플래시 메모리 모델 상에서의 트리 인덱스 구조들에 대한 비교 및 분석
(Analysis and Comparison of Tree Indexing Structures in Flash Memory Models)

조 승 범 [†] 비니트 판데이 [††] 스리니바사 라오 사티 [†††]
(SeungBum Jo) (Vineet Pandey) (Srinivasa Rao Satti)

요 약 현재 플래시 메모리 장치들은 다양한 곳에 이용되고 있으며 기존의 외부 저장 장치들을 빠르게 대체하고 있다. 플래시 메모리 하에서 자료들을 효과적으로 저장하기 위해서는 이에 적합한 파일 시스템과 인덱스 구조를 사용할 필요가 있으며 이에 대해 많은 연구가 이루어져 왔다. 하지만 플래시 메모리 모델에 대한 이론적인 기반의 부족으로 인해 이 구조들의 성능을 서로 비교하기 힘들었으며 기존의 외부 메모리 모델은 플래시 메모리의 특징들을 반영하는 데 있어 어려움이 있어 왔다. 이 논문에서 우리는 플래시 메모리에 적합하게 제안 된 다양한 인덱스 구조들에 대해 알아보고 최근에 제안된 플래시 메모리 모델을 사용하여 이들의 성능을 분석한다.

키워드 : 플래시 메모리, B^+-트리, 트리 인덱스, I/O 모델, 플래시 메모리 모델

Abstract Flash memory devices are becoming ubiquitous and indispensable storage devices, partly even replacing the traditional hard-disks. To store data efficiently on these devices, it is necessary to adapt the existing file systems and indexing structures to work well on the flash memory, and a significant amount of research in this field has been devoted to designing such structures. But it is hard to compare these structures owing to the lack of any theoretical memory models for flash memory and since the existing external memory models fail to capture the full potential of flash-based storage devices. In this paper we study various index structures that have been shown to perform well on flash memory, and analyze them in the recently proposed flash memory models.

Key words : Flash memory, B^+-tree, tree indexes, I/O model, flash memory model

1. Introduction

Flash memory is non-volatile computer memory which can be erased and programmed. Flash memory devices have received increasing attention amongst the storage community because they are lighter, provide greater throughput and greater shock resistance while consuming lesser power as compared to magnetic disks. Flash memory has been put to use in low power devices and it is believed that they will be used for commercial large-scale usage either alongside or by replacing traditional disk-based storage. We consider the problem of storing indexing data structures on a flash based device where the index contains pointers to the data records which are stored on a different secondary disk. The disk-based B^+-tree data structure proposed by Bayer et al. [1] is a popular index structure because of its scalability and efficiency. With the rapid growth of flash memory capacity, the implementation of index

· 이 연구는 서울대학교 발전기금에서 지원된 연구비에 의해 연구되었음

† 학생회원 : 서울대학교 전기컴퓨터공학부
sbcho@theory.snu.ac.kr
†† 학생회원 : BITS Pilani Department of CS
vineetp13@gmail.com
††† 정 회 원 : 서울대학교 전기컴퓨터공학부 교수
ssrao10@gmail.com

논문접수 : 2011년 8월 17일
심사완료 : 2011년 10월 31일

structures originally designed for disk-based external memory can become a bottleneck. Despite significant work in the field of flash-specific data structures, there is a lack of a central model to theoretically analyze the performance of the several proposed structures. We discuss various B^+-tree based data structures proposed for storage and retrieval functionality and analyze them using cost models proposed recently. By means of this study, we intend to develop a framework to compare various secondary indexing structures.

Flash memory suffers from the issue of asymmetric read and write speeds unlike magnetic disks but most of the existing applications assume similar read and write speeds. As an example for database systems, a write-intensive workload will exhibit poor performance on a flash disk if it uses traditional disk-based data structures. This property has led to investigations into designing flash specific data structures which perform efficiently on flash memory devices. This work is motivated from the lack of a central model or assumptions across the data-structures designed for flash memory and we analyze the operation times of the various data structures to distinguish between various indexing structures.

The rest of the paper is organized as follows. In the following subsections we discuss details about flash memory and the models proposed to evaluate the performance of the various indexing structures proposed along with the notation used. In Section 1.3, we introduce the B^+-tree which forms the basis of most flash-specific indexing data structures and then analyze various tree based indexing structures in Section 2 using a common flash memory cost model. Other approaches towards solving the problem of storing indexing structures on flash disk are mentioned in Section 3. We conclude by providing a comparative analysis of the indexing structures in Section 4.

1.1 Two-level I/O Model

Aggarwal and Vitter [2] proposed a standard two-level I/O model to analyze the performance of an algorithm. An I/O is the operation of reading (or writing) a block from (or into) external memory. Computation can only be performed on elements in internal memory. The measures of performance are the number of I/Os used to solve a problem and the amount of space (disk blocks) used. The model consists of a CPU, a fast internal memory of size M and a slow external memory of infinite size. The CPU can only access data stored in the internal memory, and data is transferred between internal and external memories in chunks of size B where $2 \leq B \leq M/2$. The time complexity of an algorithm is measured in terms of these memory transfers called I/Os; the CPU computation time is assumed to be free.

1.2 Flash Memory

Flash memory is becoming the preferred form of storage for a wide range of devices. Flash memory scores over traditional magnetic disk-based technologies owing to the following main reasons:

- Reduced Latency. Since flash does not use any mechanical parts but electrically accesses the location containing data, the latency is orders of magnitude lesser than that of magnetic disk.
- Increased Robustness. Due to absence of mechanical parts, flash disks are more shock-resistant.
- Increased Throughput. Flash disks enjoy greater throughput than magnetic disks.

