Tree Indexing on Solid State Drives

Yinan Li*, Bingsheng He, Robin Jun Yang, Qiong Luo, Ke Yi

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

ABSTRACT

Large flash disks, or solid state drives (SSDs), 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 small random writes, on flash disks are inherently much slower than reads because of the erase-before-write mechanism.

To address this asymmetry of read-write speeds in tree indexing on the flash disk, we propose 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 a small area – 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. Given an FD-tree of n entries, we analytically show that it performs an update in $O(\log_B n)$ sequential I/Os and completes a search in $O(\log_B n)$ random I/Os, where B is the flash page size. We evaluate FD-tree in comparison with representative B+-tree variants under a variety of workloads on three commodity flash SSDs. Our results show that 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 dominates the other B+-tree index variants on the overall performance on flash disks as well as on magnetic disks.

1. INTRODUCTION

Solid State Drives (SSDs), or flash disks, have emerged as a viable alternative to the magnetic disk for non-volatile storage. The advantages of flash SSDs include high random read performance, low power consumption and excellent shock resistance. With the capacity doubles every year [16], flash SSDs have been considered device to replace magnetic disks for enterprise database servers [12, 17, 18, 26]. 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.

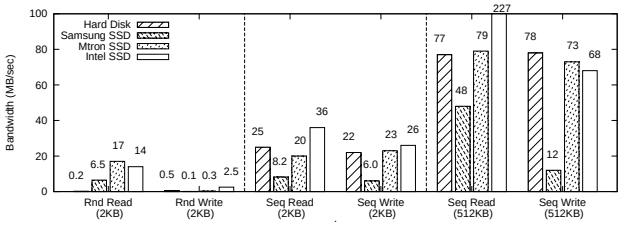


Figure 1: Bandwidths with 2KB or 512KB pages *Random Read*, *Random Write*, *Sequential Read*, and *Sequential Write*.

The flash SSD is a type of electrically-erasable programmable read-only memory (EEPROM). Unlike magnetic disks where seek and rotational delays are the dominant cost in reading or writing a page, the flash SSD has no mechanic movement overhead. As a result, random reads of a flash SSD are up to two orders of magnitude faster than a magnetic disk, as shown in Figure 1. However, due to the erase-before-write mechanism of the flash memory, each write operation may require erasing a large block, called an *erase block*. This mechanism makes random writes one to two orders of magnitude slower than random reads. As shown in Figure 1, random writes are 163.3X, 62.7X and 5.6X slower than random reads on Samsung SSD, Mtron SSD and Intel SSD, respectively. Additionally, previous results [6] showed that random write bandwidths further reduced 3.5-10X on fragmented flash SSDs, while the random read bandwidths were affected little.

Given the asymmetry in the read and write speeds of the flash SSD, B+-tree, the most popular tree index structure for the hard disk will benefit from the fast random read speed in search performance, but will suffer from the poor random write speed in update performance. In comparison, write-optimized indexes [24, 14, 15, 10], originally designed for disks, will mitigate the weakness of updates on flash SSDs. However, all these indexes are suboptimal on search performance. Recently, there has been initial work, in particular BFTL [28], on optimizing the B+-tree for flash memory in embedded systems. Unfortunately, BFTL improves the update performance at the expense of deteriorated search performance.

To optimize the update performance while preserving the search efficiency, we propose FD-tree [19], a tree index that is aware of the hardware features of the flash SSD. Specifically, we adopt the *logarithmic method* [3] and the *fractional cascading* [5] technique to FD-tree for efficient update and search performance, respectively.

We design FD-tree to be a logarithmic data structure to reduce the amortized update cost. It consists of a small B+-tree, called the head tree, on top of a few levels of sorted runs of increasing sizes. We determine the size ratio between adjacent sorted runs consid-

*Yinan Li is currently with University of Wisconsin-Madison.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

ering the read and write speeds of the flash disk in addition to the workload composition. In an FD-tree, updates are only applied to the head tree, and then merged to the lower level sorted runs in batches. As a result, most random writes are transformed into sequential ones through the merge. Since the sequential accesses on flash SSDs exhibit much higher bandwidths than the random ones, as shown in Figure 1, the update performance of FD-tree is improved significantly. Our idea of adopting the logarithmic method is similar to LSM-tree [24]. The difference is that an FD-tree consists of sorted runs instead of tree components, which allows us to improve the search performance using fractional cascading. Moreover, we propose a deamortized scheme to reduce the worst case cost of insertions on FD-tree while preserving the average cost.

