Dynamic In-Page Logging for B^+-tree Index

Gap-Joo Na, Sang-Won Lee, and Bongki Moon

Abstract—Unlike database tables, B^+-tree indexes are hierarchical and their structures change over time by node splitting operations, which may propagate changes from one node to another. The node splitting operation is difficult for the basic In-Page Logging (IPL) scheme to deal with, because it involves more than one node that may be stored separately in different flash blocks. In this paper, we propose *Dynamic IPL B^+-tree* (*d-IPL B^+-tree* in short) as a variant of the IPL scheme tailored for flash-based B^+-tree indexes. The *d-IPL B^+-tree* addresses the problem of frequent log overflow by allocating a log area in a flash block dynamically. It also avoids a page evaporation problem, imposed by the contemporary NAND flash chips, by introducing ghost nodes to *d-IPL B^+-tree*. This simple but elegant design of the *d-IPL B^+-tree* provides significant performance improvement over existing approaches. For a random insertion workload, the *d-IPL B^+-tree* outperformed a B^+-tree with the plain IPL scheme by more than a factor of two in terms of page write and block erase operations.

Index Terms—Dynamic in-page logging, flash memory indexing, B^+-tree.

1 Introduction

The recent advances in the flash memory technology have garnered much attention from various sectors of industry from mobile devices to home entertainment and appliances to business enterprises. Due to its superiority in access latency, energy consumption and the two-fold annual increase in its density, flash memory storage devices, mostly in the form of solid state drives (SSDs), are being adopted rapidly by storage and database vendors for large-scale enterprise servers. However, the erase-before-update property of flash memory is still considered the most serious limitation that would result in tardy small random writes, which are fairly a dominant access pattern in database tables and indexes.

Recently, a new buffer and storage model called *In-Page Logging (IPL)* has been proposed to optimize the write performance of flash-based database systems [1]. The key idea of the IPL scheme is to colocate data pages and their associated log records in the same flash block such that the amount of physical writes is minimized at the nominal overhead of read operations. By writing physiological log records into the same block containing the corresponding data pages without updating the data pages themselves in place, the IPL scheme can effectively overcome the erase-before-update limitation of flash memory. It has been shown that the IPL scheme can improve substantially the I/O performance of a flash-based database system for database tables [1] and B^+-tree indexes [2].

Unlike database tables, B^+-tree indexes are hierarchical and their structures change over time by node splitting operations, which may propagate changes from one node to another. The node splitting operation is difficult for the IPL scheme to deal with, because it involves more than one index node that may be stored separately in different flash blocks. This will lead to serious concerns we call *frequent log overflow* and *page evaporation* problems. As we discuss in more detail in Section 2.3, the performance of the IPL scheme might deteriorate without addressing these concerns adequately.

In this paper, we present *Dynamic IPL B^+-tree* (*d-IPL B^+-tree* in short) as a variant of the IPL scheme tailored for flash-based B^+-tree indexes so that the frequent log overflow and page evaporation problems can be addressed. The *d-IPL B^+-tree* improves the utilization of flash blocks by allocating a log area within a flash block dynamically, and avoids frequent log overflow by reducing the number of log records required by a node splitting operation. Specifically, the *d-IPL B^+-tree* stores a new index node and an old index node from which the new one is split in the same flash block, so that the structural change caused by a node splitting operation can be represented by a few physiological log records. This simple but elegant design of the *d-IPL B^+-tree* improves the performance significantly. For a random insertion workload, the *d-IPL B^+-tree* outperformed a B^+-tree with the plain IPL scheme by more than a factor of two in terms of page write and block erase operations.

There have been several studies for flash-based B^+-tree indexes [3], [4], [5]. They assume a software layer called a Flash Translation Layer (FTL) as a substratum for the B^+-trees, which provides a block device interface for upper layers by emulating operations of a disk drive using flash memory. Due to the FTL dependency, however, the FTL-based approaches cannot utilize the characteristics of flash memory fully, and their performance may not be scalable or predictable.

- *G.-J. Na is with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, 440-746, Korea, and the Electronics and Telecommunications Research Institute (ETRI), Daejon, 305-700, Korea. E-mail: funkygap@etri.re.kr.*
- *S.-W. Lee is with the School of Information and Communication Engineering, Sungkyunkwan University, 300 Cheoncheon-Dong, Jangan-Gu, Suwon, Gyeonggi-Do, Korea. E-mail: wonlee@ece.skku.ac.kr.*
- *B. Moon is with the Department of Computer Science, University of Arizona, Gould-Simpson Bldg., Room 746 PO Box 210077, Tucson, AZ 85721-0077. E-mail: bkmoon@cs.arizona.edu.*

Manuscript received 24 Feb. 2010; revised 24 Aug. 2010; accepted 30 Oct. 2010; published online 1 Feb. 2011.
Recommended for acceptance by B.C. Ooi.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-02-0108.
Digital Object Identifier no. 10.1109/TKDE.2011.32.

The key contributions of this work are summarized as follows: first, we discover the problem of frequent log overflow caused by a node splitting operation in a B^+-tree under the plain IPL scheme. Second, in order to address the problem, we extend the IPL scheme such that log areas are dynamically allocated, two splitting nodes are colocated in the same flash block, and a node split event can be represented succinctly in a few physiological log records. Third, to address the page evaporation problem imposed by the contemporary NAND flash chips, we introduce the notion of a ghost node to the dynamic IPL scheme. Fourth, we evaluate the performance of *d-IPL B^+-tree* in comparison with existing FTL-based B^+-tree approaches.

The rest of this paper is organized as follows: Section 2 reviews related work such as the IPL scheme and existing FTL-based B^+-tree approaches, and discusses the motivations of the *d-IPL B^+-tree* we propose in this paper. Section 3 describes the structure of the *d-IPL B^+-tree* index and presents its key ideas such as dynamic log areas, ghost nodes, and log write policies. Sections 4 and 5 present the procedures for insertion, deletion, and search operations. Section 6 analyzes the performance of the *d-IPL B^+-tree* index. Finally, Section 7 summarizes the contribution of this paper.

2 Related Work and Motivation

In this section, we review existing FTL-based B^+-tree approaches as well as the IPL scheme proposed for flash memory-based database systems. We also provide the motivations of this work by defining the problems that would occur when the IPL scheme was applied to B^+-tree indexes without taking the characteristics of hierarchical B^+-tree indexes into consideration.

2.1 B^+-tree Indexes with Underlying FTL

Flash memory does not allow any data item or a page containing the data item to be updated in place just by overwriting it [6]. In order to update an existing data item stored in flash memory, a time-consuming erase operation must be performed in advance. Besides, the erase operation cannot be performed selectively on a particular page, and can only be done for an entire block (or an erase unit) containing the page to be updated. A block is much larger (typically 64 or 128 times) than a page, and a block erase operation is much costlier than a page read or page write operation.

In order to hide its unique characteristics different from existing block devices such as disk drives, most flash memory storage devices are equipped with a firmware or software layer called a FTL [6]. The FTL provides a block device interface for upper layers by emulating operations of a disk drive, and is responsible for several essential functions of flash memory storage devices such as address mapping and wear leveling. Therefore, it will be convenient to build a conventional disk-based B^+-tree index on top of FTL. However, the frequent random writes of B^+-tree indexes make the conventional approach vulnerable to performance degradation due to the erase-before-update limitation of flash memory. Most existing flash-aware approaches still utilize FTL as an underlying layer to build B^+-tree indexes but they adopt additional strategies to avoid random write operations as much as possible. This section introduces such FTL-based flash-aware B^+-tree indexes as BFTL [3] and FlashDB [4], and discusses their limitations.

2.1.1 BFTL

To the best of our knowledge, BFTL [3] is the first flash-aware B^+-tree designed on top of FTL. Its objective is to minimize the amount of redundant writes that may be required by flash memory limitations. BFTL represents an insert/update/delete operation applied to a B^+-tree node as a log record, and stores the log data sequentially in a RAM area called a reservation buffer. The log data are flushed to flash memory when a node-sized buffer slot becomes full.

Since BFTL does not dedicate a physical storage unit to a set of log records belonging to the same tree node, the log records of a tree node can be scattered in many different physical pages in flash memory. For this reason, BFTL needs to maintain a complex node translation table (NTT) that maps each logical node in a B^+-tree index to the pages containing any log record of the node. Consequently, when a node is accessed, the node should be constructed on the fly by reading all the pages containing its log records from flash memory.