Since flash storage is non-volatile and relatively inexpensive, it is used in devices like mobile phones and hand-helds to store the data permanently. Due to its low energy consumption, high densities and low cost (compared to the cost of RAM), flash memory is commonly used in sensor networks and embedded systems. Flash memory is also becoming a popular component of large scale storage devices by replacing the traditional hard disks. Compared to magnetic disks, flash memory exhibits some unique characteristics as described below:

Asymmetric read and write speeds. The time taken to read and write to a location in flash memory is different owing to the underlying hardware structure. A write is performed in flash memory by injecting charge into a cell and waiting for it to reach a stable status. A read only reads the current status of a cell without changing it. Due to this read-write bias, the use of traditional disk-based data structures (which assume similar read-write time) on flash memory yields sub-optimal results.

- Erase before write. Any location in a flash memory cannot be written to independently owing to the existence of Erase Units as described below.
- Out-of-place update. To modify a value, the entire page has to be read and written to a new page along with the new value while the old page is invalidated.
- Weariness. Memory cells in a flash disk can be written only a limited number of times after which they wear out and become unreliable. This number lies between 10,000 and 1,000,000 [3].
- Garbage collection. Since updating a value requires making an out-of-place update, many versions of same data exist on the disk, which need to be collected and cleared. This frequent erase procedure also has an adverse effect on the lifetime of the flash memory since each Erase Unit can be written a limited number of times after which it becomes unreliable.

1.2.1 NOR and NAND flash

There are two types of flash memories: NOR and NAND. In both types, a write operation can clear bits (change value from 1 to 0) efficiently, but the only way to set the bits is to erase an entire region of memory called an *Erase Unit* (EU). The erase unit is a characteristic of the device and its size typically ranges from 16 KB to 128 KB.

- NOR flash memory: This is the older of the two types and provides random access. It can be addressed at the byte level and can be used as RAM if needed. It is very slow to write and hence, normally used as storage for static data such as codes. NOR flash memory is more expensive than NAND flash since it provides its own address bus and thus supports random-access.
- NAND flash memory: NAND flash memory was developed after NOR flash memory and has much faster erase times but it is not directly addressable at the byte level. It works much like the block devices such as hard disks and the flash disk is divided into EUs. Each EU consists of several pages and a page is the smallest size which can be read or written. Typical page size of NAND flash is 512 bytes, whereas the EU size is around 128 KB.

Flash disks come as raw NAND memory or as commercial Solid State Disks (SSDs). The SSDs come prepackaged with a *Flash Translation Layer* (FTL) (described in Section 2.3) which distributes the erase operations uniformly across the disk to prevent early failure.

1.2.2 Flash Memory Models

Flash memory allows many (up to two orders of magnitude) more random I/Os per second than the traditional hard disks. However, they cannot support general in-place updates. When writing, we distinguish between changing bits from 1 to 0 and from 0 to 1. To change a bit from 0 to 1, the device first “erases” the entire erase unit containing the given bit, i.e., all the bits in the EU are set to 1. However, changing a bit from 1 to 0 is done by writing only the page containing it, and each page can be programmed only a small number of times (typically 1 to 3 times) before it must be erased again. Erase times are relatively high (several milliseconds). Reading a bit is performed by reading the whole page containing the given bit. Reading and writing pages is relatively fast, whereas erasing an EU is significantly slower. This difference in read and write (erase) times for pages and also the bias in erasing and setting bits in a flash device requires a different memory model incomparable to other models like the I/O model and the RAM model. For these reasons, storage management techniques (algorithms and data structures) that were designed for other memory models are not always appropriate for the flash memory devices.

In the past few years, there has been a considerable amount of research on trying to characterize the flash devices in order to develop theoretical models of flash memory and thus to understand the possibilities and limitations of these devices, and also to develop efficient algorithms for them. These theoretical considerations may ultimately influence the exact architecture of future flash devices. Ajwani et al. [4] proposed the following two computational models for analyzing the performance of algorithms on flash memories:

- General flash model. This is similar to the I/O model, with the exception that read and write-block sizes are different and that they incur different costs. It assumes a two-level memory hierarchy, with fast internal memory of size M and a slow external flash memory of infinite size. Read and write I/Os from and to the flash memory occur in blocks of sizes B_R and B_W respectively. The complexity of an algorithm is $x + c \cdot y$, where x and y are the number of read and write I/Os respectively, and c is a penalty factor for writing. Typically, we assume that $B_R \le B_W < M$, and $c \ge 1$.

- Unit-cost flash model. The unit-cost flash model is the general flash model augmented with the assumption of an equal access time per element for reading and writing. In this model, the cost of an algorithm performing x read I/Os and y write I/Os is given by $xB_R + yB_W$, where B_R and B_W denote the read and write-block sizes respectively. This simplifies the model considerably, as it becomes easier to adapt external-memory results.

In this paper, we analyze the cost of performing various operations such as find, insert and delete in the different indexing structures by separately counting the number of read and write I/Os performed during the execution of an operation. Once we have these values (x and y above), we can obtain the performance in either the general-cost or the unit-cost model. We assume that $B_W \ge B_R$. Note that B_R and B_W are the parameters in the flash memory models (similar to the block-size B in the I/O model). These are simply the parameters to the theoretical models, and can be set to different values in the experiments. Choosing different values for these parameters in the practical setting changes the performance of an algorithm (data structure) implemented on a flash disk. For example, for the 64 GB HAMA flash disk, one can get optimal performance for random reads and random writes by setting these values to 128 KB and 16 MB respectively [4].

1.3 B^+-tree

A B^+-tree [5] is a modification of the well-known B-tree data structure proposed by Bayer et al. [1] which is used to manage databases efficiently. It maintains a collection of records in the sorted order of their keys and allows for find, insert and delete operations to be performed in time proportional to the height of the tree.

Every node in a B^+-tree consists of multiple key and pointer values. Specifically, a node contains d key values $K_1 \cdots K_d$ and $d{+}1$ pointer values $P_1 \cdots P_{d+1}$. If $i < j$, then $K_i < K_j$ which maintains the key values sorted in every node. The order of a B^+-tree is given by the number of keys d contained in a node, though later modifications have employed nodes of varying size [6]. Every pointer P_i, where $1 \le i \le d - 1$ points to the node which contains values between K_i and K_{i+1}. The pointers P_1 and P_{d+1} point to nodes at the next level which contain values less than K_1 and greater than K_{d+1} respectively. The pointers in the leaf nodes point to the data records on the disk. A simple B^+-tree of order 2 is shown in Figure 1.