Fractional cascading is a general technique to speed up binary searches in a sequence of data structures [5]. We adapt this technique to FD-tree to speed up the search efficiency. 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 with the fences guiding the position to start in the sorted run of the next level.

We analytically estimate the search and update costs of FD-tree. Our cost estimation considers the asymmetry of read and write speeds of the flash SSD, as well as the different patterns of sequential and random accesses. Subsequently, we analytically compare the costs of FD-tree with the representative B+-tree variants including the standard B+-tree [8], the LSM-tree [24], and BFTL [28]. Given n index entries, the search cost of FD-tree is close to that of B+-tree, and matches the optimal search cost $O(log_B n)$ I/Os, where B is the page size. In the meanwhile, FD-tree supports an update in $O(log_B n)$ sequential page writes, as efficiently as the LSM-tree. In short, FD-tree captures the best of both worlds. Additionally, considering the significant differences in the performance of various flash SSDs, we develop a cost model to determine the optimal settings on the sizes of the sorted runs in the FD-tree for individual flash SSDs, given the characteristics of the workload.

We empirically evaluate the FD-tree in comparison with the three B+-tree variants. Our result on all three commodity SSDs shows that the FD-tree captures the best of both search and insertion performance among all competitors. In particular, it is 5.7-27.9X, 1.4-1.6X and 3.7-5.5X faster than B+-tree, LSM-tree and BFTL, respectively, under various mixed workloads on an Mtron SSD, and it is 1.7-3.6X, 1.4-1.8X, and 1.9-3.4X faster than B+-tree, LSM-tree and BFTL, respectively, on an Intel SSD. Additionally, on the hard disk, FD-tree achieves a similar search performance to B+-tree under read-intensive workloads and outperforms all others under update-intensive workloads.

The paper is organized as follows. In Section 2, we review the I/O optimization techniques for the hard disk and the flash disk. We present the design of FD-tree and its cost analysis in Section 3 and 4, respectively. In Section 5, we experimentally evaluate the efficiency of FD-tree. Finally, we conclude in Section 6.

2. PRELIMINARY AND RELATED WORK

This section reviews the related work on the techniques optimizing the random writes on the flash SSD and on the hard disk. For more details on flash SSDs, we refer the readers to Appendix A.

2.1 Optimizing random writes on SSDs

Flash-specific file systems [27, 20] have been proposed based on the log file system [25]. With a mapping between logical and physical page identifiers dynamically maintained, every updated page is sequentially appended and its mapping table entry is correspondingly updated. However, both random and sequential read performance of log file systems significantly suffers from the overhead of looking up and maintaining the mapping table [22]. Moreover, the log file system is likely to quickly consume pages, which in turn requires frequent garbage collection to reclaim obsolete pages [9].

Database researchers attempt to address the random write issues by designing specific data structures and algorithms. Lee et al. [17] proposed the In-Page Logging (IPL) to improve the update performance in a DBMS. Different from the log file system, IPL appends the update logs into a special page that is placed in the same erase block as the updated data page in order to improve the search efficiency of log-structure method. However, it is hard to make flash SSDs support the fine granularity write, e.g. a few bytes, on an erased page. The performance of key components in DBMS was evaluated on the flash SSDs [18]. Tsirogiannis et al. [26] demonstrated the column-based layout within a page can leverage fast random reads of flash SSDs to speed up different query operators. Chen exploited flash devices for logging based on the observation that flash devices are suitable for small sequential writes [7].