In order to reduce the read overhead in BFTL, a compaction operation is invoked for a node, when the number of pages containing any log data of the node exceeds a predefined threshold value. For the compaction operation, BFTL reads the pages containing the log data of the node, builds the node in RAM, and then write the node back into a series of pages. The more compaction operations are invoked by BFTL, the less page read operations may be needed but the more write and erase operations will be required. In addition, BFTL does not suggest any kind of garbage collection for the invalid pages at the logical file system level, after the compaction operation.

2.1.2 FlashDB

Nath et al. have proposed another approach called FlashDB with self-tuning features [4]. FlashDB tunes the performance of a B^+-tree index by separating tree nodes into two groups by their access patterns. If a tree node is more read than written, the node is referred to as disk type and stored just like a tree node of a conventional disk-based B^+-tree index. If a tree node is more written than read, the node is referred to as log type and managed by a strategy similar to BFTL. In order to deal with changing access patterns, a cost function is used to determine the type of each tree node such that the overall IO cost of a B^+-tree index is minimized. However, this will also increase the memory usage to keep track of read and write operations and incur additional overhead to change the type of a tree node back and forth.

Like BFTL, FlashDB relies on the heavy use of data structures stored in RAM, which makes it vulnerable to sudden power failures. In addition, the performance of BFTL and FlashDB is not predictable because they use FTL as an underlying layer, and it can vary considerably depending on the characteristics of a chosen FTL implementation.

2.2 In-Page Logging B^+-tree Index

The IPL scheme takes advantage of logging for write reduction so that the overall I/O performance of a flash memory-based database system is improved [1]. In contrast to conventional logging where log data are appended sequentially (e.g., log-structured file system [7]), IPL attempts to address the erase-before-update limitation of flash memory by colocating log records with their corresponding data pages in the same flash blocks. This can reduce the absolute volume of writes significantly at the nominally increased cost of read operations. As a pure electronic device without any moving part, the write speed of flash memory is uniform regardless of the physical location where a write operation needs to be performed. Therefore, with flash memory, log records can be stored in the same flash block with the corresponding data pages regardless of their physical location without incurring excessive latency for random writes.

One obvious benefit of colocating data pages and their log records is that the cost of a page read (including the cost of accessing its log records) cannot be more than accessing a flash block containing the page and the log records, because they can always be found in the same block. Besides, the IPL scheme attaches an in-memory log sector to a buffer frame, when the buffer frame becomes dirty. An in-memory log sector (512 bytes) can absorb several update operations before it becomes full or its associated page frame is evicted by the buffer manager. When a dirty page frame is evicted, its in-memory log sector is written to flash memory but the page itself is not. Note that such a sector write, smaller than a page write, is feasible, because most contemporary (SLC-type) flash memory chips support *partial programming*.[1] We assume that sector writes are allowed by the partial programming of flash memory chips.

IPL B^+-tree [2] was the first attempt to apply the IPL scheme to B^+-tree indexes without having to rely on a particular FTL implementation. It adopts the in-page logging strategy to deal with frequent updates required for a flash-based B^+-tree index. As will be described in the next section, however, the plain IPL scheme is not capable of dealing with frequent updates adequately, because a node splitting operation of B^+-tree involves more than one index node that may be stored separately in different flash blocks.

In this paper, we propose a new flash-aware and FTL-independent index structure called *d-IPL B^+-tree* to minimize the number of required block level operations. The *d-IPL B^+-tree* index uses log areas *dynamically* to improve the utilization of flash memory and stores newly split tree nodes temporarily as *ghost nodes* to deal with node splitting operations efficiently.

2.3 Problem Definition

Insertion and deletion operations of a B^+-tree index do not consume log areas rapidly unless they cause structural changes in the B^+-tree. When a tree node needs to be split, however, if the old and new nodes have to be stored separately in different flash blocks, log areas will be consumed rapidly because the node split operation cannot be represented by a few physiological log records. Furthermore, this problem is often exacerbated due to the sequential page write requirement of most contemporary NAND flash memory chips.

1. Although a page (typically 2 KB) is the basic unit of read and write operations, SLC-type flash memory chips allow a limited number of partial writes to be performed on a flash memory page [8], [9]. Therefore, it is possible to write a page with a single page write operation or with four separate (512 byte) sector write operations.

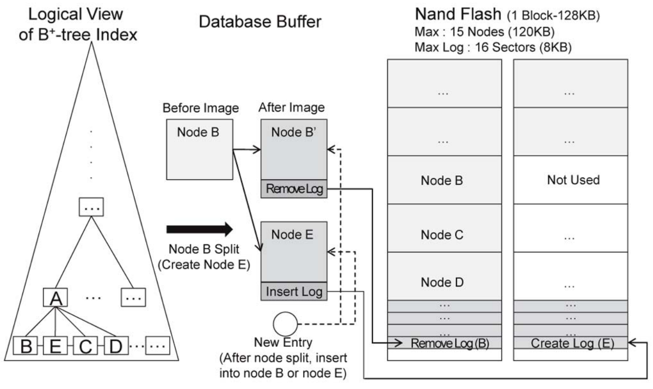
Logical View of B+-tree Index
Database Buffer
Nand Flash (1 Block-128KB)
Max : 15 Nodes (120KB)
Max Log : 16 Sectors (8KB)
Before Image
After Image
Node B
Node B'
Remove Log
Node B Split (Create Node E)
Node E
Insert Log
New Entry (After node split, insert into node B or node E)
A
B
E
C
D
Node B
Node C
Node D
Not Used
Remove Log (B)
Create Log (E)

Fig. 1. Node split of IPL B^+-tree.

2.3.1 Frequent Log Overflow

Fig. 1 illustrates a node split operation of an IPL B^+-tree index. When a tree node E is split from an existing node B, it may suffice to produce a few log records to describe this operation physiologically. Since the IPL scheme requires that data pages are associated with their own log sectors independently from each other, the node B needs a log record that denotes the removal of half of its entries and the node E needs a log record that denotes the insertion of the other half of B's entries. Although the IPL scheme requires that data pages and their log sectors are colocated in the same flash block, it is not guaranteed that the nodes B and E will be stored in the same flash block. Therefore, if the nodes B and E are written to two different blocks, then these blocks become subject to subsequent *block cleansing* operations independently from each other. If the block containing the node B is cleansed, then the log records of B will not be available to the node E any longer. This makes it impossible to compute the new version of node E. One way of avoiding this problem is to store "physical" log records—one for each entry in a tree node—instead of physiological log records, when a tree node is split. This approach, however, would require each node split operation produces as many log records as the entries stored in a node, which would in turn end up consuming log sectors in the block very quickly. We call this problem a *frequent log overflow*.

2.3.2 Page Evaporation

Recently, as the capacity of flash memory chips grows, most flash memory manufacturers have imposed a new restriction that pages in a flash block should be written in a sequential order [8]. Under the original IPL scheme, a fixed size of log area is allocated in a preset portion of a flash block—typically in highly addressed consecutive sectors. Consequently, as shown in Fig. 2, if a page in a nonfull block is updated, a new log sector will be written into the log area of the block, which may leave a region of free pages in the middle of the block that can never be written into

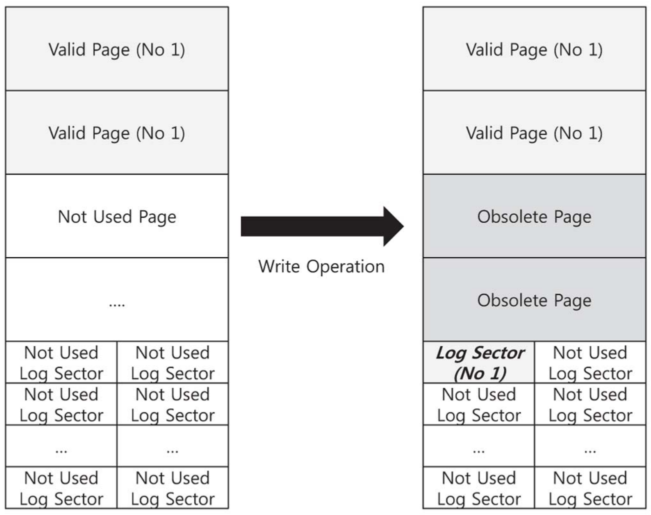

Fig. 2. Page evaporation problem.

because of the *sequential page write* requirement. We call this a *page evaporation* problem.

3 The Dynamic IPL B^+-tree Index

3.1 Structure of the Index

Unlike magnetic disk drives that have a page (or a sector) as a single unit of I/O operations, flash memory have two units of operations, namely, a page for read/write operations and a block for erase operations. As a flash-aware indexing structure, the *d-IPL B*$^+$*-tree* incorporates both the notions of nodes (a node consist of several pages) and blocks in its design of hierarchical structure.