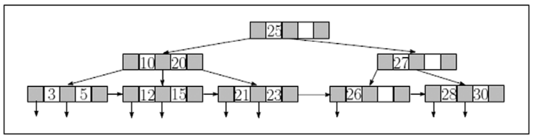


Figure 1 A B^+-tree of order 2

Let us consider a B^+-tree where each node is of size B and the total number of elements in the tree is N. The number of elements in a node and hence the branching factor at every node is maintained between $B/2$ and B. Therefore, the height of a B^+-tree is $O(\log_B N)$.

1.3.1 Operations

Search. To find a value in the tree, nodes from the root to the leaf node are visited while determining the next node to visit using the pointers. For any key K being searched, at every node, the key K_i is searched such that $K_i \le K$ and the pointer P_i is used to visit the next node. The same process continues till a leaf-node is reached. The key K is searched in the node and if any $K_i = K$, then P_{i-1} gives the location of the data on the disk, else the search fails. To enable faster range queries operations, the last pointer in the leaf node P_{d+1} can be used to point to the next leaf node (sibling pointer).

Update. To perform an insert operation, find operation is used till we reach the leaf node where insertion needs to be performed. If the node is not full, the value is inserted at its appropriate location in the sorted list of keys. Else, the node is split into two and the first value in the second node is pushed up to its appropriate position in the parent node. If the parent node becomes full, then we perform a

Table 1 List of symbols used in the text

N	Number of records in the indexing tree
M	Size of the internal memory
B	Size of a node in the indexing tree
B_R	Size of a read-block in flash
B_W	Size of a write-block in flash
h	Height of the tree
c	Cost penalty parameter in the general flash model
B_U	Size of buffer
k	parameter for the LA-tree
r	parameter for the FD-tree
t	parameter for the BFTL

similar split at the parent, and this propagates recursively to the top of the tree till the first non-full node is encountered or the root is split.

1.4 Notation

A *record* refers to a data item consisting of a key value and associated satellite data. A node in a tree comprises of (key, pointer) pairs, each such pair is referred to as an *element*. A record on the disk is pointed to by the leaf node of the index-tree. The size of a record is much larger than an element, which stores just the key value from the record. The structure is shown in Figure 2 and the list of symbols used in the paper is shown in Table 1.

The size B_U of the buffer (when needed) is decided according to the requirement of the data structure. We use B_R to denote the size of a read-block. This effectively corresponds to a page, which is the smallest unit on NAND flash devices which can be read or written. Similarly, B_W denotes the write-block size of the flash memory model. In other words, read and write I/Os from and to the flash memory occur in blocks of consecutive data of sizes B_R and B_W respectively.

For our description, we assume NAND flash in which every available EU is cleared (set to 1) to begin with, and writing a page means changing the 1s selectively to 0s (NAND flash manufacturers

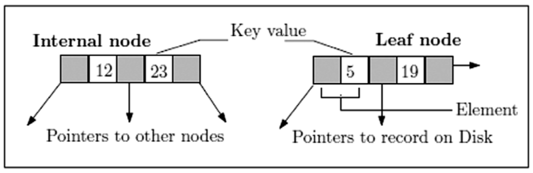


Figure 2 Structure of a B^+-tree node

typically allow a few bad EU in the disk (which are marked as such) since such a flash disk can be manufactured cost effectively). Once a page has been written, it cannot be rewritten without erasing the entire EU which has a huge cost penalty associated with it because of large size of EU. A page though can be invalidated and its contents can be written to a new page in the same or a different EU. All pointers using the physical address of the previous page (now invalidated) need to be changed accordingly.

The size of a page and EU are hardware-dependent while the sizes of a read-block and write-block are parameters to the flash memory model. We try to answer the following questions for the different indexing structures proposed:

- What should be the size of the node with regards to the read-block and write-block sizes?
- What is the number of read I/Os required to perform an operation?
- What is the number of write I/Os required to perform an operation?

We analyze the worst-case complexity of searches and calculate the amortized cost for updates. Based on the cost of operations, we predict theoretically which data structure works well for different conditions.

2 B^+-tree and proposed variants for flash memory

Various modifications have been proposed to B^+-tree to reduce the number of writes by using different techniques such as using a main memory buffer to apply a group of updates together and varying the size of the node according to the level in the tree. We discuss these ideas and provide the cost of performing searches, insertions and deletions in these data structures.

2.1 B^+-tree in flash memory

Considering $B \geq B_R$, we would need B/B_R read I/Os to read a node. To find the location of the record on disk, we will need to read $O(h)$ nodes from root till leaf node. Hence, the overall cost is $O((B/B_R) \log_B N)$ read I/Os. Since $(B/B_R) \log_B N$ is an increasing function in B, therefore to minimize the number of block-reads during search, we need to choose B as small as possible. Thus, we choose the node size B (and hence the fanout of the tree) to be equal to B_R.

Along with key values, a B^+-tree node stores pointers to child nodes. In the I/O model, we can simply update these pointers in place. On a flash memory device however, if an insertion is made at a node, then the write-block containing that node needs to be rewritten to a different location. Without loss of generality, we can assume that all the nodes in this rewritten block are pointed to by the same parent node. Now the pointers in this parent node have to be modified appropriately (again, since in-place update cannot be done). This procedure affects all nodes in the path to the root and all of them need to be modified and rewritten. For commercial database, the branching factor is high (~10,000) and the height is small (~4), but even then for every update, writing multiple nodes becomes a costly process and requires garbage collection more frequently since it uses up existing flash pages faster.

An update to a leaf node requires three phases: finding the appropriate leaf node, writing the leaf node and updating all nodes from the leaf to the root. The first phase requires the find operation described above, while the second operation costs $O(1)$ write I/Os. The third phase is most costly since we need $O(h)$ write I/Os in the path from the leaf node to the root.