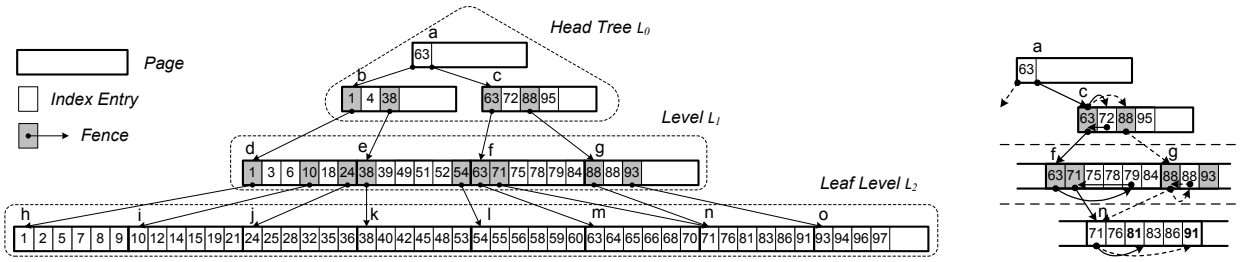
2.2 Write Optimized Tree Indexing

Due to the poor random write performance of flash SSDs, write optimized tree indexes [28, 23] have been proposed to improve the update performance. BFTL [28] was proposed to balance the inferior random write performance and fast random read performance for flash memory based sensor nodes and embedded systems. It allows the index entries in one logical B-tree node to span over multiple physical pages, and maintains an in-memory table to map each B-tree node to multiple physical pages. Newly inserted entries are packed and then written together to some new blocks. The table entries of corresponding B-tree nodes are updated, thus reducing the number of random writes. However, BFTL entails a high search cost since it accesses multiple disk pages to search a single tree node. Furthermore, even though the in-memory mapping table is compact, the memory consumption is still high. FlashDB [23] was proposed to implement a self-tuning scheme between standard B+-tree and BFTL, depending on the workloads and the types of flash devices. Since our proposed index mostly outperforms both B+-tree and BFTL under various workloads on different flash SSDs, we do not compare our index with this self-tuning index in this paper. More recently, LA-tree [1] was proposed for flash memory devices by adding adaptive buffers between tree nodes. LA-tree focuses on raw, small-capacity and byte addressable flash memory devices, such as sensor nodes, whereas our work is targeted for off-the-shelf large flash SSDs, which provide only a block-based access interface. Different target devices of these two indexes result in their differences in design.

On the hard disk, many disk-based indexes optimized for write operations have also been proposed. Graefe proposed a write-optimized B-tree [10] by applying the idea of the log file system [25] to the B-tree index. Y-tree [15] supports high volume insertions for data warehouses following the idea of buffer tree [2]. The logarithmic structures have been widely applied to optimize the write performance. O'Neil et al. proposed LSM-tree [24] and its variant LHAM [21] for multi-version databases. Jagadish et al. [14] used a similar idea to design a stepped tree index and the hash index for data warehouses. Our FD-tree follows the idea of logarithmic method. The major difference is that we propose a novel method based on the fractional cascading technique to improve the search performance on the logarithmic structure.

3. FD-TREE

In this section, we present the design of FD-tree. Our goal is



(a) The overview of the example FD-tree

(b) Searching $key = 81, 91$

Figure 2: An example FD-tree

to minimize the number of small random writes and to limit these random writes within a small area, while maintaining a high search efficiency. For simplicity, we assume all index keys in an FD-tree are unique. The notations used throughout this paper are summarized in Table 1.

Table 1: Parameters used in this paper

Parameters	Description
B	page size (bytes)
L_i	i^{th} level of FD-tree
$\|L_i\|$	capacity of L_i (the number of entries)
l	number of levels in FD-tree
k	logarithmic size ratio between adjacent levels
n	number of records in the indexed relation
f	number of entries in a page

3.1 Design principles for indexing on SSDs

In our design for an index on flash SSDs, we consider the following three principles.

- $\mathcal{P}1$. *Transforming random writes into sequential ones.* We should take advantage of sequential writes, and avoid the random writes by designing sophisticated data structures.
- $\mathcal{P}2$. *Limiting random writes within a small region.* Previous studies [4, 6] reported that random writes on flash SSDs with a small area (512KB-8MB) have a comparable performance to sequential writes.
- $\mathcal{P}3$. *Supporting multi-page I/O optimization.* Accessing multiple pages in an I/O operation is more efficient than accessing each page separately.

3.2 Overview of FD-Tree

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*. The node size of the head tree is the page size B of the flash SSD. Each of the other levels, $L_i (1 \leq i < l)$, is a sorted run stored in contiguous pages. Figure 2(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 contain pointers to the sorted run L_1. Each non-leaf level in FD-tree contains pointers to the sorted run of the immediate lower level.