The node-level structure of a *d-IPL B*$^+$*-tree* is exactly the same as that of a conventional B$^+$-tree, except for following definition:

N.a. Every node except the root is created by a node split when an existing node becomes full. The new node split from an existing one is first created as a form of *ghost node* and embodied later to a regular node by either a block split or a block cleansing operation. (See Section 4 for the block split and block cleansing operations.)

A ghost node defined above is essentially a group of physiological log records stored in a log area rather than a regular node stored physically in consecutive pages.

In addition to the node-level structure, the *d-IPL B*$^+$*-tree* has a hierarchical block-level structure defined as follows:

B.a. A flash block consists of a data area and a log area. The data area stores regular nodes, while the log area stores the physiological log records of the regular nodes that have been updated. The ghost nodes are also represented as physiological log records stored in the log area.

B.b. The *d-IPL B*$^+$*-tree* has only one block containing the root node. Each nonroot block stores a group of nonroot sibling nodes (either regular or ghost) residing at a consecutive location of the same level of the *d-IPL B*$^+$*-tree*.

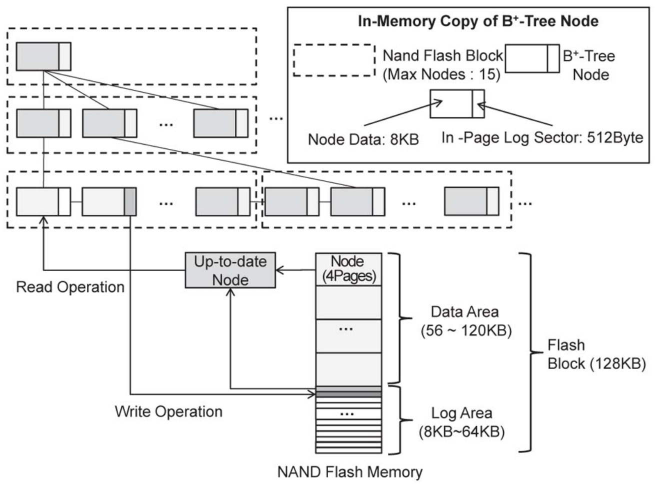

Fig. 3. Structure of the *d-IPL B*$^+$*-tree* index.

B.c. For insertion only workload, the minimum occupancy of 50 percent is guaranteed for each block except for the root block and the child blocks of the root block.

B.d. When a new node is split from an existing one, the new node is always created in the same block where the existing node is stored. If a block runs out of space for a new node, the block is split such that the requirement (B.b) is satisfied.

Following the IPL scheme, the nodes of a *d-IPL B*$^+$*-tree* index are colocated with their log records in the same flash block. Unlike the plain IPL scheme, however, the amount of log data that a flash block can store varies from block to block, because a log area stores not only log records but also ghost nodes starting right after where regular nodes are stored in the flash block.

When a block overflows with too many tree nodes, the block is split into two blocks, each with half of the nodes, satisfying the 50 percent minimum occupancy. Exceptions of the guaranteed occupancy are the root block and the child blocks of the root block. This is because the root index node is allowed to have a fewer child nodes than the other index nodes, and the number of child nodes may not be enough to fill up the child blocks of the root block.

Fig. 3 shows an example *d-IPL B*$^+$*-tree* index structure. As illustrated in the upper half of Fig. 3, *d-IPL B*$^+$*-tree* has a hierarchical node structure like a conventional B$^+$-tree, and each node of the *d-IPL B*$^+$*-tree* is stored in a flash block denoted by dotted boxes. The lower half of Fig. 3 shows an example of flash blocks and the relationships between a block and its member nodes.

3.2 Dynamic Log Area and Ghost Node

As described in Section 2.3.1, the problem of frequent log overflow is caused by a node split operation when an existing node and a new node split from it are stored in two different flash memory blocks. In order to prevent frequent log overflows, *d-IPL B*$^+$*-tree* executes a *block split* operation prior to a node split operation, if the flash memory block containing the node to be split runs out of space for a new node. In other words, if a block is full of nodes, either regular or ghost, then the block is split preemptively so that a node split operation can be carried out within a single block with enough room for a new node. Specifically, a block split operation allocates two clean flash memory blocks, and distributes the index nodes between the two blocks equally. While nodes are moved from an old block to the two clean blocks, their log records are applied to the nodes so that all the nodes are stored as the current versions in the new blocks.

Right after a preemptive block split is completed, each of the two blocks will be half full of regular index nodes, and the other (or bottom) half of each block will remain clean. Since the bottom half of each block, right next to where index nodes are stored, is clean, we can always write into the rest of the block sequentially from top to bottom without violating the sequential write restriction of flash memory. This allows us to avoid the page evaporation problem.

Note that the purpose of a preemptive block split operation is to guarantee that a node split always occurs within a flash memory block containing the node to be split. By having an existing node and a new one split from it stored in the same flash memory block, a node split operation can be represented by a few physiological log records instead of a large number of physical log records. This allows us to avoid the problem of frequent log overflows in flash blocks.

When a node is split from an existing node without causing a block split, the new node will be stored in the current block where the existing node is stored. The new node, however, will not be stored as a regular node because it will be stored in a log area in the block. Instead, the new node will be stored as a group of log record, and this type of an index node is called *ghost node* in *d-IPL* B^+*-tree*. The novelty of our approach lies in that newly split ghost nodes as well as updates made to a regular node are uniformly represented by log records.

A ghost node in a block will be embodied into a regular node by a subsequent block cleansing operation, which is invoked when a block runs out of free log sectors. If a block cleansing is invoked for a block, all the update log records in the block are applied to their corresponding index nodes to compute the current versions and all the ghost nodes in the block are embodied by creating regular nodes for them. All the current and regular nodes created by the block cleansing operation will be stored in a clean flash memory block. When a block cleansing operation completes, the log area of a block will be smaller than it used to be, because the number of regular nodes stored in the block will increase as much as ghost nodes become regular. In fact, the log area of a flash memory block is set to occupy the half of the block by a block split operation, but its size can shrink as small as just a single index node as the block is cleansed repeatedly.

3.3 Log Write Policy

As *d-IPL* B^+*-tree* adopts the in-page logging strategy, log records collected in the in-memory log sector for an index node are written to flash memory following the rules below.

- *Rule 1*. When a dirty buffer frame is evicted by a buffer replacement mechanism, the corresponding log sector is written to a log area in the corresponding flash block.
- *Rule 2*. When an in-memory log sector becomes full, the log sector is written to the log area in the corresponding flash block.

The first rule follows the traditional disk-based buffer replacement mechanism except that only the log records are written to a log area without writing the buffer frame (or index node) itself. The second rule is related to the fact that the IPL scheme assigns a fixed size in-memory log sector to each dirty page. A full in-memory log sector needs to be flushed to flash memory so that further updates on the buffer frame can be logged in a clean in-memory log sector.

Additional care should be taken for *d-IPL* B^+*-tree*, because node splitting operations should also be recorded as a log record. Once a node is split into two nodes, each of the two nodes will eventually write its log sectors into a flash block. As required by the IPL scheme, a tree node and its log sectors must be co-located in the same block. Furthermore, *d-IPL* B^+*-tree* requires that the two nodes split from the old one must reside in the same block. Consequently, the log sectors produced by a node splitting operation must be written to the same flash block. Hence, an additional rule about writing log sectors is needed.

- *Rule 3*. The log sectors involved in a node splitting operation should be written to the same log area.

4 Insertion and Deletion

The *d-IPL* B^+*-tree* index deals with both insertion and deletion operations in the same way, by storing physiological log records in the log area. In this section, we first show how an insertion operation works in a *d-IPL* B^+*-tree* index, and then describe its deletion operation. Like most conventional B^+-tree indexes, *d-IPL* B^+*-tree* performs an update request by a deletion operation followed by an insertion operation.

4.1 Insertion of an Entry

Even for a conventional disk-based B^+-tree index, an insertion is a complex operation that may involve recursive node splits and propagation of splitting key values to the ancestor nodes along the path from a leaf node. The structure of *d-IPL* B^+*-tree* makes an insertion operation even more complex, because there are a few factors concerning the log areas that should be taken into account for maintaining the integrity of block hierarchy of a *d-IPL* B^+*-tree* index.

