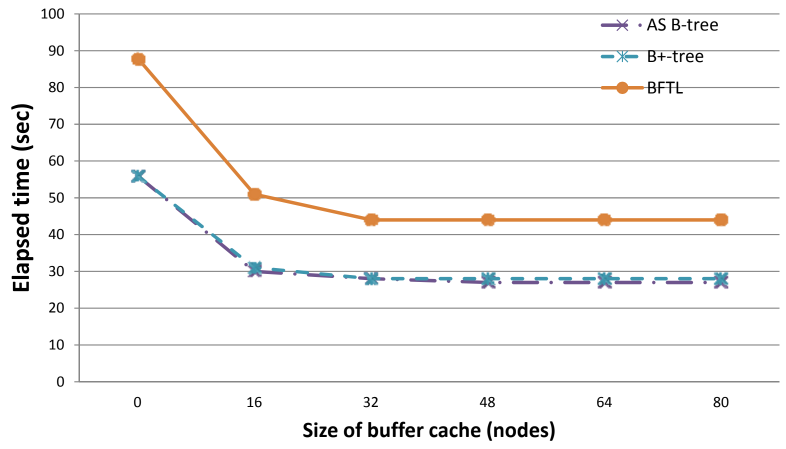


Fig. 14. Elapsed time of search operations with a buffer cache (cache replacement policy: LRU, F: 256, size of node: 8KB, the number of search operations: 100,000)

Table 3 shows the file sizes used in B^+-Tree, BFTL and AS B-tree. The size of the file used by BFTL was 1,615 MB, larger than those of the other methods, where B^+-Tree and AS B-tree used 22.4M B and 70 MB files on SSDs, respectively. When the Node Validation Manager was not used in AS B-tree, the file size was 642 MB. In flash memory, a node that is overwritten on the same LBA is placed at a different physical address by FTL. Therefore, if the same workload was used for insertions, the file size difference on the physical address layer between AS B-tree and B^+-Tree is trivial.

	B^+-tree	BFTL	AS B-tree
Size of file	22.4 MB	1,615 MB	70 MB

Table 3. File sizes in B+-tree, BFTL and AS B-tree

4.2 AS B-tree Performance in Mixed Workloads

In this section, we compare AS B-tree performance with other flash-aware indexes such as BFTL [16], FD-tree [19], and LA-tree [20] by using various synthetic workloads. We created synthetic workloads that consist of insert and search operations in total of 1 million operations. We varied the insert/search ratios from 10% insert and 90% search to 90% insert and 10% search, increasing insert ratio by 20% at a time. To simplify the workloads, search and insert operations are only included (the cost of update or delete operation is similar to the cost of insert operation). The indexes were created with initial 500,000 records. Each index was configured with the 8KB node size and fanout of 256, and each index utilized 896KB main memory space, which is nearly 4% of the index file sizes. AS B-tree was configured with the 640KB buffer cache (80 nodes) and 256KB write buffer (30 nodes). The BFTL parameter configuration followed its settings in Section 4.1. FD-tree and LA-tree's parameters were configured, following the description in their papers (e.g. FD-tree's parameter *k* was set at 128, which is a half of fanout (256) as was used in the paper [19]).

Figure 15 and Table 4 present the response times of the indexes when one million index operations were performed to each index. The elapsed time of AS B-tree was gradually reduced whereas that of the other flash-aware indexes was increased with increasing search-ratio. AS B-tree excelled in the search-oriented workloads (10/90, 30/70), where it was 3.55 to 4.56, 1.57 to 1.81, 0.99 to 1.05 times faster than BFTL, FD-tree, and LA-tree, respectively. Since the basic strategy of the other flash-aware indexes is to replace relatively expensive random-writes with random-reads, the other flash-aware indexes showed lower search performance than AS B-tree. Therefore, AS B-tree, which enhances update performance without sacrificing search performance, demonstrated nearly the best in the search-oriented workloads (10/90, 30/70). We found out that the insert/search ratio of the index operations created by OLTP applications are close to the 30/70 workload. We tried to reveal the insert/search ratio of OLTP workloads. We obtained index operation traces from the inside of a PostgreSQL DBMS by executing TPC-C benchmark [24], which is a representative OLTP benchmark, upon the DBMS. To do so, we modified the relevant source codes of the PostgreSQL DBMS. We obtained index operation traces that consist of 72 % search, 24 % insert, 3% range search, and 1% delete index operations. This implies that AS B-tree can perform nearly the best among the flash-aware indexes in OLTP workloads since the performance of AS B-tree was almost the best among the flash-aware indexes with the 30/70 workload used in this experiment (3.55, 1.57, and 0.99 times faster than BFTL, FD-tree, and LA-tree, respectively).