Each level of FD-tree has a *capacity* in terms of entries, denoted as $|L_i|$. 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 adjacent levels. 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. Following the design principle $\mathcal{P}2$, the maximum size of the head tree is set to the size of the *locality area*, within which random writes have similar performance as sequential ones. The size of locality area measured on nowaday devices is typically between 128KB and 8MB [4, 6].

We categorize the entries in FD-tree into two kinds, index entry and fence. In each level of FD-tree, the index entries and fences are organized in the ascending order of their keys.

- *Index Entry*. An index entry contains three fields: an index key, key, and a record ID, rid, for the indexed data record, and $type$ indicating its role in the logarithmic deletion of FD-tree. Depending on the $type$, we further categorize index entries into two kinds, filter entries and normal entries.
  - *Filter Entry* ($type = Filter$). A filter entry is a mark of deletion. The filter entry is inserted into FD-tree upon a deletion to indicate that its corresponding record and index entry have been nominally deleted. It has the same key and record ID as that deleted index entry. We call that deleted index entry as a *phantom entry*, as it has been logically deleted but has not been physically removed from the index.
  - *Normal Entry* ($type = Normal$). All index entries other than filter entries are called normal entries.
- *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. Essentially, a fence is a pointer, whose key is selected from a index entry in FD-tree.

INVARIANT 1. *The first entry of each page is a fence.*

INVARIANT 2. *The key range between a fence and its next fence at the same level is contained in the key range of the page pointed by the fence.*

Depending on whether the key value of the fence in L_i is selected from L_i or L_{i+1}, we 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 first entry in P to be the key of the fence, and set the pid field of the fence to be the id of P, in order to satisfy Invariant 2.
- *Internal fence* ($type = Internal$). The key value of an internal fence in L_i is selected from L_i. If the first entry of any page P is not a external fence, we add an internal fence to the first slot of this page in order to satisfy Invariant 1. The key value of the internal fence is set to be the key of the first index entry e in page P. The pid field of the internal fence is set to the id of the page in the next level whose key range covers the key of e. For example, in Figure 2(a), entry 88 in page g is an internal fence that points to page n, the same as the external fence 71 in page f.

According to the definition of the external fence, the number of external fences in L_i is the number of pages in L_{i+1}, i.e. $|L_{i+1}|/f$, where f is the number of entries in a page. The number of internal fences in L_i is at most $|L_i|/f$, because each page contains at most one internal fence. The maximum total number of fences in L_i, $(|L_i| + |L_{i+1}|)/f$, should be smaller than the number of entries in L_i, obtaining $k < f - 1$.

3.3 Operations on FD-Tree

FD-tree supports five common query operations: search, insertion, merge, deletion and update. For the algorithm pseudocode, please see Appendix 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, or a range search with a range predicate.

To perform a point search on a search key K, we first perform a lookup on the head tree, the same as that on the standard B+-tree. Next, we perform a search on the each level following the pid of the fence. Within a page P in L_i, a binary search is performed to find the greatest key equal to or less than K. Suppose the entry e_i contains this key. We then scan the sorted run from right to left until we find a fence e_j. Since all entries with a key between $e_i.key$ and $e_j.key$ in the next level L_{i+1} appear in the page $e_j.pid$ (Invariant 2), we then follow the pointer $e_j.pid$ to this page to search for entries in the next level. This way, the tree is traversed top to bottom, following the pid of desired fences.

Since a filter entry is inserted into the FD-tree upon a deletion and makes the old entry become a phantom entry, a search may get a result set containing both the filter entry and its corresponding phantom entry. If so, we need to remove filter entries and phantom entries of the same key and pointer value pair from the result set in a search.

According to Invariant 1, a search can at least find the fence in the first slot of any page when scanning the page backward. Thus, a search only fetches one page each level, if there are no duplicates. This is the reason why we introduce internal fences. If data is skewed, the index entries between two consecutive external entries, F_j and F_{j+1}, may span multiple pages. A scan starting between F_j and F_{j+1} need to go over multiple pages to get the previous external fence F_j. With the internal fences, the scan is stopped by the internal fences at the first slot of each page.