As summarized in Fig. 4, the insertion algorithm of *d-IPL* B^+*-tree* first obtains the physical block number of an index node by invoking `getBlockNumber` function. Then, depending on the status of the node (availability of a free entry in the node) and the block (availability of a free log sector in the block), it performs an insertion operation with or without invoking *Node Split*, *Block Split*, and/or *Block Cleansing* operations.[2]

2. Note that *d-IPL* B^+*-tree* provides block cleansing operations instead of block merge operations. While a block split operation splits a flash memory block into two, a block cleansing operation is applied to a single flash block. *d-IPL* B^+*-tree* need not support a block merge operation that merges two blocks into one, because deleting an index entry is dealt with by adding a physiological log record and underflow in index nodes is allowed.

```
Input : key k, value v, node c
 1: BlockNumber $B \leftarrow getBlockNumber(c)$;
 2: if $isFull(c)$ is true then
 3:    if $isFull(B)$ is true then
 4:       $blockSplit(B)$;
 5:    end if
 6:    $nodeSplit(c)$;
 7: end if
 8: $l \leftarrow getLogSize(B)$;
 9: if $l$ has space then
10:    $insertEntry(k, v, c)$;
11: else
12:    $blockCleansing(B)$;
13:    $insertEntry(k, v, c)$;
14: end if
```

Fig. 4. Algorithm 1: Insertion

Fig. 5 shows an example of *d-IPL B*$^+$*-tree* and describes how an insertion operation is performed. In the figure, it is assumed that the maximum and minimum fanouts are 4 and 2, respectively, for both internal and leaf nodes. It is also assumed that a flash memory block can store at most two nodes and the log area of a block can have up to four log sectors. The block numbers shown in Fig. 5 are either a logical block address (LBA) or a physical block address (PBA). For ease of description, we assume that the effect of an operation is immediately written to a log area.

When new entries are inserted into a leaf node, they are written to log sectors as insertion log records. For example, in Fig. 5, suppose a new index entry with a key value 9 is inserted into a node D stored in Block 3. Since Block 3 has a free log sector available in its log area, this insertion is done just by calling the `insertEntry()` function. If two more new index entries with key values 17 and 20 are inserted into Block 4, two log sectors are appended in the log area of Block 4, as shown in the same figure.

When a target node for an insertion is already full, the target node will be split. Following the block-level requirement (B.d) given in Section 3.1, the new node split from the target node is created in the same block as the target node. Fig. 6 shows an insertion of an entry with key value 21 that results in a node split. To split the node E, a new ghost node I is created. Then, half of the entries are removed from node E and moved to node I. At the same time, two log records are produced—one for the `remove` from node E and the other for the `insert` into node I—and appended to the log sectors of corresponding nodes in the block. (See the two last log sectors in Block 4 of Fig. 6.) In this example, the log record for the new node I is written to the log area, but not in the data area of Block 4, because the node I will remain as a ghost node until it is embodied.

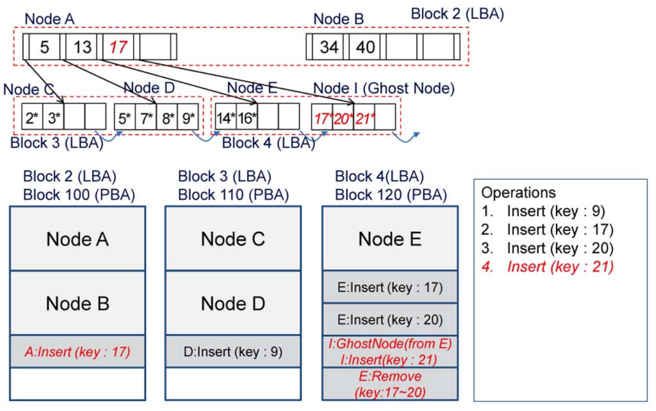

Fig. 6. Node split by an insertion.

If the block containing a target node has no space available for a new split node, then the node split operation described above must be preceded by a block split operation, which will be presented in the following section.

4.2 Block Split

A block split is triggered by a node split operation as discussed above, when the block containing a node to split is already full. Suppose, for example, in Fig. 6 node D in Block 3 is about to split and Block 3 is already full. Then, Block 3 must be split before node D is split. Fig. 7 shows the part of a resulting tree after a new Block 9 is split from the block 3.

The algorithmic description of a block split operation is given in Fig. 8. The *d-IPL B*$^+$*-tree* index allocates two free blocks, one for the new version of the old block and the other for the new block being split out. Then, it computes the new version of each node in the old block by applying the relevant log records to the old node, writes the first d nodes in the new version of old block and the remaining

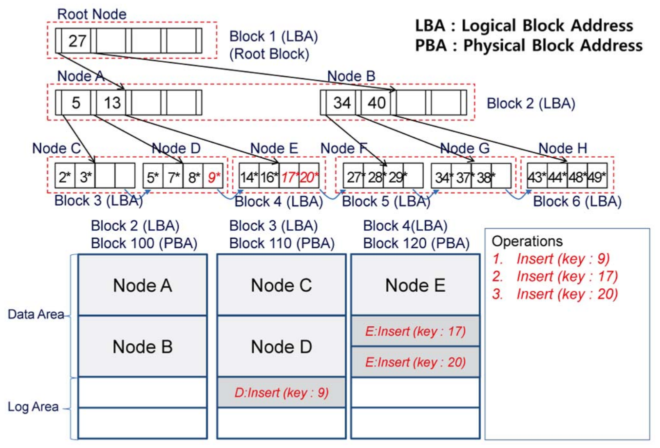

Fig. 5. Insertion of an entry.

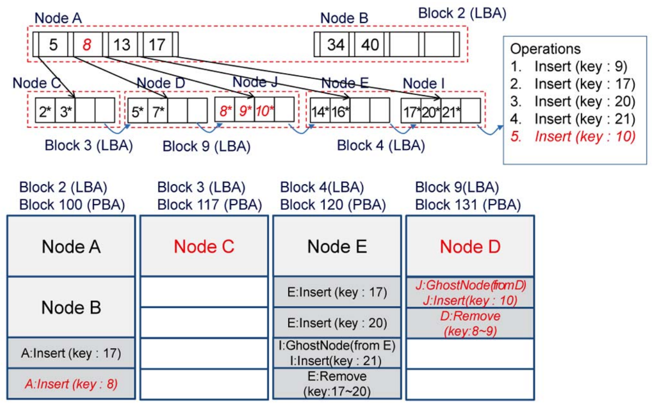

Fig. 7. Block split by a node split.

```
Input : $B_o$: an old block (PBA) to split
Output : $B_{o'}, B_n$: two blocks (PBA) with cleansed nodes
evenly distributed
 1: allocate two free blocks $B_{o'}$ and $B_n$;
    /* We assume that $B_o$ has (2d + 1) nodes
    Move the first d nodes to $B_{o'}$, and the rest to $B_n$. */
 2: for each node n in $B_o$ do
 3:    $n' \leftarrow$ apply the (any) log record(s) to $n$;
 4:    if logical position $\leq d$ then
 5:       write $n'$ to $B_{o'}$;
 6:    else
 7:       write $n'$ to $B_n$;
 8:    end if
 9: end for
10: erase and free $B_o$;
```

Fig. 8. Algorithm 2: block split.

$(d + 1)$ nodes in the new block. After copying all the nodes in the old block, the old block is erased and freed.

4.3 Block Cleansing

Following the trait of the IPL scheme, when a flash block runs out of its log sectors, all the log records stored in the block are applied to the corresponding index nodes, so that all *current* nodes are relocated to a new flash block with an empty log area. This new block with the current nodes and an empty log area will then be ready for further operations. We call this operation a *block cleansing*.

As is shown in Fig. 6, the ghost node I, which was created by a node split operation in the previous example, has a free slot to store a new entry with key value 23. However, Block 4 has already run out of free log sectors in its log area. Thus, Block 4 should be cleansed to make its log area available again for further operations. Fig. 10 shows the part of a resulting tree after a block cleansing operation is carried out. Node I, which used to be a ghost node before the block cleansing, is embodied to a regular node, and the log area of the block stores only a single log record for the insertion of key value 23 just flushed from the in-memory log sector.

The algorithmic description of a block cleansing operation is given in Fig. 9. The block cleansing algorithm allocates a new free block, computes the new versions of index nodes stored in the block by applying the corresponding log records, writes the new versions into the free block, and then finally erases and frees the old block.

4.4 Deletion of an Entry

Just as an insertion operation can cause a node to be split, a deletion operation can cause nodes to be merged. While

```
Input : $B_o$: an old block (PBA) to cleanse
Output : $B_{new}$: a new block (PBA) with current normal
nodes and clean log area
 1: allocate a free block $B_{new}$;
 2: for each node n in $B_o$ do
 3:    $n' \leftarrow$ apply the (any) log record(s) to $n$;
 4:    write $n'$ to $B_{new}$;
 5: end for
 6: erase and free $B_o$;
```

Fig. 9. Algorithm 3: block cleansing.

