Dynamic In-Page Logging for Flash-Aware B-Tree Index*

Gap-Joo Na
School of Info & Comm Engr
Sungkyunkwan University,
Suwon 440-746, Korea
factory@skku.edu

Sang-Won Lee
School of Info & Comm Engr
Sungkyunkwan University,
Suwon 440-746, Korea
wonlee@ece.skku.ac.kr

Bongki Moon
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
bkmoon@cs.arizona.edu

ABSTRACT

This paper presents *Dynamic IPL B^+-tree* (*d-IPL* in short) as a B^+-tree index variant for flash-based storage systems. The *d-IPL B^+-tree* adopts a *dynamic* In-Page Logging (IPL) scheme in order to address a few new problems that are caused by the unique characteristics of B^+-tree indexes. The *d-IPL B^+-tree* avoids the frequent log overflow problem by allocating a log area in a flash block dynamically. It also addresses elegantly the problem of page evaporation, imposed by the contemporary NAND flash chips, by introducing ghost nodes within the context of the dynamic IPL scheme. 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* index outperformed a B^+-tree with a *plain* IPL scheme by more than a factor of two in terms of page write and block erase operations.

Categories and Subject Descriptors: H.2.4 [Database Management]: System

General Terms: Design, Performance, Experimentation

1. INTRODUCTION

Due to its superiority in access latency, energy consumption and the two-fold annual increase in its density, flash memory storage devices (*e.g.*, flash memory SSD) are being adopted by storage and database vendors for large-scale enterprise servers. Recently, a new flash-based database storage model called *in-page logging (IPL)* [2] was proposed to optimize the write performance for database tables and indexes where small random writes are dominant. The key idea of the IPL scheme is to co-locate 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. Unlike database tables, however, B^+-tree indexes are hierarchical and their structures change over time by the 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 using physiological log records, because it involves more than one tree nodes that may be stored separately in different blocks. This will lead to serious concerns we call *frequent log overflow* and *page evaporation* problems.

In this paper, we present *Dynamic IPL B^+-tree* (*d-IPL* in short) as a variant of the IPL scheme tailored for 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. Furthermore, the *d-IPL B^+-tree* co-locates a new node in the same flash block as the old node from which the new node is split, so that the structural change caused by a node splitting operation can be represented by a few physiological log records in the same log area.

This simple but elegant design of the B^+-tree variant improves the performance significantly. For a random insertion workload, the *d-IPL B^+-tree* index outperformed a B^+-tree with a plain IPL scheme by more than a factor of two in terms of page write and block erase operations. It also outperformed a conventional B^+-tree running on an existing flash transaction layer (FTL) by up to an order of magnitude.

The key contributions of this work are summarized as follows. First, we discover the problem of frequent log overflows caused by node splitting in a B^+-tree under the IPL scheme. Second, in order to overcome the problem, we extend the IPL scheme such that log areas are dynamically allocated, two splitting nodes are co-located in the same flash block, and the node splits can be represented in succinct physiological log records. Third, as a way of addressing the page evaporation problem imposed by the contemporary NAND flash chips, we introduce the concept of ghost nodes within the context of the dynamic IPL scheme.

2. PROBLEM DEFINITION

The In-Page Logging (IPL) scheme takes advantage of the idea of write-reduction-by-logging and the uniform access speed of flash memory to improve the I/O performance of

* This work was partly supported by MKE, Korea under ITRC IITA-2009-(C1090-0902-0046) and the Korea Research Foundation Grant funded by the Korean Government (KRF-2008-0641). This work was also sponsored in part by the U.S. National Science Foundation Grant IIS-0848503. The authors assume all responsibility for the contents of the paper.

CIKM'09, November 2–6, 2009, Hong Kong, China.

flash memory based database systems [2]. The IPL scheme attempts to address the limitation of flash memory such as high write latency by co-locating log records and their corresponding data pages in the same flash block, so that the frequency of flash write operations can be reduced drastically at the nominally increased cost of read operations. Unlike conventional sequential logging approaches (e.g., log-structured file system[5]), log records can be written to the same flash block as the corresponding data pages without causing high latency from random writes, because the write speed of flash memory is uniform regardless of the physical location of a write operation.