Figure 2(b) illustrates the search paths of key 81 (in solid line) and key 91 (in dotted line) on the example FD-tree in Figure 2(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. In the search in L_1 for 91, the internal fence 88 in page g prevents the scan from fetching page f to find the external fence 71.

The range search is similar to that for the point search except that it may fetch multiple pages in each level. Given the fences satisfying the predicate in the current level L_i, we are aware of the number of pages that will be scanned in the next level L_{i+1} before fetching those pages. Moreover, those pages are stored contiguously. These properties provide an opportunity to fetch the exact number of matched pages in the next level in a I/O operation by using multi-page I/O optimization ($\mathcal{P}3$).

Insertion. A new entry is initially 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 on L_0 and L_1 to migrate all entries in L_0 to L_1. As a result, the random writes are limited within the head tree following design principle $\mathcal{P}2$.

Merge. 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, and combines them into one sorted run in contiguous pages. A newly generated level L_i consists of all index entries from L_{i-1}, all index entries and external fences from L_i. We keep all external fences in L_i because the level (L_{i+1}) pointed by these external fences does not change. 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 the 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.

The merge operation involves only sequential reads and writes, thus we successfully transform the random writes of insertion into sequential reads and writes, following the design principle $\mathcal{P}1$. We further optimize the I/O performance by applying the multi-page I/O optimization, following our design principle $\mathcal{P}3$. Since the pages in each level of FD-tree are stored contiguously on the flash disk, we fetch multiple pages in a single I/O request. Similarly, as the newly generated sorted runs are sequentially written, we write multiple pages in a single request. The suitable number of pages in an I/O request is set to be the access unit size when the transfer rate of the sequential access pattern reaches the maximum.

Deletion. A deletion on the FD-tree is handled in a way similar to an insertion: it is first performed on the head tree, and then migrated to the lower levels as the merge process occurs. This logarithmic deletion scheme reduces the amortized cost. Note, the lazy deletion method widely used in hard disk based indexes, which marks an entry invalid, is inefficient on the flash SSD, because a marking operation is a small random write.

The first step is to perform the deletion on the head tree L_0, because random writes on the head tree are limited within a locality area, and are very efficient. Next, we perform deletion in the other levels by inserting a special entry called a *filter entry*. The entry to be deleted 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 index entry to be deleted. New entries (filter entries) with the same key and pointer value as these entries are inserted into the FD-tree. The actual deletion is performed in the merge operation when a filter entry encounters its corresponding phantom entry.

During the merge, physical deletions are performed in batches. When a filter entry encounters its corresponding phantom entry, both entries are discarded, and will not appear in the merge result. Thus, a deletion is physically completed. Note, due to the processing on filter entries and their phantom entries, a newly generated sorted run may be smaller than the old one.

The space overhead of filter and phantom entries is low. Since the lowest level L_{l-1} does not contain any filter and phantom entry, these entries at worst occupy all levels except of the lowest one, whose total size is only around $1/k$ space of the whole index. Since k is typically large, filer and phantom entries have low impact on performance.

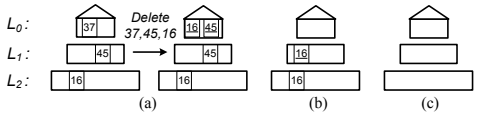


Figure 3: An example of the logarithmic deletion process

Figure 3 illustrates an example of the deletion process. We mark the filter entries with a solid underline. In Figure 3(a), we delete

Table 2: The I/O cost comparison of four tree indexes

Index Name	Search Cost	Insertion Cost			
	Random Read	Random Read	Sequential Read	Random Write	Sequential Write
FD-tree	$O(\log_k n)$		$O(\frac{k}{f-k} \log_k n)$		$O(\frac{k}{f-k} \log_k n)$
B+-tree	$O(\log_f n)$	$O(\log_f n)$		$O(1)$	
LSM-tree	$O(\log_k n \cdot \log_f n)$		$O(\frac{k}{f} \log_k n)$		$O(\frac{k}{f} \log_k n)$
BFTL	$O(c \log_f n)$	$O(c \log_f n)$		$O(1/c)$	

the index entries 37 in L_0, 45 in L_2 and 16 in L_2. 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. As other insertions and deletions performing, the head tree is gradually becoming larger. When it is full and a merge is performed on L_0 and L_1 as shown in Figure 3(b), the filter entry 45 encounters its phantom entry, and both entries are discarded. When more entries are inserted into the index and a merge between L_1 and L_2 occurs, as shown in Figure 3(c), the filter entry 16 and its corresponding phantom entry are discarded.