This will require $O(\log_{B_R} N)$ write I/Os, in the worst-case when each node on a root-to-leaf path resides on a different write-block. Hence, the total cost of performing an update operation includes $O(\log_{B_R} N)$ read I/Os and $O(\log_{B_R} N)$ write I/Os. We now describe how to improve the write-complexity of this structure by arranging the nodes appropriately in write-blocks.

2.2 Efficient tree layout to improve the write cost

To improve the write cost of the above B-tree structure, we first choose B to be equal to B_W (instead of B_R). This improves the write cost for updates to $O(\log_{B_W} N)$ write I/Os, but increases the read cost for searches to $O((B_W/B_R) \log_{B_W} N)$ read I/Os. To improve this to $O(\log_{B_R} N)$, we implement each node of this B-tree (whose size is B_W) as a B-tree with B equal to B_R. Thus by arranging the nodes of the B-tree in this layout (as shown in Figure 3), we can perform searches by $O(\log_{B_R} N)$ read I/Os, and updates by amortized $O(\log_{B_W} N)$ write I/Os.

2.3 Flash Translation Layer (FTL)

One basic issue with tree structures on flash disk is the cost of update of a single node. Since in-place update is not allowed by flash memory, updating a page requires one to invalidate that page and write the contents (along with modifications) to a different page altogether. Although this looks like a small extra cost, it becomes a basic issue for structures which use pointers to physical address. Technically, an in-place update can be

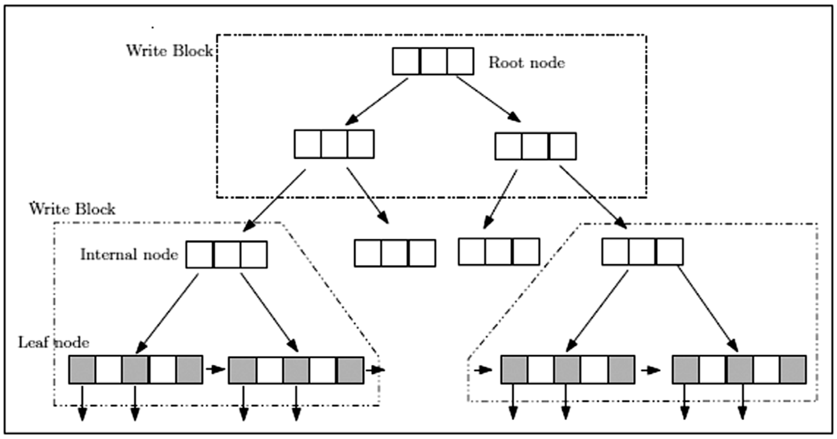


Figure 3 Efficient tree layout. Nodes are stored on disk in top-down manner. Dashed area represents a write block

performed and it would require reading all the pages of the EU into memory, resetting the EU to all 1's (which is a costly operation) and then writing back the old pages along with the modified value. Apart from being expensive, frequently erasing an EU for small updates brings down the life of the disk and the EU might become unreliable after few updates.

Flash Translation Layer (FTL) [7] solves the problem of updating multiple nodes for one update by using logical addresses instead of physical addresses for pointers and maintaining a logical to physical address map. This allows any update to write just the target node and not all the nodes up to the root node. The logical to physical mapping can be stored in two ways:

- By allocating some EUs on the disk solely to maintaining the table which are updated with every modification to the table. Although it sounds reasonable, the frequency at which EUs will get used might lead to weariness.
- Every physical page contains logical id and the actual table is stored in RAM. Whenever flash disk is plugged in, this table can be constructed. Due to the weariness of the flash disk and the large cost associated with erasing an EU, the second option sounds more practical since it uses faster read operations to construct the table.

FTL enables a wear-leveling policy to be used, which distributes erases uniformly across EUs. It can also provide a sector based access to the flash disk so that the existing magnetic-disk based algorithms can be directly implemented on the flash disk. When a standard B^+-tree is used along with FTL, the cost of find operation remains the same, i.e., $O(\log_{B_R} N)$, read I/Os, while updates require $O(1)$ write I/Os as against $O(\log_{B_W} N)$, write I/Os for B^+-tree without FTL.

2.4 Lazy Adaptive tree

The Lazy Adaptive tree proposed by Agrawal et al. [8] is a B^+-tree with flash-resident buffers at every k^{th} level from the root. The idea is to avoid high update cost of flash memory by grouping together update operations in buffers. Whenever search or update operation is performed in the tree, the optimal online algorithm named ADAPT dynamically determines whether to empty the buffer and update the contents of the descendant nodes or append the update request to the buffer. The ADAPT algorithm estimates benefit of emptying buffer using buffer size and buffer scan cost at each lookup and if this benefit is larger than the cost of emptying the buffer, then the ADAPT algorithm empties the buffer at that lookup. We assume that the node size of the B^+-tree (and hence also the fanout of the tree) is equal to B_R.

2.4.1 Cost of operations

Our analysis makes a simplifying assumption that the effective buffer size B_U is same at every level,

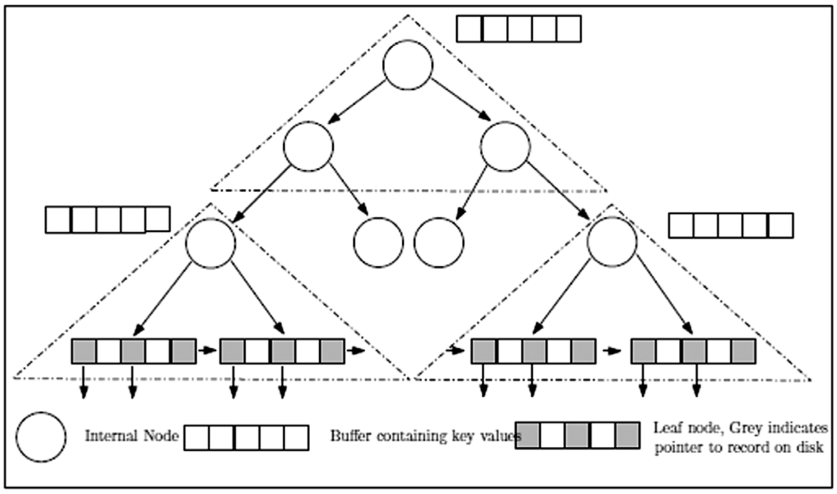

Internal Node
Buffer containing key values
Leaf node, Grey indicates pointer to record on disk