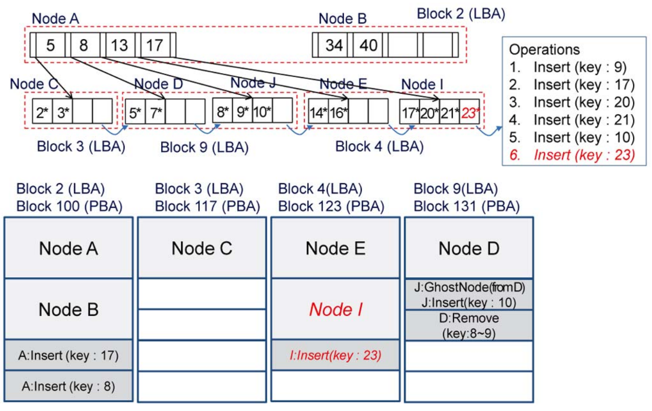

Fig. 10. Block cleansing.

merging nodes is necessary to guarantee the minimum occupancy of 50 percent for index nodes, it is not always desired in practice to merge index nodes, because merging index nodes frequently may lead to serious degradation in performance [10, Chap. 15]. Most commercial database systems (e.g., Oracle [11, Chap. 8], Sybase [12]) allow underflow in index nodes to happen so that performance penalty caused by frequent node merge operations can be avoided. Furthermore, such frequent structural changes by merging nodes might exacerbate the problem of frequent log overflow for the IPL B^+-tree indexes.

Taking this factor into consideration, the *d-IPL B^+-tree* index allows underflow to occur in index nodes and does not support node merge operations explicitly. Consequently, a deletion operation can be carried out easily by leaving a log record in the log area. When a node becomes empty by repeated deletions, it will be released and returned to a free node list by a block cleansing operation.

For the same reason, the *d-IPL B^+-tree* index allows underflow to occur in flash memory blocks by letting them occupied by regular index nodes less than 50 percent. In a rare case where a block becomes empty, the block will be released and returned to a free block list.

5 Search

Fundamentally, the search algorithm of *d-IPL B^+-tree* works the same way as a conventional B^+-tree index. It starts from the root node and traverses down the index tree to find a target leaf node. Unlike a conventional B^+-tree index, however, on a page fault, the *current* version of an index node has to be brought into a buffer frame from flash memory. Due to the IPL update logic, the current version of the node may have to be computed on the fly by applying its relevant log records to the version stored in the data area of a flash block. Under the dynamic IPL scheme, the log area can be as large as the half of a flash block. In order not to degrade the read performance of *d-IPL B^+-tree*, it is crucial to minimize the number of log sectors required to perform a read request for an index node. Therefore, in this section, we focus on the minimization of the read cost.

5.1 Layout of a Log Sector

The key to minimizing the overhead of a read operation is to fetch only the log sectors relevant to an index node

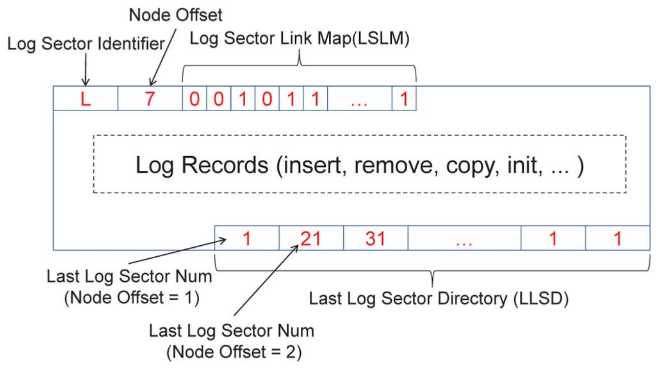
Node Offset
Log Sector Identifier
Log Sector Link Map(LSLM)
L
7
0 0 1 0 1 1 ... 1
Log Records (insert, remove, copy, init, ...)
1
21
31
...
1
1
Last Log Sector Num (Node Offset = 1)
Last Log Sector Num (Node Offset = 2)
Last Log Sector Directory (LLSD)

Fig. 11. Layout of a log sector.

being requested, rather than scanning the entire log area. In order to make such selective fetches possible, we propose an elaborate design for the layout of a log sector as shown in Fig. 11.

In addition to log records and other metadata such as a log sector identifier and the offset of a relevant node, each log sector maintains two data structures: *last log sector directory (LLSD)* and *log sector link map (LSLM)*. The LLSD keeps track of the most recent log sector for every tree node stored in the block, and the LSLM connects the log sectors belonging to the current tree node together.

When an tree node is to be read from the flash block, the current version of the node has to be computed by applying its relevant log records, which may be stored in one or more log sectors in the block. In order to compute the current version of the node without scanning the entire log area, only the log sectors relevant to the node should be chased backward from the last log sector in the log area. The LLSD records the identifier of the most recent log sector associated with every node stored in a flash block, so that the last log sector of any node in the block can be found in the directory stored in the log sector most recently written to the block.

Starting from the last log sector in a block, the log sectors relevant to a particular index node can be chased selectively via the LSLM stored in each log sector. The LSLM is a bitmap that encodes the information about which log sectors among all the previous ones are related to a particular tree node. For example, if there are five log sectors ahead of the current log sector, and only the first and the third log sectors are associated with a target tree node, then the LSLM can be encoded as `10100`. Therefore, once the latest log sector is found for a target tree node, all the relevant log sectors can be found by accessing the LSLM field of the latest log sector without scanning the entire log area.

5.2 Fetching a Ghost Node

A ghost node is yet another type of nodes stored in a *d-IPL* B^+*-tree* index. The search algorithm should be able to reconstruct a regular node from a ghost node. When an index node is to be fetched, the first step to be taken by the search algorithm is to determine whether the requested index node is a regular node or a ghost node. If the offset of the node falls within the log area of the block, then the requested node is a ghost. Otherwise, the requested node is a regular one.

The procedure for fetching the current version of a ghost node is quite different from the one for a regular node described in the previous section. Since a ghost node is created only from a node splitting operation, the first log record for the ghost node must be a physiological log that denotes copying half of the entries from the old node of which the ghost node was split from. (The information about the old node is encoded in the first log sector of the ghost node.) This implies that, in order to compute the current version of a ghost node, the old node of which the ghost node was split from must also be fetched from the flash block. Furthermore, the version of the old node must be current as of the time when the ghost node was about to be split from the old node.

While the overhead of computing the current version of a regular node is proportional to the number of log sectors associated with the node, computing the current version of a ghost node incurs additional overhead for computing the particular version of the old node the ghost node was split from. Due to this extra overhead, the overall search performance of a *d-IPL* B^+*-tree* index may deteriorate at the presence of a large number of ghost nodes. Furthermore, if a node being split is already a ghost, the node that will be created from the split will be another ghost node, and thus the read overhead for the new ghost node would be higher. However, this would happen very rarely because it is highly probable, as will be demonstrated in Section 6.2.3, that the block containing a ghost node will be cleansed before the ghost node itself is split. Furthermore, even the rare scenario can be prevented by eagerly cleansing blocks so that ghost nodes are transformed to regular ones before the ghost nodes have to be split again.

5.3 Read Optimization by Block Cleansing

A block cleansing operation, presented in Section 4.3, is invoked when a target flash memory block runs out of its log sectors. The objective of a block cleansing operation is to relocate the index nodes, either regular or ghost, stored in a block along with their log records to a new flash memory block, such that all the index nodes in the new block are *regular* and *current*, and the log area in the block becomes empty (by applying all the log records to their corresponding nodes).

Once a block is cleansed, there remains no log record in the block and any request to fetch a node from the block can be processed quickly with no additional overhead of accessing log records. To optimize the read performance of a *d-IPL* B^+*-tree* index, it may be beneficial to apply the block cleansing operation globally to all the blocks belonging to the index. This global block cleansing can be done at a regular interval, as a background process, or when the workload changes from write-intensive to read-intensive. As will be seen in Section 6, the cost of a global block cleansing was insignificant and would be amortized quickly with a read-intensive workload.