Update. An update operation is implemented as a deletion on the old value followed by an insertion.

3.4 Deamortized Operations on FD-Tree

While the logarithmic method reduces the amortized cost of insertions on FD-tree, the worst case cost is still high. In the worst case, all sorted runs exceed their capacities after a single insertion and the whole FD-tree has to be entirely rewritten. This process may result in an unacceptable response time. Thus, we propose a simple and effective scheme to address this problem. We take the deamortization for insertions as an example, since deletions and updates are handled in a similar way.

Figure 4 demonstrates the basic idea of deamortized insertions on FD-tree, which is to overlap the execution of insertions and the merge operation. Specifically, given N_{merge} entries to be merged, we divide these N_{merge} entries into $|L_0|$ partitions, and progressively combine entries in a partition after executing an insertion. As a result, the expensive cost of the merge operation is amortized to $|L_0|$ insertion operations. Thus, the worst elapsed time of insertions is reduced by around a factor of $|L_0|$, with the average cost unchanged.

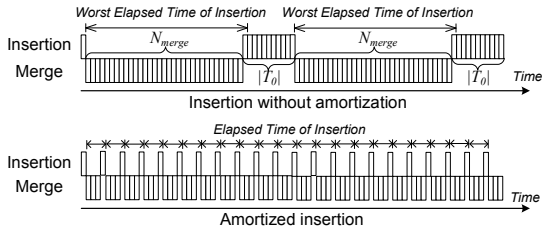


Figure 4: Insertion w/ and w/o deamortization

In order to overlap the execution of insertions and merge, we maintain two head trees. Once a head tree L_0 is full, new entries are inserted into the other one, i.e. the temporary head tree L'_0, while the merge is performed on L_0. The merge process is similar to that we described in Section 3.3, except that external fences from the lower levels are inserted into L'_0 one by one, rather than bulk-loading. Once the merge is complete, L'_0 has already been filled, and all external fences have been inserted into L'_0. With deamortization, when a merge completes, we swap L_0 and L'_0 for subsequent insertions and merges.

With deamortization, index search requests can proceed even when a merge is on-going. Since the original FD-tree ($L_0 \sim L_{l-1}$) contains all original entries inserted before the merge operation, and the temporary head tree L'_0 stores all newly inserted entries, we can perform lookups on both of the original FD-tree ($L_0 \sim L_{l-1}$), and the temporary head tree L'_0. The size of temporary head tree (T'_0) is so small that it is very likely to fit into memory, and the performance overhead of deamortized searches is insignificant.

4. COST ANALYSIS AND COST MODEL

Cost Analysis. We present the major results in Table 2, and leave the details on deriving the results in Appendix C.1 and C.2.

THEOREM 1. *Given an FD-tree with l levels, the amortized cost of insertion is minimized when all size ratios between adjacent levels are equal.*

Theorem 1 justifies our setting on the equal size ratios in our FD-tree design. The proof is omitted here, and the reader is referred to Appendix C.2. Given an index of n entries, our analysis shows that the search cost of the FD-tree matches the optimal search cost $O(log_B n)$ I/Os, where B is the page size. In the meanwhile, FD-tree supports an update in $O(log_B n)$ sequential page writes.

We compare the I/O cost of FD-tree with other B+-tree variants including the standard B+-tree [8], LSM-tree [24] and BFTL [28]. Table 2 shows their costs on the search and insertion. FD-tree has a complexity comparable to B+-tree on search and similar to LSM-tree on insertion. As a result, FD-tree captures the best of both worlds. Compared to B+-tree, BFTL increases search cost by c times while reducing insertion cost by c times ($c \geq 1$, a tuning parameter in BFTL [28]) to balance the asymmetry of read and write speeds.

Cost Model for Parameter Setting. We present an analytical model to determine the optimal k value for the overall performance. We focus on the following three aspects in our cost model. More details on our cost model can be found in Appendix C.3.

Firstly, the complexity results in Table 2 need to be refined for an accurate estimation.