Figure 4 Lazy Adaptive tree

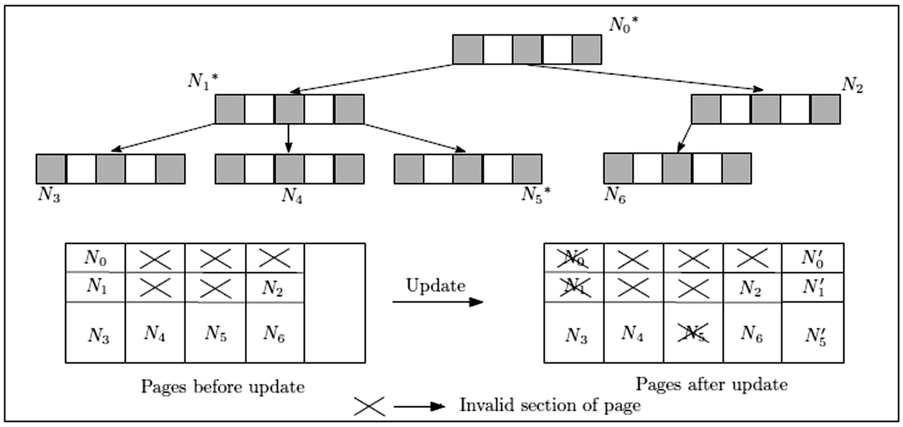


Figure 5 Update in a μ-tree. The nodes marked with * are the ones updated. The pages are shown before and after the update. New nodes are marked with

which is not the case as described by Agrawal et al. [8]. We also assume that the ADAPT algorithm does not work during a find operation, and the algorithm empties the buffer only when the buffer is full. We analyze the performance assuming an FTL running on top. Without FTL, the performance (with our simplified assumptions) is worse than that of the standard B^+-tree.

To search for given key, we perform normal search operation as in a B^+-tree and at every k^{th} level we search in the buffer as well. The cost of searching all the nodes along a path is $O(\log_{B_R} N)$ and the cost of searching all the buffers along the search path is $O((B_U / k \cdot B_R) \log_{B_R} N)$. Hence the total number of read I/Os during search is $O((1 + (B_U / k \cdot B_R)) \log_{B_R} N)$.

To perform an update, we simply add an update record to the buffer at the root. If the at any internal node gets full, then we flush all the records in that buffer to the next lower-level buffers (or to the leaves, if there are no buffers below). Since the branching factor of each node is $\Theta(B_R)$, the number of next-level buffers is $O((B_R)^k)$. Assuming that $B_U \leq (B_R)^k$ (otherwise, the search cost will be quite high), in the worst-case each update in the buffer may have to be pushed to a different next-level buffer. Thus the cost of flushing the buffer is $O(B_U)$ write I/Os, apart from the read cost. Thus, each update takes $O(1)$ write I/Os to be pushed to the next-level buffer, which is k levels below, or in other words $O(h/k)$ write I/Os to be pushed from the root to a leaf. And the read cost is at most one read I/O per level, or $O(h)$ read I/Os overall. Therefore the amortized cost of inserting an element into the LA-tree is given by $O((\log_{B_R} N)/k)$ write I/Os (assuming that B_U is not too large compared to B_R), and $O(\log_{B_R} N)$ read I/Os.

2.4.2 Comparison with B^+-tree

LA-tree performs better if the number of write I/Os to perform an update in LA-tree is smaller than that of B^+-tree, i.e.,

$$(\log_{B_R} N)/k < \log_{B_W} N \Rightarrow$$
$$\log B_W < k \log B_R \Rightarrow B_W < (B_R)^k.$$

For most practical values of B_R and B_W, the above inequality holds even for $k = 2$, and hence LA-tree performs better than B^+-tree. The above analysis does not take into account the read-cost (i.e., the number of blocks read) while performing an update. The read-cost is more for the LA-tree since in addition to searching in the nodes at each level, it also has to search in the buffers. Also, for the same reason, searches in the LA-tree are slower when compared to the B^+-tree.

The search and update costs of LA-tree are both inversely proportional to the parameter k. Hence, it would seem to make sense to choose k as large as possible (i.e., equal to the height of the tree), in which case, the performance matches that of B^+-tree. But the actual performance of LA-tree is better than that of B^+-tree in practice as the buffers are flushed adaptively during both searches and updates, and buffer sizes are also not the same for all the nodes.

2.5 μ-tree: minimally updated tree

The minimally updated tree or μ-tree proposed by Kang et al. [6] is a balanced tree similar to B^+-tree, which reduces the number of pages written by using varying sizes for nodes depending on their distance from the root. It requires a single flash write operation to perform an update to the tree, if no nodes are split during the update.

A page is occupied by root to begin with, and the size occupied by root gets decreased by a factor of two for every increase in height of the tree, as shown in Figure 6. The size of a node depends on its level and height of the tree. Since the nodes have lesser size as their height increases, the number of children varies as well. The μ-tree is stored on the flash disk as follows: Each read-block corresponds to a leaf-to-root path in the μ-tree, and has the capacity to store all the nodes along this path. The nodes are stored in the blocks such that no node is stored on more than one block.

To perform a search, we simply follow a root-to-leaf path by reading the blocks containing the corresponding nodes. To perform an update, we first find the leaf x in which the update has to be performed. We then write all nodes on the root-to-leaf path to x on a new block. The previous copies of these nodes (along the root-to-leaf path to x) are effectively invalidated as they are not reachable from the root of the tree anymore.