Unlike database tables, however, B$^+$-tree indexes change their structures over time by the node splitting operations, which may propagate changes from one node to other nodes. The node splitting operation is difficult for the IPL scheme to deal with using physiological log records, because it involves more than one tree nodes that can be stored separately in different blocks.

A. Frequent Log Overflows

Under the IPL scheme, when a data block runs out of free log sectors, the new version of the data block is computed by applying the log records to the current pages in the block, and the new version is written to a free flash block. This operation, known as *block merge* in IPL scheme(we call this block cleansing in *d-IPL B$^+$-tree*), is costly because it involves copying an old block to a new one and erasing the old block. However, the node split operations of a B$^+$-tree may consume log sectors quickly, which leads to frequent invocation of block merge operations. We call this problem a *frequent log overflow*.

For example, when a tree node B is split from an existing node A, it may suffice to produce a few log records to describe this operation physiologically. Since the IPL scheme requires that data pages be associated with their own log sectors separately from each other, the node A needs a log record that denotes removal of half of its entries and the node B needs a log record that denotes insertion of the other half of A's entries. Although the IPL scheme requires that data pages and their log sectors are co-located in the same flash block, it is not guaranteed that the nodes A and B will be stored in the same flash block. Therefore, if the nodes A and B are written to two different blocks, then these blocks become subject to subsequent block merge operations independently from each other. If the block containing the node A is merged, then the log records of A will not be available to the node B any longer. This makes it impossible to compute the new version of node B. 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. Consequently, each node split operation will produce as many log records as the number of entries stored in a tree node, which will in turn consume log sectors in the blocks very quickly.

B. 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 [6]. 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, if a page in a non-full 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 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, flash memory have two units of operations, namely, a page for read/write and a block for erase operations. As a flash-aware indexing structure, the *d-IPL B$^+$-tree* incorporates both the notions of nodes 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 [4], except for following definition.

(N.a) 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.

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. The ghost nodes will be explained further in Section 3.2.

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 non-root block stores a group of non-root sibling nodes (either regular or ghost) residing at a consecutive location of the same level of the *d-IPL B$^+$-tree*.

(B.c) A minimum occupancy of 50% 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 co-located with their log records in the same flash block. Besides, the size of a log area is determined dynamically, because the log area is created right after the tree nodes stored in a flash block and the number of tree nodes stored in a flash block can change over time by block split or block cleansing operations.

Not only the root block but also the child blocks of the root block are an exception of the guaranteed occupancy of blocks (described in (B.c)). This is because the root node is allowed to have a fewer child nodes than the other non-root nodes and the number of child nodes is not enough to fill up the child blocks of the root block. When a block overflows

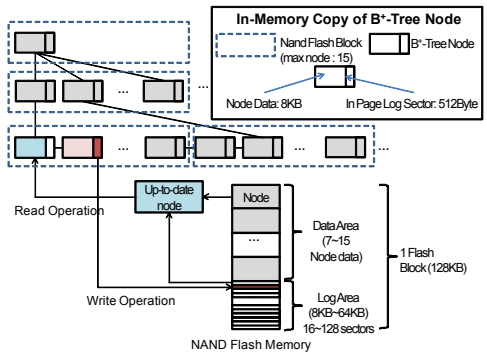


Figure 1: Structure of the *d-IPL B$^+$-tree* Index

with too many tree nodes, the block is split into two blocks, each with half of the nodes.

Figure 1 shows an example *d-IPL B$^+$-tree* index structure. As illustrated in the upper half, *d-IPL B$^+$-tree* has a hierarchical node structure like a conventional B$^+$-tree, and each node is stored in a flash block denoted by dotted boxes. The lower half shows an example of flash blocks and the relationships between a block and its member nodes.

