Tree Indexing on Flash Disks

Yinan Li, Bingsheng He†, Qiong Luo, and Ke Yi

Hong Kong University of Science and Technology
{yinanli,saven,luo,yike}@cse.ust.hk

***Abstract*— Large flash disks have become an attractive alternative to magnetic hard disks, due to their high random read performance, low energy consumption and other features. However, writes, especially random writes, on the flash disk are inherently much slower than reads because of the erase-before-write mechanism. To address this asymmetry of read-write speeds in indexing on the flash disk, we propose the FD-tree, a tree index designed with the logarithmic method and fractional cascading techniques. With the logarithmic method, an FD-tree consists of the head tree – a small B+-tree on the top, and a few levels of sorted runs of increasing sizes at the bottom. This design is write-optimized for the flash disk; in particular, an index search will potentially go through more levels or visit more nodes, but random writes are limited to the head tree and are subsequently transformed into sequential ones through merging into the lower runs. With the fractional cascading technique, we store pointers, called fences, in lower level runs to speed up the search. We evaluate the FD-tree in comparison with representative B+-tree variants under a variety of workloads. Our results show that the FD-tree has a similar search performance to the standard B+-tree, and a similar update performance to the write-optimized B+-tree variant. As a result, FD-tree outperforms all these B+-tree index variants on both update- and search-intensive workloads.**

I. Introduction

The flash disk, or flash Solid State Drive (SSD), has emerged as a viable alternative to the magnetic hard disk for non-volatile storage. The advantages of the flash disk include high random read performance, low power consumption and excellent shock resistance. Moreover, the capacity of the flash disk doubles every year [1]. Flash disks have been considered recently as a new storage device that can replace magnetic disk and achieve a much higher performance for enterprise database servers [2], [3], [4]. Since tree indexes are a primary access method in databases, we study how to adapt them to the flash disk exploiting the hardware features for efficiency.

The flash disk is a type of electrically-erasable programmable read-only memory (EEPROM). Unlike magnetic disks where seek and rotational delays are the dominant cost of reading or writing a page, the flash disk has no mechanic movement overhead. As a result, random reads of a flash disk are up to two orders of magnitude faster than a magnetic disk [5]. However, due to the erase-before-write mechanism of the flash disk, each write operation may require erasing a large block, called the *erase block*. This mechanism makes random writes almost two orders of magnitude slower than both the random read and the sequential access patterns. As shown in Table I, our Samsung 32GB flash disk provides 3100 IO/sec for random reads, but only 25 IO/sec for random writes. While high-end flash disks with a better random write performance have recently been announced, such as the Intel Extreme series SSD [6], their random writes are an order of magnitude slower than random reads.

TABLE I
Performance Comparison of Random Access Patterns (IO/Sec)

	Samsung 32GB SSD	Intel Extreme 32/64GB SSD	Segate 7200RPM SATA Magnetic Disk
Random Read	3100	35000	100
Random Write	25	3300	110

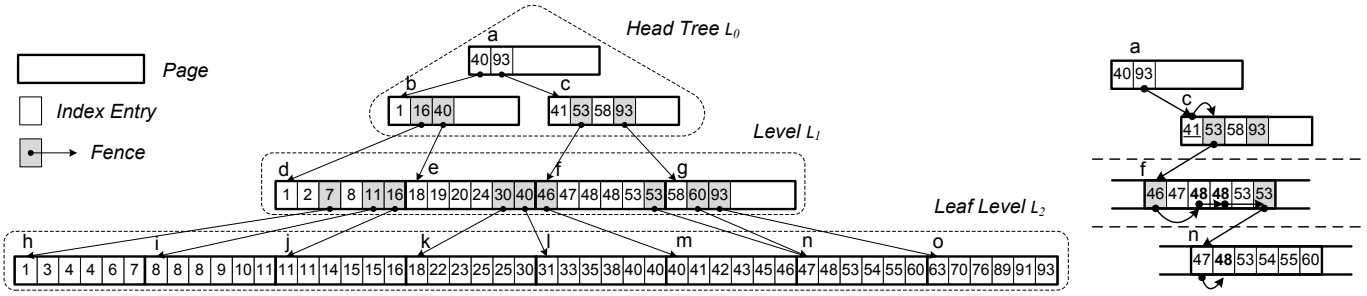
Given the asymmetry of the read and write speeds of the flash disk, write-optimized indexes [7], [8], [9], [10], traditionally optimized for magnetic disks, become a possible alternative for flash-based tree indexing. Especially, the log-structured merge tree (LSM-tree) [7] and its variant [8], proposed for append-only or write-dominant environments, consists of multiple B+-trees and is optimized for the write access patterns: a new entry is firstly inserted into the smallest one and gradually migrated to larger ones. However, a search on these log-structured indexes requires searching multiple B+-tree components. This can degrade the search performance significantly.