Since the size of a node decreases exponentially as we move from a leaf to the root and all the nodes in the path are contained in one read-block, log B_R is an upper bound on the height of a μ-tree. The exact height of a μ-tree is given by the following equation [6]:

$$h = \log\frac{B_R}{\sqrt{2}} - \sqrt{\left(\log\frac{B_R}{\sqrt{2}}\right) - 2\log\left(\frac{N}{2}\right)}$$

This gives the following upper bound on the maximum number of records that can be stored in a μ-tree, if we assume that any root-to-leaf path fits in a single read-block (using the fact that the quantity under the square-root in Equation 1 is non-negative):

$$N \le 2^{(1/2)(\log B_R)^2}$$

2.5.1 Cost of operations

Search. Searching in a μ-tree requires reading the nodes along a root-to-leaf path. Since all these might lie in different pages in worst-case, we might need $O(h)$ read I/Os in the worst case.

Update. Update in a μ-tree involves $O(h)$ read I/Os to perform search and 1 write I/O, since entire path is contained in one read-block. The read and write costs are essentially same those of the B-tree with FTL, although μ-tree does not have the overhead of an FTL.

2.6 FD-tree

The FD-tree proposed by Li et al. [9] consist of multiple levels, denoted as $L_0, L_1, \cdots, L_{h-1}$. At the top level, L_0, it has a head tree which is a small (i.e., constant height) B^+-tree with node size equal to the read-block size. Each of the other levels, $L_1, \cdots, L_{h-1}$, is a sorted run of key values stored in contiguous pages. Each level of the tree has a capacity which is the maximum number of elements that can be stored in that level. The ratio of capacities between any two adjacent levels is same, and is equal to r, for some parameter r.

To support efficient searches, in each level the FD-tree stores entries called fences that point to the immediate lower level. Given a search key x, we call the page at level L_i that contains the largest key less than or equal to x as the target page at level L_i. The fences are chosen in such a way that given a search key x, once we find the target page at level L_i, the fence pointer with largest key value less than or equal to x in that page points to the target page at level L_{i+1}.

To search for a given key x, we first search for

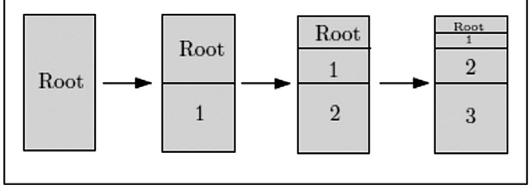


Figure 6 Resizing a node in μ-tree as the height increases from 1 to 4

x in the head tree, and then follow the appropriate fence pointers to find the target pages at each level, and search in those target pages. To perform an insertion, we first insert the element into the head tree. If at any time, the number of elements in any level L_i, for $0 \le i < h-1$, exceeds its capacity, the FD-tree merges the elements of L_i with the elements in the adjacent lower level L_{i+1} into a single sorted run (stored in contiguous pages). Also FD-tree deals deletion as a special case of insertion by inserting some entry to be deleted (called Filter entry) to the head tree. In the merge process, the tree uses only sequential writes and random writes occur only in head tree. FD-tree performs better than original B^+-tree for update operation in flash memories by converting random writes (which are typically slow) to sequential writes.

2.6.1 Cost of operations

Let L_i denote the capacity of level i, and let r be capacity ratio between adjacent levels, i.e., for $0 \le i \le h-2$, $|L_{i+1}| = r \cdot |L_i|$. If the FD-tree contains N keys, then the height h (i.e., the number of levels) of the FD-tree is $O(\log_r N)$.

Search. The search procedure first searches the head tree, which requires $O(1)$ read I/Os, and then accesses one read-block at each of the h levels. Thus the search cost is $O(\log_r N)$ read I/Os.

Update. Li et al. [9] show that the update cost of FD-tree is amortized $O((r/(f-r)) \log_r N)$ sequential I/Os, where f is the size of the read-block. But this cost is in terms of read-blocks. Thus to obtain the actual update cost of FD-tree, we need to divide this by B_W/B_R. By choosing $r = \Theta(B_R)$ and such that $r \le f/2$, we get the update cost to be amortized $O((B_R/B_W) \log_{B_R} N)$ sequential write I/Os.

2.6.2 Comparison with B^+-tree

The search of FD-tree is same as the search of B^+-tree. The update cost of FD-tree is better than that of B^+-tree if

$$(B_R/B_W) \log_{B_R} N < \log_{B_W} N \Rightarrow$$
$$B_R/\log B_R < B_W/\log B_W.$$

Since the function $f(x) = x/\log x$ is an increasing function, for $x > 0$, the above inequality is always true. Thus the update performance of FD-tree is better than that of the B^+-tree. In addition, the FD-tree only uses sequential writes.

3. Alternate techniques

Apart from the B^+-tree based structures discussed in Section 2, several other ideas have been suggested to maintain indexes efficiently on flash disks. In this section, we discuss some of these alternate techniques.

3.1 In-page logging

In-page logging uses some pages in every EU to maintain a log of changes rather than modifying and rewriting a node with each update [10]. To provide better efficiency, a buffer is maintained in main memory which acts as a reservation buffer for the updates to be committed to the tree on flash disk, and as a cache to enable quicker reads. Assuming *update locality*, i.e., when several updates have to be performed on the same page, the approach performs well, but in the worst-case where every update in buffer causes a change on a different page of the EU, the performance is comparable to that of standard B^+-tree.