Secondly, the search cost analysis does not take the buffer pool into consideration. To set the k value accurately, we develop a cost model with the buffer pool considered, in particular, its effect on different levels of the tree.

Finally, our cost model is able to estimate the execution time for a given workload. Deletion and update operations are implemented using search and insertions. For example, a deletion is implemented by a search followed by an insertion. Thus, a workload can be treated as a mix of searches and insertions.

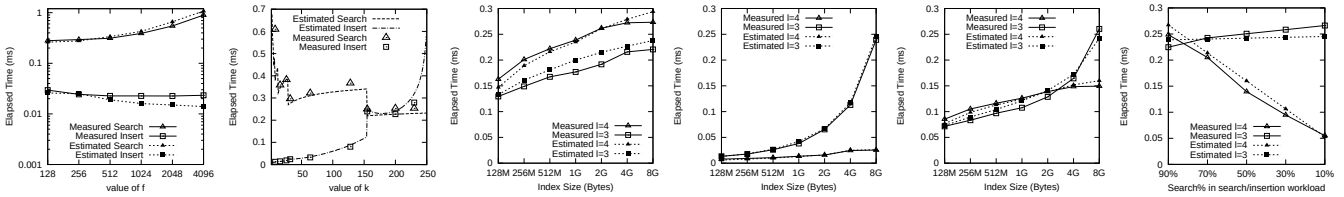
We enumerate the candidate k values, calculate the estimated time cost for each value, and determine the suitable k value that minimizes the estimated time cost of the given workload.

5. EXPERIMENTAL RESULTS

In this section, we empirically evaluate the cost model, and the efficiency of FD-tree in comparison with representative indexes.

5.1 Experimental Setup

We ran our experiments on a workstation powered by Intel 2.4GHz quad-core CPU on Windows XP with 2GB main memory, a 160GB



(a) Search and insertion performance with f value varied

(b) Search and insertion performance with k value varied

(c) Search performance with index size varied

(d) Insertion performance with index size varied

(e) Overall performance with index size varied

(f) Overall performance with search/insertion ratios varied

Figure 5: Comparison between measured and estimated performance of FD-tree on Mtron SSD

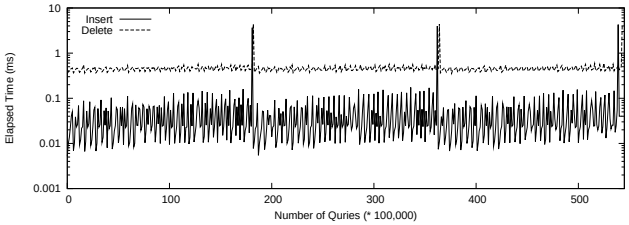
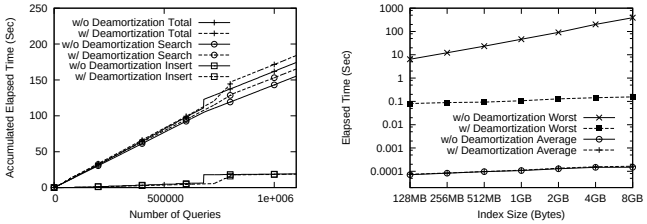


Figure 6: Performance of insertions and deletions



(a) accumulated elapsed time

(b) Average and worst time

Figure 7: Normal Query vs. Deamortized Query

7200rpm SATA magnetic hard disk and three SSDs. We selected three commodity SSDs: Samsung 32GB, Mtron MSD-SATA3035 64GB, and an Intel X25-M 80GB. The detailed features of the three SSDs are summarized in Table 3 in Appendix D. Since some experimental results on Samsung SSD have been reported in previous work [19], we focus on the results on Mtron and Intel SSDs due to the space limitation.

We now briefly describe the implementation and the workload used in the experiment. More details can be found in Appendix D. We have implemented FD-tree in comparison with other fine tuned indexes, including B+-tree [8], LSM-tree [24] and BFTL [28]. We have implemented a storage manager with standard OS file system facilities, with an LRU buffer manager for caching recently accessed disk pages.

We have used our synthetic data sets and workload for a better control on their characteristics. The index entry contains a 4-byte unique key, 30 bits for rid or pid and 2 bits for $type$. Thus, the number of entries in a page, f, is around 250, given a 2KB page size. The key values are uniformly distributed within the range $[0, 2^{30} - 1]$. We have also evaluated the performance for skewed distributions, and the experimental result is similar to that of the uniform data because all the indexes we evaluated are balanced.