To optimize the update performance by reducing small random writes while preserving the search efficiency, we propose the FD-tree, a tree index that is aware of the hardware features of the flash disk. Specifically, we adopt the *logarithmic method* [11] and the *fractional cascading* [12] technique to FD-tree for efficient update and search performance, respectively.

The FD-tree is a logarithmic data structure for reducing the amortized cost of the update. It consists of a small B+-tree, called the head tree, on top of a few levels of sorted runs of increasing sizes. In an FD-tree, updates are only applied to the head tree, and then merged to the lower level sorted runs in batches. Since the head tree is likely to fit into the main memory, most random writes to the flash disk are transformed into sequential ones through the merge. The idea of adopting the logarithmic method is similar to the LSM-tree [7]. The difference is that the FD-tree consists of sorted runs instead of tree components, which allows us to improve the search performance using the fractional cascading.

Fractional cascading was originally proposed to speed up binary searches on multiple sequences of sorted data [12]. We adapt this technique to flash disks to speed up the search on FD-tree. Specifically, we store fences, or pointers to pages in a lower level of sorted run into the immediate higher level. With these fences, a search on an FD-tree is first performed on the small tree, and next on the sorted runs level by level

† Bingsheng He is currently with Microsoft Research at Asia.



(a) The overview of the example FD-tree

(b) Searching $key = 48$

Fig. 1. An example FD-tree

with the fences guiding the position to start in the sorted run of the next level.

In the following, we describe our FD-tree designed and empirically evaluate its efficiency in comparison with three other existing indexes on flash disks.

II. FD-TREE

In this section, we present the design of FD-tree. Our goal is to minimize the number of small random writes, while maintaining a high search efficiency.

A. Index Structure

An FD-tree consists of multiple levels denoted as $L_0 \sim L_{l-1}$. The top level, L_0, is a small B+-tree called the *head tree*. Each of the other levels, $L_i (1 \leq i < l)$, is a sorted run stored in contiguous pages. Figure 1(a) illustrates the structure of an FD-tree. The FD-tree has three levels, the head tree and two sorted runs. The head tree is a two-level B+-tree. With the fractional cascading technique, the leaf nodes of the head tree have pointers to the sorted run, L_1. Each non-leaf level in turn has pointers to the sorted run of the immediate lower level.

Each level of FD-tree has a *capacity*. Following the logarithmic method, we set the levels with a stepped capacity, i.e., $\|L_{i+1}\| = k \cdot \|L_i\|$ $(0 \leq i \leq l-2)$, where k is the logarithmic size ratio between L_{i+1} and L_i. Therefore, $\|L_i\| = k^i \cdot \|L_0\|$. The updates are initially performed on the head tree, and then are gradually migrated to the sorted runs at the lower levels in batches when the capacity of a level is exceeded. The maximum size of the head tree, $\|L_0\|$ is far smaller than the available amount of main memory, so that it is likely to reside in the main memory. Specifically, we set $\|L_0\|$ to the erase block size of flash disk so that the head tree fits into one erase block.

We categorize the entries in an FD-tree into two kinds, index entry and fence.