3.2 BFTL

BFTL is an efficient B-tree layer for flash memory storage systems which looks to combine the efficiency of B^+-tree index structures with the block-emulation provided by flash translation layer [11]. A layer called BFTL is proposed which manages B^+-tree indexes at the OS level using Flash Trans-

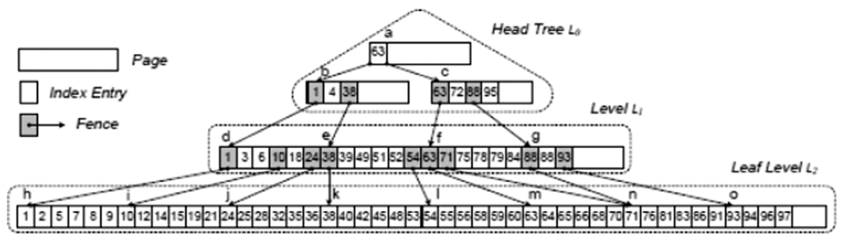


Figure 7 FD-tree [9]

lation Layer present at the device level. BFTL ensures that all updates to the index are not instantaneously committed to the disk but are kept in a buffer of fixed size, which is flushed out when full. The design of BFTL makes it appropriate to be used with log-type filesystems. BFTL works over FTL and provides the functions at filesystem level to create and maintain B^+-tree index structures. B^+-tree index services requested by the upper-level applications are handled and translated from file systems to BFTL and then block-device requests are sent from BFTL to FTL. Hence, BFTL adds an interface in between without requiring any changes to be done to FTL, as shown in Figure 8.

BFTL introduces two new components. A *Node Translation Table* (NTT) is kept in memory to store the physical address of all nodes and the data units related with it in a sequential list. A *reservation buffer* is used to store modifications to nodes in main memory before they are flushed out and applied to flash memory. When nodes are modified, deleted or inserted, they are not written directly to disk but stored in reservation buffer as dirty record. Deletions are handled by adding *invalidation dirty records* to the reservation buffer. A dirty record contains the primary key and data to be stored on secondary disk.

We use t to denote the upper bound on the length of entry for a node in the Node Translation Table. The find operation now needs to look through every node's entry in the NTT apart from the node itself to find the index units corresponding to updates to that node, which are stored on the disk.

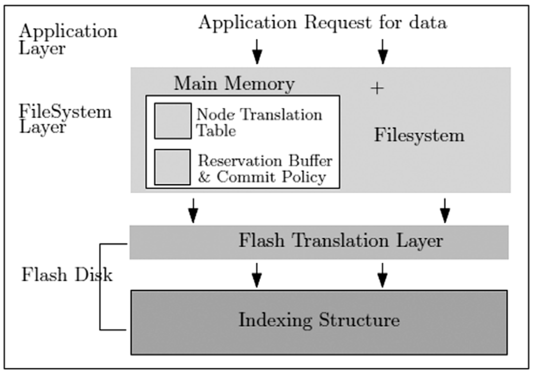


Figure 8 Architecture of BFTL

Thus we need $O(t)$ read I/Os for every node searched and in worst-case, the cost of find operation needs $O(t \log_{B_R} N)$ read I/Os.

Performing an insert operation involves searching for the node and then performing an update which is first stored in the reservation buffer as a dirty record, and then flushed out to the disk as an index unit, taking constant time. When the NTT entry for a node gets full, then the index units are read and merged together into the node, thus updating the current node and all the nodes till the root. This happens after t steps on an average and involves $O(\log_{B_R} N)$ write I/Os in the worst-case. Therefore the update cost includes $O(t \log_{B_R} N)$ read I/Os and $O((1/t) \log_{B_R} N)$ write I/Os.

Once the reservation buffer gets full, index units are constructed for every dirty record. Multiple index units can be stored on one sector as opposed to conventionally storing one node per sector. So, when the buffer is emptied, index units are created which are stored on the disk using FTL using a commit policy which is heuristics-based since the problem of packing index units on minimum number of sectors is NP-hard. The other data is modified or written accordingly on the data disk. The address of the index units is appended to the corresponding entry in the NTT. So, when performing find operation, the node is searched and then all the sectors in the node's entry in NTT are checked. This makes find costlier due to greater number of reads, but BFTL aims to lower the number of writes for greater number of reads.

The reservation buffer cannot be huge because it is stored in the main memory. BFTL assumes that there will be enough space to contain NTT in RAM. Since it consumes internal memory in embedded systems, it might not be suitable for usage in devices with low internal memory.

3.3 FlashDB

Despite the modifications proposed for B^+-tree for flash devices, Nath and Kansal [12] claim that the performance gain is not the same for different flash devices. They make the following observations:

1. Performance gain depends on factors such as read-to-write ratio and data pattern of the workload.

2. Gain also depends on the flash device since the read/write costs and their ratios differ significantly across flash packages. As an example, a Compact Flash (CF) card has write-to-read ratio of 2, while a mini Secure Digital (SD) card has write-to-read ratio of 200.
3. Re-writing to the same (logical) page address is slower than writing to a new page address in sequential address while the variation is small for a read operation.

The chief contributions of FlashDB are:

1. Design of a self-tuning index which dynamically adapts its storage structure to workload and underlying storage device.
2. A framework to determine the optimal size of the index node to minimize the latency and energy consumption.

A tree called *B^+-tree(ST)* is constructed (where ST stands for Self Tuning) in which a node can exist in two modes: *Log* or *Disk*. Log mode refers to the structure similar to BFTL where each update to a node is written as a separate node entry and to read a node, all its entries are parsed. Disk mode refers to storing a node on contiguous pages so that reading a node involves sequential reads. A node can switch between the two modes depending on the workload and the properties of the flash-disk. Disk mode is favored for a read-intensive workload while Log mode works better for a write-intensive workload.

3.4 Other similar approaches

BFTL and FlashDB can reduce the write I/Os in flash memory. But these structures use large amount of internal memory and have poor search time. The *MB-tree* (modified B-tree) proposed by Roh et al. [13] is an extension of B-tree index which reduces not only the overall write I/Os but also the internal memory usage and the search time. The MB-tree reduces write I/Os by writing many entries which belongs to the same leaf node at once. Also MB-tree stores entries and logical structure information on flash memory to reduce the internal memory usage.