As mentioned earlier in Section 2.4, the other flash-aware indexes compromise search performance to enhance update performance. This is the key reason why AS B-tree showed good performance in search-oriented workloads (10/90, 30/70). With consideration of the fact that AS B-tree, LA-tree, and BFTL are B+-tree variants, the performance of each index can be compared with that of B+-tree. BFTL enhances update performance, at the expense of degrading the search performance by a factor of *C* (BFTL's compaction parameter), compared with B+-tree. LA-tree adopts write buffers to defer node updates, so search performance is worse than that of B+-tree due to the additional cost of reading the write buffers. On the contrary, AS B-tree has almost the same search performance as B+-tree (refer to the search operation experiments in Section 4.1). FD-tree has a smaller fanout of nodes than the original B+-tree has, whereas AS B-tree has the same fanout of nodes as the original B+-tree's. FD-tree, hence, has the greater tree-height, thus demonstrating lower search performance than B+-tree and AS B-tree. In addition, FD-tree includes B+-tree as its head tree. In order for the FD-tree to efficiently operate, the nodes of the head tree should be loaded onto main memory, so it requires a certain amount of main memory space to hold the head tree nodes. Therefore, the performance of the FD-tree degrades considerably with the limited amount of main memory.

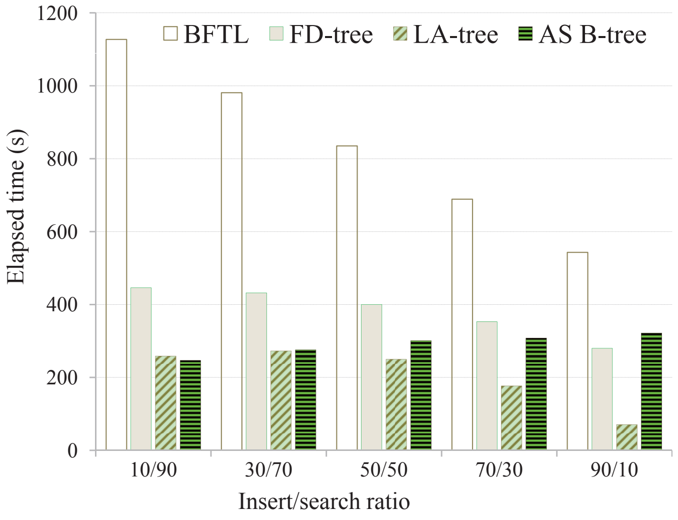


Fig. 15. Elapsed time of 1 million operations

	BFTL	FD-tree	LA-tree	AS B-tree
10/90	1127 s	446 s	259 s	247 s
30/70	981 s	432 s	273 s	276 s
50/50	835 s	400 s	250 s	301 s
70/30	689 s	353 s	177 s	308 s
90/10	543 s	280 s	71 s	322 s

Table 4. Elapsed time of 1 million operations

5. CONCLUSION

SSDs, which have demonstrated large growth in the memory storage market, have an array of NAND flash memory packages; an SSD is able to achieve concurrent parallel access to one or more flash memory packages. In order to effectively take advantage of the internal parallelism of SSDs, it is helpful for the DBMSs to request I/O operations on sequential LBAs. The B^+-Tree structure, which is a representative index scheme of current relational DBMSs, produces excessive I/O operations in random order when its node structures are updated. Therefore, the original B^+-Tree is not favourable for SSDs. In this paper, we proposed AS B-tree which consists of Legacy B^+-Tree, a Sequential Writer, a Write Buffer, an Address Mapping Table, and a Node Validation Manager. All of the modified nodes in Legacy B^+-Tree are stored in the Write Buffer. If the Write Buffer becomes full, the Sequential Writer contiguously writes each node of the Write Buffer at the end of the file. To facilitate this algorithm, the Address Mapping Table links the NodeIDs of Legacy B^+-Tree to the LBA of the corresponding node. Using this method, AS B-tree is able to take advantage of the internal parallelism of SSDs.