- *Index Entry*. An index entry contains three fields: an index key, key, a record ID, rid, of the indexed data record, and $type$ indicating its role in the logarithmic deletion of FD-tree.
- *Fence*. A fence is an entry with three fields: a key value, key, a $type$, and a pid, the id of the page in the immediate lower level that a search will go next. Depending on whether the key value of the fence in L_i is selected from L_i or L_{i+1}, we further categorize fences in L_i into two kinds, internal fences and external fences.
  - *External fence* ($type = External$). The key value of an external fence in L_i is selected from L_{i+1}. We create a fence for each page of L_{i+1}. For page P in L_{i+1}, we select the key of the last entry in P to be the key of the fence, and set the pid field of the fence to be the id of P.
  - *Internal fence* ($type = Internal$). The key value of an internal fence is selected from L_i. We add internal fences to handle data skews. If the index entries between two consecutive external fences, F_j and F_{j+1}, span multiple pages (denoted as P_0, P_1, ..., P_p, p is the number of spanned pages), we add an internal fence to page P_i $(0 \leq i < p)$ as the last entry of the page. The key value of the internal fence at P_i is set to be the key of the last index entry in P_i. The pid field of the internal fence is set to be the same as that of F_{j+1}. For example, in Figure 1(a), entry 53 in page `f` is an internal fence that points to page `n`. With this fence, we can avoid fetching page `g` when searching keys between 47 and 53.

In each level of FD-tree, the fences and index entries are organized in the ascending order of their keys. With the same key, the external fences follow the index entries. By design, the number of the external fences in L_i is the number of pages in L_{i+1}, $|L_{i+1}|/f$, where f is the number of entries in a page. The number of the internal fences in L_i is a maximum of $|L_i|/f$, because each page contains at most one internal fence. Then, the total number of fences in L_i, $(|L_i| + |L_{i+1}|)/f$, is less than the number of entries in L_i, i.e., $k < f - 1$.

B. Search

An index search on the FD-tree requires searching each level from top down. A query can be either a point search with an equality predicate (an exact match), or a range search with a range predicate. Since the algorithm for the point search is similar to that for the range search except the difference in evaluating entries with predicates, we focus on the algorithm for the point search.

To search an entry, we first perform a lookup on the head

tree, the same as that on the traditional B+-tree. Next, we perform a search on each level following the pid of the fence. Within a page in L_i, a binary search is performed to find the first matching entry, if any. We then scan the sorted run from the first matching entry to find all matching entries. Next, we continue the scan until we find the fence whose key value is equal to or larger than the search key. All matching entries are added to the result set. Since each page has at least one fence, the scan is performed only on the pages having matches. After finishing at level L_i, following the pid of the fence, we go down to the next level L_{i+1}.

Figure 1(b) illustrates searching 48 on the example FD-tree in Figure 1(a). At each level, it searches a page until it encounters a fence and follows the fence to search the page in the next level of sorted run.

The search performance of the FD-tree is not necessarily worse than the B+-tree. Although an FD-tree may be higher than a B+-tree with the same size because the fanout of FD-tree is less than that of B+-tree with the same page size, i.e., $k < f-1$, two features in the design of the FD-tree inherently benefits the index search performance. First, the pages in the FD-tree are full of entries, and the entries in the levels of the FD-tree except of L_0 are stored contiguously. Note, the nodes in the B+-tree are not full, typically with a utilization of 70% [13]. Second, FD-tree does not have the aging problem [14] like the B+-tree, where the locality of leaf nodes degrades after a large number of updates.

C. Insert and Merge

To insert an entry, the new entry is inserted into the head tree L_0 first. If the number of entries in the head tree L_0 exceeds its capacity $|L_0|$, a merge operation is performed between L_0 and L_1 to migrate all entries in L_0 to L_1.

The merge process is performed on two adjacent levels when the smaller one of the two exceeds its capacity. The merge operation sequentially scans the two inputs in the order of key values, and combines them into one sorted run in contiguous pages. A newly generated L_i consists of all index entries from L_{i-1}, all index entries and external fences from L_i. The new internal fences of L_i are constructed during the merge when necessary. At the same time, the new levels $L_j (0 \leq j < i)$ are rebuilt with the external fences constructed from the newly generated L_i. That is, given two adjacent levels, L_{i-1} and L_i, the merge process generates $i+1$ new sorted runs to update all levels from L_0 to L_i. If new L_i exceeds its capacity, L_i and L_{i+1} are merged. This process continues until the larger one of the two newly generated levels does not exceed the capacity.

D. Delete and Update

Deletion of an entry in the FD-tree is performed by inserting a special entry called a *filter entry*. The existing then becomes a *phantom entry*, and is left untouched. Specifically, we first perform a search on the FD-tree using the predicate of the deletion. This search identifies the set of index entries to be deleted. New entries (filter entries) with the same key and pointer value as these entries are inserted into the FD-tree. The reason for inserting new entries instead of marking existing entries invalid is to avoid the small random write of marking.