Lee et al. [14] proposed a buffer management scheme named *IBSF* to improve the search time and reduce the internal memory usage in BFTL. An index buffer is used to store the index units, which reflects the modified B-tree node when updating the records. When the index buffer gets full, IBSF collects index units which will be in the same B-tree node and stores in one page so it does not need node translation table which can be an overhead in search time. Also IBSF eliminates redundant index units in index buffer to save additional write operations in BFTL. Experimentally, IBSF has been shown to perform better than BFTL in terms of read, write and erase operations.

Xiang et al. [15] proposed a reliable B-tree implementation named *RBFTL*, which is a B-tree layer system for NAND flash memories which is placed between the application layer and the FTL. This reduces the loss of records when a system crash occurs, which is a problem in BFTL. RBFTL has similar structure as IBSF but the index buffer keeps a fixed number of index units and uses a NOR flash memory to store the backup index units before they are written to NAND flash memory. This minimizes the loss of data when a system crash occurs.

3.5 Lazy Update tree

The Lazy Update tree proposed by On et al. [16] uses main memory for two goals. One is for caching the recently used nodes as in normal B^+-trees, and the other is for buffering update requests into a buffer called lazy-update pool. The lazy-update pool contains update requests which are grouped by same target nodes. When an update operation is performed in the tree, if the lazy-update pool is not full, then the request is added to the pool. Otherwise, the tree selects one group as the victim by some commit policy and commits these victims to the nodes. The experimental results in [16] show that lazy update method along with a well-designed commit policy improves the update performance of the traditional B^+-tree while preserving the query efficiency.

4. Discussion and Conclusions

We discussed the various B^+-tree based indexing data structures which have been designed specifically for flash disks. We analyze the cost of performing search and update operations in the recently pro-

Table 2 Complexity of operations in proposed index structures in terms of the number of read and write I/Os. The update cost does not include the search cost. h represents the height of the tree

Data Structure	Search coast (read I/Os)	Update cost (write I/Os)	Reference
B^+-tree	$O(\log_{B_R} N)$	$O(\log_{B_W} N)$	[5]
B^+-tree (w/ FTL)	$O(\log_{B_R} N)$	$O(1)$	[7]
LA-tree (w/ FTL)	$O((1 + (B_U / k \cdot B_R)) \log_{B_R} N)$	$O((\log_{B_R} N)/k)$	[8]
FD-tree	$O(\log_{B_R} N)$	$O((B_R/B_W) \log_{B_R} N)$	[9]
μ-tree	$O(h)$	$O(1)$	[6]

posed flash memory models. Table 2 summarizes the performance of the index structures that we have analyzed in terms of the read and write I/Os. From this table, one can easily obtain the complexity of the search and update operations in either the general-cost model or the unit-cost model. The search cost for all the structures is essentially the same. The update cost of μ-tree and B^+-tree with FTL is better than that of the LA-tree and the standard B^+-tree without FTL. Note that the our analysis of LA-tree made several simplified assumptions, and hence the practical performance of LA-tree may be better than that of μ-tree or B^+-tree with FTL for some workloads. When $(B_W/B_R) \geq \log_{B_R} N$, the FD-tree outperforms all the other structures, as its amortized update cost is less than 1, and also since it has very few random writes. Otherwise, μ-tree gives the best performance, without FTL.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pp.107-141, New York, NY, USA, 1970.

[2] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116-1127, Sep. 1988.

[3] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138-163, Jun. 2005.

[4] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In Jan Vahrenhold, editor, SEA, volume 5526 of *Lecture Notes in Computer Science*, pp.16-27, Springer, 2009.

[5] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys*, 11:121-137, Jun. 1979.

[6] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree: an ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT)*, pp.144-153, New York, NY, USA, 2007.

[7] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55:332-343, May 2009.

[8] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy adaptive tree: an optimized index structure for flash devices. *Proceedings of the VLDB Endowment*, 2:361-372, Aug. 2009.

[9] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. Proceedings of the VLDB Endowment, 3:1195-1206, Sep. 2010.

[10] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data (SIGMOD)*, pp.55-66, New York, NY, USA, 2007.

[11] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems*, 6, Jul. 2007.

[12] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN)*, pp.410-419, New York, NY, USA, 2007.

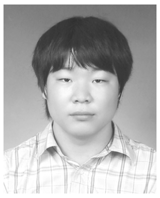
[13] Hongchan Roh, Woo-Cheol Kim, Seung-Woo Kim, and Sanghyun Park. A b-tree index extension to enhance response time and the life cycle of flash memory. *Information Sciences*, 179(18):3136-3161, 2009.

[14] Hyun-Seob Lee and Dong-Ho Lee. An efficient index buffer management scheme for implementing a b-tree on nand flash memory. *Data Knowledge Engineering*, 69(9):901-916, 2010.

[15] Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, and PengWei. A reliable b-tree implementation over flash memory. In Roger L. Wainwright and

Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC)*, pp.1487-1491, 2008.

[16] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update b$^+$-tree for flash devices. In *Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware (MDM)*, pp.323-328, Washington, DC, USA, 2009.

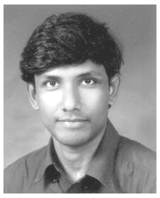


조 승 범

2009년 KAIST 전산학과(학사). 2011년 KAIST 전산학과(석사). 2011년~서울대학교 컴퓨터공학부 박사과정. 관심분야는 데이터 구조, 알고리즘

Vineet Pandey

2011 B.E(Hons.) BITS Pilani, Computer Science. since 2011-Member of Technical Staff, Advanced Technology Group, NetApp, Bangalore. Research interests: Algorithms, Flash memory, Storage systems



Srinivasa Rao, Satti

1995 BTech NIT Warangal, Computer Science and Engineering. 1997 MSc Institute of Mathematical Sciences, Theoretical Computer Science. 2002 PhD Institute of Mathematical Sciences, Theoretical Computer Science. since 2009~Assistant Professor, Seoul National University, School of Computer Science and Engineering. Research interests: succinct data structures, text indexing, algorithms for external memories