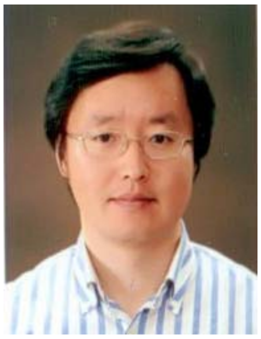
REFERENCES

1. Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber, "A Design for High-Performance Flash Disks," ACM SIGOPS Operating Systems Review, Vol. 41, 2007, pp. 88–93.
2. Feng Chen, David A. Koufaty, and Xiaodong Zhang, "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives," SIGMETRICS '09 Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, 2009, pp. 181–192.
3. Li-Pin Chang,Tei-Wei Kuo, "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation," ACM Transactions on Storage (TOS), Vol. 1, 2005, pp. 381–418.
4. Bayer, R. and McCreight, "Organization and Maintenance of Large Ordered Indices," ACTA Informatica, Vol. 1, 1972, pp. 173–189.
5. Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, "μ-tree: an ordered index structure for NAND flash memory," EMSOFT '07 Proceedings of the 7th ACM & IEEE international conference on Embedded software, 2007, pp. 144–153.
6. Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D, "Davis, Mark Manasse and Rzna. Design tradeoffs for SSD performance," USENIX 2008 Annual Technical Conference on Annual Technical Conference, 2008, pp. 57–70.
7. 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 (TECS), Vol. 6, 2007, pp. 1–23.
8. Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh and Wonhee cho, "A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications," ACM Transactions on Embedded Computing Systems (TECS), Vol. 7,

2008, pp. 1–23.
9. Douglas Comer, "Ubiquitous B-Tree," ACM Computing Surveys (CSUR), Vol. 11, 1979, pp. 121–137.
10. Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, Sang-Woo Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," SIGMOD '08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008, pp. 1075–1086.
11. Chen, Feng, "On Performance Optimization and System Design of Flash Memory based Solid State Drives in the Storage Hierarchy," 2010.
12. Sang-Won Lee, Bongki Moon, Chanik Park, "Advances in flash memory SSD technology for enterprise database applications," SIGMOD '09 Proceedings of the 35th SIGMOD international conference on Management of data, 2009, pp. 863–870.
13. Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Sing, "Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices," Proceedings of the VLDB Endowment, Vol. 2, 2009, pp. 361–372.
14. Gap-Joo Na, Bongki Moon and Sang-Won Lee, "In-Page Logging B-Tree for Flash Memory," Database Systems for Advanced Applications, Lecture Notes in Computer Science, Vol. 5463, 2009, pp. 755–758.
15. Hyun-Seob Lee, Sangwon Park, Ha-Joo Song and Dong-Ho Lee, "An Efficient Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory," Embedded Software and Systems, Lecture Notes in Computer Science, Vol. 4523, 2007, pp. 181–192.
16. Chin-Hsien Wu, Li-Pin Chang and Tei-Wei Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems," Real-Time and Embedded Computing Systems and Applications, Lecture Notes in Computer Science, Vol. 2968, 2004, pp. 409–430.
17. D. Kang, D. Jung, J. U. Kang, and J. S. Kim, "μ-Tree : An Ordered Index Structure for NAND Flash Memory," Proceedings of the 7th ACM & IEEE international conference on Embedded software, 2007.
18. D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and Walid A. Najjar, "Microhash: An Efficient Index Structure for Flash-based Sensor Devices," In Proc. USENIX Conference on File and Storage Technologies (FAST), 2005.
19. Y. Li, B. He, Q, Luo, and K. Yi, "Tree indexing on flash disks," in Proceedings of International Conference on Data Engineering (ICDE), IEEE, 2009.
20. D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, "Lazy-adaptive tree: An optimized index structure for flash devices," in Proceedings of International Conference on Very Large Data Bases (VLDB), 2009.
21. Intel. intel x25-e.
http://download.intel.com/design/ash/nand/extreme/319984.pdf.
22. The FD-tree source code.
http://pages.cs.wisc.edu/~yinan/fdtree.html.
23. The LA-tree source code
http://www.cs.umass.edu/~dagrawal/code/laTreeCodeRelease.tar.gz.
24. Transaction Processing Performance Council. TPC benchmark C, standard specification version 5.

Sungho Kim is working for a master's degree in Department of Computer Science, Yonsei University in 2011. He is working in Department of Computer Science, Yonsei University, Seoul, Korea. His current research interests include flash memory, SSD and database.

Hongchan Rho is working for a Ph.D. degree in Department of Computer Science, Yonsei University in 2011. He is working in Department of Computer Science, Yonsei University, Seoul, Korea. His current research interests include flash memory, SSD and database.



Daewook Lee received the Ph.D. degree in Department of Computer Science, Seoul National University in 2009. He is working in Department of Computer Science and Engineering, Sogang University, Seoul, Korea. His current research interests include DBMS, XML and SSD.

Sanghyun Park received the Ph.D. degree (computer science) from University of California at Los Angeles (UCLA) in 2001. He is now a Professor in Department of Computer Science, Yonsei University, Seoul, Korea. His current research interests include database, data mining, bioinformatics, flash memory and SSD.