Since deletions insert filter entries and make old entries become phantom entries, a subsequent search may get a result set containing both types of entries. Therefore, we need to remove filter entries and phantom entries of the same key and pointer value from the result set in a search. As the merge process occurs, both filter entries and phantom entries are migrated to the lower levels. When they encounter each other at the same level, they will be skipped and not appear in the newly merged run. Thus, the phantom and their filter entries are eventually deleted.

Figure 2 illustrates an example of the deletion process. We mark the filter entries with a solid underline, and their phantom entries with a dashed underline. In Figure 2(a), we delete the index entries 37 in L_0, 45 in L_2 and the second 16 in L_2. Here, we use the key to represent the index entry. Since entry 37 is in the head tree L_0, it is deleted from L_0 directly. The filter entries 45 and 16 are inserted into the head tree. When a merge is performed on L_0 and L_1 as shown in Figure 2(b), the filter entry 45 encounters its phantom entry, and both entries are discarded. When a merge is performed on L_1 and L_2, as shown in Figure 2(c), the filter entry 16 and its corresponding phantom entry are discarded. Note, the first index entry 16 remains in the index after the merge.

In the FD-tree, an update operation is implemented as a deletion on the old value and a following insertion with the new value.

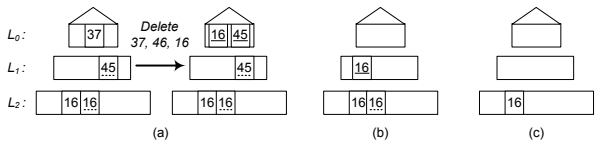


Fig. 2. An example of the logarithmic deletion process

E. Discussion

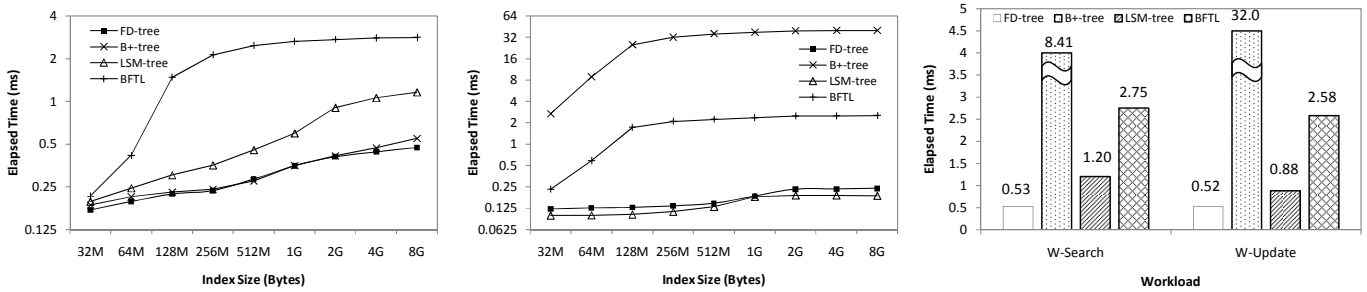
We compare the costs of FD-tree with the representative B+-tree variants including the standard B+-tree, the LSM-tree [7], and BFTL [15], a B+-tree variant solely designed for the flash memory used in embedded systems. The search cost of the FD-tree is close to that of the B+-tree, which is known to have the optimal search cost among all secondary storage index structures. At the same time, the FD-tree supports updates as efficiently as the LSM-tree. In some sense, the FD-tree captures the best of both worlds.

III. Experimental Results

In this section, we empirically evaluate FD-tree in comparison with the standard B+-tree, LSM-tree [7] and BFTL [15].

We ran our experiments on a PC powered by Intel QuadCore CPU 2.4GHz on Windows XP with 2GB main memory, and a 32GB Samsung NAND flash disk.

The entries in the indexes contain a 4-byte integer key and another 4-byte field shared by *type* and *pointer*. The



(a) Search performance vs. index size (b) Insertion performance vs. index size (c) Overall performance of two workloads

Fig. 3. Comparison of the four indexes

key values are uniformly distributed. The page size is set to 2KB, which is the physical page size of the flash disk. In our experiments, we set $f = k/2$ for simplicity. Further tuning on k could be done to achieve the optimal overall performance by balancing read/write performance. An LRU buffer manager is implemented for caching recently accessed disk pages. The size of buffer pool is set to 8MB. We disabled the buffering functionality of the operating system to avoid the interference.