6 Performance Evaluation

This section presents the results of performance evaluation for *d-IPL* B^+*-tree*. In the first set of experiments, presented

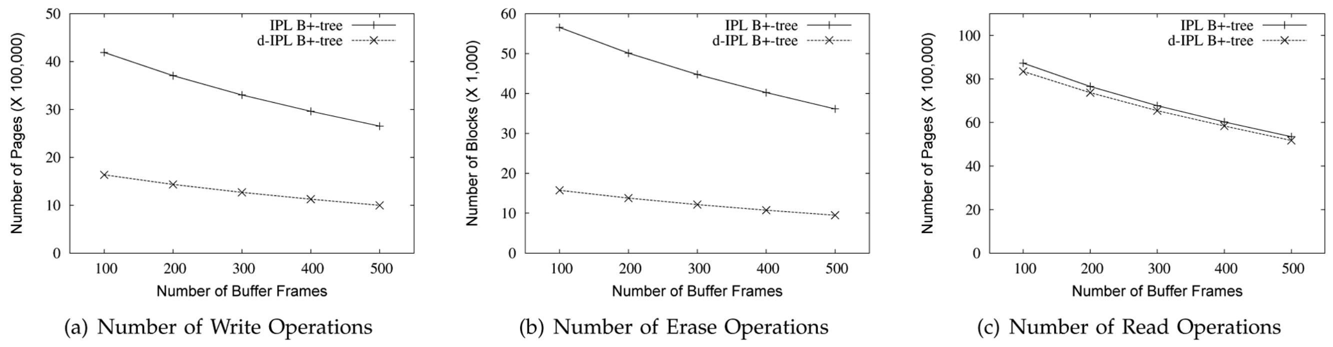

Fig. 12. Performance of random insertions. (a) Number of write operations. (b) Number of erase operations. (c) Number of read operations.

in Section 6.2, we compare *d-IPL B^+-tree* with IPL B^+-tree with respect to their performance of insertion, deletion, and search operations, and evaluate how effectively *d-IPL B^+-tree* deals with frequent log overflow and page evaporation problems without increasing the amount of reads excessively. In the second set of experiments, presented in Section 6.3, we compare *d-IPL B^+-tree* with an existing flash-aware B^+-tree index based on FTL, and demonstrate the limitations of FTL-based approaches such as FTL dependency and low space utilization.

6.1 Experimental Settings

Among the three basic operations for flash memory, namely, page read, page write and block erase, the processing time of a block erase operation is the longest and followed by that of a write and that of a read operation. The actual amount of latencies for the three operations can vary depending on the types and manufacturers of flash memory, and so are their relative ratios in latencies. In this paper, we used the frequency of the three basic operations rather than elapsed time as the performance metrics. This will help us understand the performance characteristics of different indexing methods without being interfered by such factors as caching, data bus bandwidth and channels of an actual flash memory storage device.

For the first set of experiments, we implemented IPL B^+-tree and *d-IPL B^+-tree* on top of a NAND flash simulator on a Linux platform. The NAND flash simulator was used to simulate a flash storage device, and supported 2 KB pages and 128 KB blocks. For the second set of experiments, we implemented BFTL using an FTL simulator on the same Linux platform. The FTL simulator supported a few commonly used FTLs such as FAST [13], FMAX [14], and a page-level FTL [15].

In order to obtain realistic workloads for BFTL, we built a BFTL index on a hard disk drive, and collected I/O traces from *logical sector numbers (LSNs)* at the device driver level using a block tracing tool [16]. Although the traces were obtained from a disk drive, they are identical to what would be obtained from a flash drive, because logical addresses were taken instead of physical addresses. The numbers of read, write, and erase operations were measured while the traces were fed into the FTL simulator.

The size of a B^+-tree node was set to 8 KB in all the experiments except for those evaluating the effects of different node sizes. When the size of a node was 8 KB, the maximum fanout was 840 for internal nodes and 510 for leaf nodes. The index keys were unique integers between 1 and 1,000,000.

6.2 IPL B^+-tree versus *d-IPL B^+-tree*

The first set of experiments was carried out to evaluate the performance of *d-IPL B^+-tree* and IPL B^+-tree with respect to insertion, deletion, and search operations. The objectives of the experiments were to observe how the problems of frequent log overflow and page evaporation affect the performance of insertions and deletions, and how the overhead from dynamic log areas and ghost nodes affect the performance of search operations.

6.2.1 Insertion

For the insertion test, we inserted one million index entries into each of the *d-IPL B^+-tree* and IPL B^+-tree indexes in random order. For each B^+-tree index, we repeated the same test with a varying number of buffer frames in RAM from 100 to 500 by increasing it by 100 frames at a time. The height of the tree indexes was three when the insertion was complete.

As described in Section 4, an insertion operation for a *d-IPL B^+-tree* might involve read/write/erase operations in flash memory. In order to find the target node for an entry insertion, we need to traverse down the *d-IPL B^+-tree* from the root to the leaf node, and during the traversal, several regular nodes and their relevant log pages should be brought into memory from flash memory. For an insertion to complete, we need to flush (i.e., write) the log records in the log area. In some cases, we need to cleanse old blocks and/or split blocks, which involves additional read/write/erase operations.

Fig. 12 shows the number of pages to be read and written, and the number of blocks to be erased, when a million index entries are randomly inserted. Detailed analysis of empirical evaluation will be given in the following.

First, as shown in Figs. 12a and 12b, the performance gain obtained by *d-IPL B^+-tree* over IPL B^+-tree for write and erase operations was significant. The *d-IPL B^+-tree* index outperformed the IPL B^+-tree by more than a factor of two. This clearly demonstrates how critical the problems of frequent log overflow and the page evaporation are for performance, and shows that the dynamic in-page logging scheme is a very effective solution to the problems.

TABLE 1
Node Split Number, Block Split Number and Block Cleansing Number (Buffer Frames : 500)

	Node Split	Block Split	Block Cleansing
IPL B^+-tree	2,820	N/A	36,141
d-IPL B^+-tree	2,830	277	9,212

Second, the performance gain is attributed to the fact that number of block cleansing operations is significantly reduced by *d-IPL B^+-tree*, which in turn reduces the number of read operations as well. The cost of reading a tree node from a *d-IPL B^+-tree* tree was expected higher because of the additional cost of computing the current versions of tree nodes. However, as is shown in Fig. 12c, the total number of page reads by *d-IPL B^+-tree* was even slightly less than that by IPL B^+-tree due to the reduced number of block cleansing operations.

Third, the performance of both *d-IPL B^+-tree* and IPL B^+-tree improved consistently as the number of buffer frames increased. This demonstrate that both indexes based on the in-page logging scheme are free from anomalies related to buffer management.

Table 1 compares IPL B^+-tree and *d-IPL B^+-tree* with respect to the number of node split, block split, and block cleansing operations. *d-IPL B^+-tree* required node split operations slightly more than IPL B^+-tree, and some of the node split operations caused block split operations. Since IPL B^+-tree need not perform block split operations, having to perform block split operations is clearly a disadvantage of *d-IPL B^+-tree* because block split is a costly operation. However, the node split operations incurred block split operations only at about 10 percent ratio. More importantly, *d-IPL B^+-tree* reduced the number of block cleansing operations by more than 75 percent.

Write Amplification Factor. Recently, the flash memory controller industry introduced the terminology *write amplification* [17] as a metric to represent the efficiency of FTL schemes in terms of page writes. It is commonly defined as

$$physical\ writes/logical\ writes$$

where the logical writes is the number of page writes requested by the host and the physical writes is the number of page writes actually carried out by a storage subsystem.

On the other hand, unlike the FTL approaches where any change of a page would result in a physical write into flash memory, the IPL-based approaches capture the change and result in a physical write in a smaller unit—a log sector instead of a large page. For this reason, we redefine the write amplification WA_{IPL} for the IPL-based approaches, as follows:

$$WA_{IPL} = (C_p \times B_p)/(C_n \times B_n). \quad (1)$$

Here, C_p is the number of pages written to flash memory, C_n is the number of node writes requested, B_p is the size of a page, and B_n is the size of a node. Note that the IPL scheme, with this definition, can achieve the write amplification *less than one*, which implies that the IPL can actually reduce the absolute amount of writes.

TABLE 2
Write Amplification Factor of B^+-tree Indexes

Buffer Frame	100	200	300	400	500
IPL B^+-tree	1.09	0.97	0.86	0.77	0.69
d-IPL B^+-tree	0.49	0.38	0.33	0.30	0.26

Table 2 compares the write amplification factors between the two IPL-based approaches. The IPL B^+-tree index has the write amplification of $0.69 \sim 1.09$. Such low write amplification is not surprising because the IPL scheme minimizes the amount of physical writes by storing the physiological difference between the old and new versions of a page instead of writing the new page itself. The *d-IPL B^+-tree* improves the write amplification even further to the range of $0.26 \sim 0.49$ by addressing the problems of frequent log overflow and page evaporation that cause many unnecessary write operations. In summary, even for small and random writes, the *d-IPL B^+-tree* index can achieve the write amplification twice lower than the theoretical optimum one could achieve without in-page logging.