3.2 Dynamic Log Area & Ghost Node

The log area of a flash block stores log records, and the size of a log area can vary depending on the number of regular nodes stored in the data area, the size of which is in turn determined by block split or block cleansing operations. When a block is split, each of the two split blocks is assigned the same number of tree nodes. Consequently, the storage space in each of the blocks is evenly divided into a data area and a log area of the same size. On the other hand, when a block is cleansed, the number of regular nodes stored in the block can increase by embodying ghost nodes and the size of the log area can shrink. The size of a log area can be as small as a single tree node (typically, 8 KBytes).

By having a new node split from an existing node stored in the same block as the existing one, the node splitting operation can be represented by a few short physiological log records instead of a large number of physical log records. This allows us to effectively avoid the problem of frequent log overflows in flash blocks. Besides, by allocating a log area dynamically right next to the data area within a flash block, we can always write into a flash block sequentially from top to bottom without violating the sequential write restriction and thus can avoid the page evaporation problem. Thus, the dynamic log area is fully utilized to absorb as many writes as possible for the nodes in a block, and delay the block cleansing operations until the entire log area is consumed.

When a node is split from an existing node, the new node is stored in the same block where the existing node is stored. However, the new node is not created as a regular node, but instead stored as a physiological log record in the log area of the block. Since the new node does not exist in the form of a regular node until it becomes embodied, we call it a *ghost node*. The novelty of our approach lies in the fact that both updates in nodes and newly split ghost nodes can be represented by log records.

3.3 Log Write Policy

Since the *d-IPL B$^+$-tree* adopts the in-page logging strategy, the log records collected in the in-memory log sectors attached to the tree nodes cached in memory are written to flash memory following the write 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 a log area

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 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.

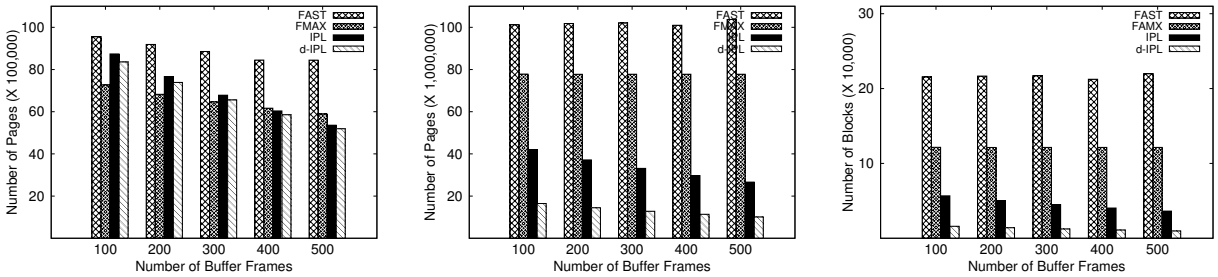
Additional care should be taken for the *d-IPL B$^+$-tree*, because node splitting operations should also be recorded as a log record. When a node is split into two nodes, each of the two nodes will eventually need to write a log sector into a flash block. As required by the IPL scheme, a tree node and its log sectors must be co-located in the same block. Furthermore, the *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 write 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. PERFORMANCE EVALUATION

In order to evaluate the performance of *d-IPL B$^+$-tree*, we perform the insertion experiment and the search experiment on four different types of B$^+$-tree indexes: conventional B$^+$-tree running on the top of either of two different FTLs, the IPL B$^+$-tree, and *d-IPL B$^+$-tree*. The two FTLs we use in this evaluation are FMAX [1] and FAST [3]. FMAX and FAST use a block-level address mapping. They avoid erase operations for out-of-place updates by using replacement blocks, which can accommodate page updates instead of updating them in-place. With a much smaller number of log blocks, FAST is known to achieve good random write performance by adopting a fully associative mapping between data blocks and log blocks.

In all the experiments, the node size is set as 8KB, and we assume the maximum entries in internal nodes is 840 and the maximum number of records allowed leaf nodes is 510. The record in leaf nodes is a pair of key and value, and the key length is 4 bytes and the length of value is 12 bytes. The key is an integer number between 1 and 1,000,000, and the total n record set is 1,000,000. For the insertion experiment, we insert one million index entries into each index in fully random order, and, for each approach, we run the same experiment by increasing the buffer frames in RAM from 100 to 500 by 100 frames. For the search experiment, we search the index built in the insertion experiment using the one million random key values. Also, we run the same experiment by increasing the buffer frames in RAM from 100 to 500 by 100 frames.