Figure 3(a) shows the search performance of the four indexes with the index size varied. Among the four indexes, BFTL is the slowest, because it requires fetching multiple pages randomly in accessing a tree node. B+-tree and FD-tree are the best, and they perform quite similarly regardless of the index size. FD-tree is slightly faster than B+-tree, because the pages in the sorted runs are entirely full and are stored consecutively. LSM-tree is slower than B+-tree and FD-tree, because a single search on LSM-tree requires searching on multiple B+-tree components.

Figure 3(b) demonstrates the insertion performance of the four indexes with the index size varied. Among the four indexes, the B+-tree has the worst insertion performance, and LSM-tree is the best. The insertion performance of FD-tree is slightly slower than LSM-tree. This is because FD-tree has more auxiliary entries such as the fences than LSM-tree. Nevertheless, both LSM-tree and FD-tree are orders of magnitude faster than the B+-tree and BFTL, due to their logarithmic structure design. Since BFTL delays and clusters the updates on the same page, it outperforms the B+-tree.

Figure 3(c) shows the overall performance of the four indexes for the W-Search and W-Update workloads on the flash disk. All these four indexes contain 10^9 entries. Their sizes are approximately 8GB. The y-axis is the average elapsed time per request of the workload. We define W-Search as a workload consisting of 80% searches, 10% insertions, 5% deletions and 5% updates to simulate a workload dominated by writes. We use a workload of 20% searches, 40% insertions, 20% deletions and 20% updates to simulate a workload dominated by reads, denoted as W-Update. These two workloads are representatives of the read/write-intensity of commercial workloads. For W-Search on the flash disk, FD-tree is 15.8X, 2.3X, 5.2X faster than B+-tree, LSM-tree and BFTL, respectively. For W-Update on the flash disk, FD-tree is 61.4X, 1.7X, 4.9X faster than B+-tree, LSM-tree and BFTL, respectively.

IV. CONCLUSIONS

In this paper, we identify that the B+-tree indexes designed for the hard disk are unsuitable for the flash disk, and propose a flash disk aware tree index, FD-tree. We design our tree index with the logarithmic and the fractional cascading techniques to improve its overall performance. Our tree index takes advantage of hardware features of the flash disk by utilizing efficient random reads and sequential accesses, and eliminating the slow random writes. Both of our analytical and empirical results show that FD-tree captures the best of both search and insertion performance among existing tree indexes, and outperforms these indexes for both search- and update-intensive workloads.

REFERENCES

[1] K. Kimura and T. Kobayashi, "Trends in high-density flash memory technologies," in *IEEE Conference on Electron Devices and Solid-State Circuits*, 2003.
[2] J. Gray and B. Fitzgerald, "Flash disk opportunity for server applications," *ACM Queue*, vol. 6, no. 4, pp. 18–23, 2008.
[3] S.-W. Lee and B. Moon, "Design of flash-based dbms: an in-page logging approach," in *SIGMOD Conference*, 2007, pp. 55–66.
[4] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *SIGMOD Conference*, 2008, pp. 1075–1086.
[5] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.
[6] *Intel X25-E SATA Solid State Drive Datasheet*. Intel Corp., 2008.
[7] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.
[8] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *VLDB*, 1997, pp. 16–25.
[9] C. Jermaine, A. Datta, and E. Omiecinski, "A novel index supporting high volume data warehouse insertion," in *VLDB*, 1999, pp. 235–246.
[10] G. Graefe, "Write-optimized b-trees," in *VLDB*, 2004, pp. 672–683.
[11] J. L. Bentley, "Decomposable searching problems," *Inf. Process. Lett.*, vol. 8, no. 5, pp. 244–251, 1979.
[12] B. Chazelle and L. J. Guibas, "Fractional cascading: I. a data structuring technique," *Algorithmica*, vol. 1, no. 2, pp. 133–162, 1986.
[13] D. Comer, "The ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.
[14] N. Ponnekanti and H. Kodavalla, "Online index rebuild," in *SIGMOD Conference*, 2000, pp. 529–538.
[15] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An efficient b-tree layer implementation for flash-memory storage systems," in *RTCSA*, 2003, pp. 409–430.