6.2.2 Deletion

For a deletion test, we used workloads mixed with insertions and deletions of index entries for both *d-IPL B^+-tree* and IPL B^+-tree indexes. The number of insertions was a million in each workload, and the number of deletions was varied such that the ratio of insertion to deletion was 5:5, 7:3, and 9:1.

Table 3 summaries the deletion test results and compares *d-IPL B^+-tree* and IPL B^+-tree indexes with respect to the number of required read, write and erase operations as well as the sizes of the indexes resulted from the insertions and deletions. Except for the last column (denoted by Size), the table shows the number of pages read and written and the number of blocks erased during the insertions and deletions for *d-IPL B^+-tree* and IPL B^+-tree indexes. The *d-IPL B^+-tree* index evidently outperformed the IPL B^+-tree index in all three measurements with a wide margin.

On the other hand, the size of a *d-IPL B^+-tree* index was 35 to 45 percent larger than that of a IPL B^+-tree index (except for the case of 5:5 ratio where the entire set of index entries were deleted). The last column of the table shows the size of a *d-IPL B^+-tree* index or an IPL B^+-tree index in blocks resulted from the insertions and deletions. This is because *d-IPL B^+-tree* does not support operations for

TABLE 3
Number of Operation under Different Ratios of Insertions/Deletions (Insertion:Deletion)

	RATIO	READ	WRITE	ERASE	BLOCKS
IPL	(5:5)	155,900	164,106	2,735	1
	(7:3)	2,103,462	1,103,261	15,304	80
	(9:1)	4,633,756	2,314,523	31,631	146
d-IPL	(5:5)	12,774	47,390	246	1
	(7:3)	1,914,706	425,444	4,174	110
	(9:1)	4,462,403	910,851	9,051	211

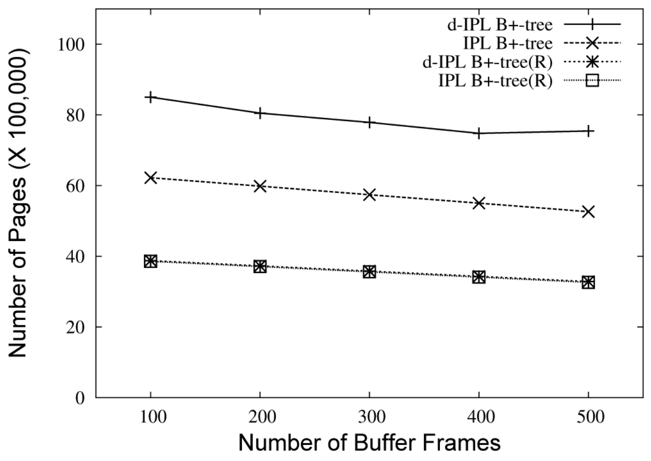

Fig. 13. Performance of random selections.

merging tree nodes nor merging flash memory blocks, as described in Section 4.4. In other words, *d-IPL B$^+$-tree* traded space utilization for processing speed.

6.2.3 Search

For a page read operation, like IPL B$^+$-tree, *d-IPL B$^+$-tree* needs to access log data in addition to a data page itself in order to compute the current version of the page. In this section, we empirically demonstrate that the read overhead by *d-IPL B$^+$-tree* is small enough to be justified by the gain in its random write performance. Moreover, because of the asymmetry in the read and write speed of flash memory, the gain in random write performance not only makes up for the read overhead but also improves the overall performance of the *d-IPL B$^+$-tree* index.

Fig. 13 shows the number of pages to be read when a million keys were searched in random order against the two indexes built in Section 6.2.1. The plots annotated by (R) are the search results from the versions of *d-IPL B$^+$-tree* and IPL B$^+$-tree indexes obtained by applying the read optimization presented in Section 5.3. For the read optimized indexes, the block cleansing operation was carried out for each of the flash memory blocks that stored the indexes resulted from a million key insertions, so that an index search does not have to access log area of the read optimized indexes. Consequently, the plots of the read optimized *d-IPL B$^+$-tree* and IPL B$^+$-tree indexes overlap completely.

When the read optimization was not applied, a search operation by IPL B$^+$-tree required reading log data as much as 4 KB on average, which amounted to about 50 percent additional reads compared with a read optimized IPL B$^+$-tree index. In the case of *d-IPL B$^+$-tree*, the log area can be as large as 64 KB, and a search operation by *d-IPL B$^+$-tree* could have required reading log data as much as 32 KB on average, which is eight times that of IPL B$^+$-tree. Besides, reading a ghost node would have increased the amount of additional page reads even further. As is shown in Fig. 13, however, the amount of additional reads incurred by *d-IPL B$^+$-tree* was just about twice that incurred by IPL B$^+$-tree. This is because *d-IPL B$^+$-tree* reduced the amount of reads effectively by utilizing the LLSD and LSLM of its log sectors.

TABLE 4
Measurement of Ghost Nodes (After Random Insertion, buffer = 500)

Metrics	
Average Number of Ghost Nodes per Block	0.28
Maximum Number of Ghost Nodes per Block	4
Number of Ghost Nodes Split Again	0

Another thing to note is that the amount of read overhead would increase if reading was done for a ghost node that had been split again before being embodied to a regular node. In our tests, however, as is shown in Table 4, the number of ghost nodes recursively split was zero. This implies that it is very improbable that a ghost node is split again before it becomes a regular node by a block cleaning operation, and the performance impact of recursively split ghost nodes is expected to be negligible.

6.2.4 Node Size

The size of an index node is one of the key factors that determine the performance of a B$^+$-tree index. In general, the use of a large index node keeps the B$^+$-tree index shallow but increases the I/O cost for accessing individual index nodes. Most contemporary flash memory devices use a smaller unit for I/O, typically 2 KB pages, than the I/O cluster chosen by most disk-based database systems, typically 8 KB or larger. In order to evaluate the impact of index node sizes on the performance of a *d-IPL B$^+$-tree* index, we repeated the same insertion test with different node sizes: 2, 4, and 8 KB. The results are summarized in Table 5. In each case, the size of a buffer pool was set equally to 800 KB, and a total of 500,000 unique keys were inserted randomly.

As is shown clearly in Table 5, the number of index nodes read or written remained largely unaffected by different node sizes, because the height of the indexes was the same in all three cases. On the other hand, the number of index nodes split decreased as much as the size of an index node (or the number of entries per node) increased.

The amount of physical reads (or the number of 2 KB flash memory pages read) increased as the size of index nodes grew. This is because reading an index node required

TABLE 5
Impact of Node Sizes on *d-IPL B$^+$-tree*

	Node sizes		
Metrics	2KB	4KB	8KB
Max. Fanout (internal)	207	418	840
Max. Fanout (leaf)	126	254	510
Index Height	3	3	3
No. of Index Nodes Read	372,135	369,967	369,197
No. of Index Nodes Written	382,423	376,366	373,500
No. of Index Nodes Split	5,721	2,850	1,390
No. of 2KB Pages Read	1,602,379	2,477,285	3,722,839
No. of 2KB Pages Written	760,530	1,058,651	704,188
No. of Blocks Split	137	124	134
No. of Blocks Cleansed	7,328	12,190	6,542
No. of Blocks Erased	7,465	12,314	6,677
No. of Blocks in Use	140	127	137

TABLE 6
Insertion Performance: BFTL versus *d-IPL* B^+*-tree*

Index Scheme	FTL (Over-provisioning 30%)		Reads	Writes	Erases	Elapsed Time	Space Usage
BFTL	Block-level Mapping	FMAX	14,533,720	2,132,676	55,353	27.87 min.	1,330 blocks
		FAST	13,246,303	824,622	12,456	20.72 min.	
	Page-level Mapping		13,832,766	1,045,478	8,168	22.13 min.	
d-IPL B^+*-tree*	N/A		8,336,982	1,608,000	15,541	16.86 min.	266 blocks

accessing all flash memory pages belonging to the index node. On the other hand, the amount of physical writes (or the number of 2 KB flash memory pages written) was independent of the size of index nodes, because writing an index node was done by writing a few log records instead of writing the entire index node. Block level operations such as block split, block cleansing, and block erasure were relatively insensitive to the size of index nodes, because the size of a flash memory block was constant regardless of the difference in node sizes.