(a) Number of Read Operation (b) Number of Write Operation (c) Number of Erase Operation

Figure 2: Performance of Random Insertions

4.1 Insertion Performance

The insertion operation in *d-IPL B$^+$-tree* might involve the 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 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.

Figure 2 shows the number of pages to be read, the number of pages to be written, and the number of block erases when one million index entries are randomly inserted in four cases: that is, two cases of disk-based B$^+$-tree indexes upon FTL (denoted FAST and FMAX), IPL B$^+$-tree, and *d-IPL B$^+$-tree*. As is seen in Figure 2, both the *d-IPL B$^+$-tree* index and the IPL scheme outperform the FTL approaches considerably. This result makes us confirm that the existing FTL approaches, even though they are equipped with considerable amount of log blocks, would underperform the IPL-based approaches against small random writes. The IPL scheme can effectively overcome the erase-before-update limitations of flash memory by minimizing the amount of physical writes in the form of physiological log. Meanwhile, the *d-IPL B$^+$-tree* index outperforms the IPL scheme at least by more than two factors, which shows that the dynamic in-page logging scheme is very effective in solving the problems of frequent overflow log and page evaporation.

One interesting point in Figure 2.(a) is that the IPL-based approaches are better than the FTL approaches even in the number of page reads. At first, we expected that the IPL-based approaches would have more page reads than the FTL approaches. But, we soon come to know that the frequent block erases in FTL approaches would bring many page reads together and this is why the FTL approaches have more page reads than IPL-based approaches.

4.2 Search Performance

The search performance of the *d-IPL B$^+$-tree* index and the IPL scheme (with the legends of IPL and d-IPL) is poorer than that FTL approaches almost by 50% and by 100%, respectively. This result is very consistent with our prediction. In the IPL scheme, the size of log area is static so that only one more access to log area of 8K bytes is necessary when the node being accessed has the relevant log sector in log area. In the *d-IPL B$^+$-tree* scheme, we need to retrieve more log sectors from dynamic log area than the IPL scheme. Compared to the IPL scheme as well as the FTL approaches, we believe that this overhead in *d-IPL B$^+$-tree*'s searches be marginal against its reduction in write/erase operations Furthermore, the overhead of the *d-IPL B$^+$-tree* read operations can be avoided altogether at a nominal cost of block cleansing. Suppose, for example, the workload changes from write-intensive to read-intensive. If this change in trend is detected, the block cleaning operation can be applied globally to all flash blocks. Once this is completed, only regular nodes will be accessed from cleansed flash blocks without any log records. From our search experiment, we affirmed that the read performance of eager block cleansing even outperform that of FTL approaches slightly.

5. CONCLUSION

Unlike database tables, B$^+$-tree indexes are hierarchical and their structures change over time by the node splitting operations, which may propagate changes from one node to other nodes. The node splitting operation is difficult for the basic IPL scheme to deal with using physiological log records and lead to serious concerns called frequent log overflow and page evaporation problems. To overcome these problems, we proposed a dynamic IPL scheme for flash-based B$^+$-tree index and we empirically showed that the *d-IPL B$^+$-tree* index improves the utilization of flash blocks by dynamically allocating a log segment within each flash block, and it can also minimize log overflows by reducing the number of log records by node splitting operations.

6. REFERENCES

[1] A. Ban. Flash File System Optimized for Page-Mode Flash Technologies. U.S Patent No. 5,937,425, 1999.

[2] S.-W. Lee and B. Moon. Design of flash-based dbms: An in-page logging approach. In *Proceedings of the ACM SIGMOD*, pages 55–66, Jun 2007.

[3] 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 Transactions on Embedded Computing Systems*, 6(3), July 2007.

[4] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2003. (3rd ed).

[5] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *SOSP*, pages 1–15, Pacific Grove, CA, Sept. 1991.

[6] Samsung Electronics. NAND Flash Memory Data Sheets. Application note.