In summary, as long as the height of a *d-IPL* B^+*-tree* index remained the same, the amount of physical reads was the dominant factor of its performance, and the best performance was attained by fitting an index node in a 2 KB page, which is the basic I/O unit for flash memory devices. Now that we demonstrated the superior performance of *d-IPL* B^+*-tree* using index nodes of 8 KB each (a default node size commonly adopted by commercial database systems) throughout the experiments presented in this section, we conjecture that the performance of *d-IPL* B^+*-tree* would have been even better if index nodes of 2 KB had been used.

6.3 BFTL versus *d-IPL* B^+*-tree*

As explained before, several flash-aware B^+-tree index structures have been recently developed to better utilize flash memory, including BFTL [3], FlashDB [4], and FD-tree [5]. They commonly assume that an underlying flash device is equipped with an FTL, and try to turn random IO operations into sequential ones using an organization similar to the log-structured file system [7]. However, this general approach requires a nontrivial amount of extra memory to cache random write requests and flush them in serialized patterns. Since BFTL is one of the first and well-known methods based on this approach, we compare the performance *d-IPL* B^+*-tree* and BFTL, and show some of the limitations of BFTL.

To measure the insertion performance, we configured the indexes the same way as the insertion experiment presented in Section 6.2.1, and set the number of buffer frames to 100. For BFTL, we tested FMAX [14], FAST [13], and a page-level mapping algorithm [15] as its underlying FTL. Although BFTL required a larger RAM space than *d-IPL* B^+*-tree*, we excluded the memory requirements in performance comparison and just compared the IO performance, so that the resource allocation was not unfavorable to BFTL.

Most contemporary flash memory SSDs come with an over-provisioned capacity to hide write latency of flash memory by utilizing the extra flash blocks internally. The overall performance of SSDs could be sensitive to the amount of over-provisioned capacity. For each underlying FTL chosen for BFTL, we set the over-provisioned capacity to 30 percent of the total capacity following the current industry trend [18], [19].

The results from the insertion experiments are summarized in Table 6. A few important observations can be made from the table. First, when FMAX was used for BFTL, BFTL was outperformed by *d-IPL* B^+*-tree* in all aspects of operations and measurements. When FAST or a page-level mapping FTL was used for BFTL, BFTL was superior to *d-IPL* B^+*-tree* with respect to the number of write and erase operations. This was mostly due to the compaction tuning parameter set to the optimal value suggested by the BFTL work [3]. However, its total elapsed time was still longer than that of *d-IPL* B^+*-tree*, because of the excessive read operations required by BFTL.

Second, the performance of BFTL, with respect to the number of read, write, and erase operations and the total elapsed time, varies significantly depending on the FTL chosen by BFTL. Such a strong FTL dependency would make the performance of BFTL unpredictable without knowing the internal workings of a chosen FTL.

Third, the size of a B^+-tree index created by BFTL was larger than the size of a B^+-tree index created by *d-IPL* B^+*-tree* by a factor of five in terms of flash memory blocks used. The low space utilization of BFTL was caused by its node management and compaction that do not necessarily separate valid log records from invalid ones, which does not allow garbage collection to work effectively. In fact, no garbage collection mechanism has been suggested by the BFTL work [3].

7 Conclusion

The hierarchical structure of B^+-tree indexes makes it difficult for the basic in-page logging scheme to deal with insertions and deletions using a few physiological log records. As is shown in the experiments, the IPL B^+*-tree* index suffers from excessive block cleansing operations caused by the frequent log overflow and page evaporation problems.

To address this concern, we propose *d-IPL* B^+*-tree* tailored for flash-aware B^+-tree indexes. We have empirically shown that the *d-IPL* B^+*-tree* index improves the utilization of flash memory blocks significantly by allocating a log area within each flash memory block dynamically. It also minimizes log overflows and keeps the write amplification factor low (often far lower than one) by introducing ghost nodes and by reducing the number of log records required for a node splitting operation.

ACKNOWLEDGMENTS

This research was partly supported by MKE, Korea under ITRC NIPA-2010-(C1090-1021-0008), MEST, Korea under Basic Science Research Program through NRF (No.2010-0025649), and MEST, Korea under NRF Grant (No.2010-0026511). This work was also sponsored in part by the US National Science Foundation Grant IIS-0848503. The authors assume all responsibility for the contents of the paper.

REFERENCES

[1] S.-W. Lee and B. Moon, "Design of Flash-Based DBMS: An In-Page Logging Approach," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07),* pp. 55-66, June 2007.
[2] G.-J. Na, B. Moon, and S.-W. Lee, "In-Page Logging B-Tree for Flash Memory," *Proc. 14th Int'l Conf. Database Systems for Advanced Applications (DASFAA '09),* pp. 755-758, Apr. 2009.
[3] C.-H. Wu, L.-P. Chang, and T.-W. Kuo, "An Efficient B-Tree Layer for Flash-Memory Storage Systems," *ACM Trans. Embedded Computing Systems,* vol. 6, no. 3, article 19, 2007.
[4] S. Nath and A. Kansal, "FlashDB: Dynamic Self-Tuning Database for NAND Flash," *Proc. Sixth Int'l Conf. Information Processing in Sensor Networks (IPSN '07),* pp. 410-419, Apr. 2007.
[5] Y. Li, B. He, Q. Luo, and K. Yi, "Tree Indexing on Flash Disks," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '09),* pp. 1303-1306, Mar. 2009.
[6] Intel, "Understanding the Flash Translation Layer (FTL) Specification,"Technical Report AP-684, Intel Corporation, Dec. 1998.
[7] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proc. ACM Symp. Operating System Principles,* pp. 1-15, Sept. 1991.
[8] Samsung Electronics, "K9XXG08XXM Flash Memory Specification,"technical report, 2007.
[9] Micron, "NAND Flash 101—An Introduction to NAND Flash and How to Design It In to Your Next Product," Technical Report TN-29-19, Apr. 2010.
[10] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, 1993.
[11] S. Dillon, C. Beck, T. Kyte, J. Kallman, and H. Rogers, *Beginning Oracle Programming.* Wrox Press, 2003.
[12] N. Ponnekanti and H. Kodavalla, "Online Index Rebuild," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD),* pp. 529-538, June 2000.
[13] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer Based Flash Translation Layer Using Fully Associative Sector Translation," *ACM Trans. Embedded Computing Systems,* vol. 6, no. 3, article 18, 2007.
[14] A. Ban and R. Hasharon, "Flash File System Optimized for Page-Mode Flash Technologies," U.S. Patent 5937425, Washington D.C., Aug. 1999.
[15] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-Performance Flash Disks," *ACM SIGOPS Operating Systems Rev.,* vol. 41, no. 2, pp. 88-93, Apr. 2007.
[16] J. Axboe and A.D. Brunelle, "Blktrace User Guide," http://kernel.org/pub/linux/kernel/people/axboe/blktrace/, Feb. 2007.
[17] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-Based Solid State Drives," *Proc. SYSTOR Israeli Experimental Systems Conf. (SYSTOR '09),* pp. 1-9, May 2009.
[18] G. Drossel, "Methodologies for Calculating SSD Useable Life," *Proc. Storage Developer Conf.,* Sept. 2009.
[19] H. Mehling, "Solid State Drives Take Out the Garbage," http://www.enterprisestorageforum.com/technology/features/article.php/3850436/Solid-State-Drives-Take-Out-the-Garbage.htm, Dec. 2009.

Gap-Joo Na received the PhD degree in computer engineering from Sungkyunkwan University, Korea, in 2011. His research interest is in flash-based database technology, flash file system, and embedded systems. Currently, he is a senior member of engineer staff at Electronics and Telecommunications Research Institute (ETRI), Korea.

Sang-Won Lee received the PhD degree from the Computer Science Department of Seoul National University in 1999. He is an associate professor with the School of Information and Communication Engineering at Sungkyunkwan University, Suwon, Korea. Before that, he was a research professor at Ewha Women University and a technical staff at Oracle, Korea. His research interest include flash-based database technology.

Bongki Moon received the MS and BS degrees in computer engineering from Seoul National University, Korea, in 1985 and 1983, and the PhD degree in computer science from University of Maryland, College Park in 1996. He is a professor of computer science at the University of Arizona, where he has been a faculty member since July 1997. His research interests include flash memory database systems, XML indexing and query processing, and information streaming and dissemination. He has served on program committees for numerous conferences and workshops as well as editorial boards and review panels for academic journals and the US National Science Foundation. He received an NSF CAREER Award in 1999 for his work on distributed cooperative web server design. He was on the research staff for Samsung Electronics and Samsung Advanced Institute of Technology, Korea, from 1985 to 1990.