The workloads include search only or update only, as well as the mixed ones with different operations. In particular, we have used search-, insertion-, and deletion-intensive workloads, namely W-Search, W-Insert and W-Delete, respectively.

We have evaluated the tree indexing with different characteristics. By default, the index contains one billion entries, whose total size is around 8GB. The size of buffer pool is set to 16MB, which is approximately 0.2% of the 8GB index size. Before running each experiment, we performed sufficient search queries to warm up the buffer pool. All indexes are tuned according to the buffer size, and only the best results are reported.

5.2 Model Evaluation

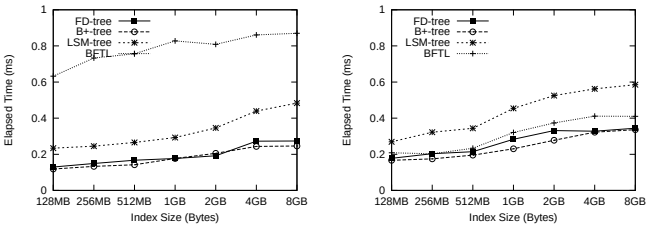
We evaluate the accuracy of our cost model by comparing the estimated and measured performance with various parameters. The estimated and measured performance for Mtron SSD is shown in Figures 5. The results for the Intel SSD are omitted, since we observed a similar trend in the comparison between measurement and estimation. In various settings on parameters, index sizes and workloads, our estimations are close to the measurements (mostly within a 10% difference).

We first analyze the impact of parameters on FD-Tree including f, k, and l. Our measurements are shown in the solid lines of Figure 5. Figure 5(a) shows the search and the insertion performance of an 8GB FD-Tree when f varies from 128 to 4096. The index page size varies from 1KB to 32KB. As the f value increases, the search performance significantly degrades, while the insertion performance slightly improves. We choose the page size of 2KB ($f = 256$), following the observation of the previous study [11].

Figure 5(b) plots the search and insertion performance varying the value of k. The search time decreases with the increased k value. The sharp increase in the search performance is due to the high increase of the FD-tree. On the other hand, the insertion performance degrades as the k increases. Specifically, when k is small, the insertion time increases slightly. Once k is close to f, the insertion performance degrades sharply.

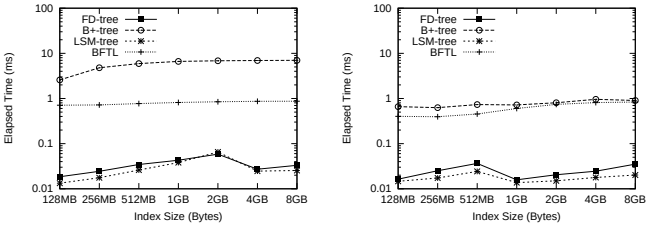
We further study the performance of FD-trees with different numbers of levels, as the index size increases. To satisfy the constraint of $k < f - 1$, an 8GB FD-tree contains at least three levels in the experiment. As shown in Figure 5(b), the performance of FD-tree with five or more levels is dominated by the low search efficiency. Therefore, we only plot the results of FD-trees with three and four levels. The search on FD-tree with three level always outperforms that with four levels, because it accesses fewer pages. In comparison, an FD-tree with fewer levels has a worse performance on insertion than on taller FD-tree. The average insertion performance of the 4-level FD-tree is 1.5–10X higher than the 3-level FD-tree when the size is varied from 128MB to 8GB.

Figure 5(e) illustrates the overall performance of FD-trees under a 50% search 50% insertion workload with various index sizes. When the FD-tree is small, k remains small. The 3-level FD-Tree exhibits good performance for both search and insertion. As the index size increases, the value of k increases as well. Once the value of k is close to f, the insertion performance degrades sharply and dominates the overall performance. In such cases, a 4-level FD-tree with a smaller k value exhibits a more balanced performance between search and insertion. Figure 5(f) plots the elapsed time of an 8GB FD-tree with various search/insertion ratios. A 3-level FD-tree outperforms a 4-level FD-tree, when 90% of the workload are searches.



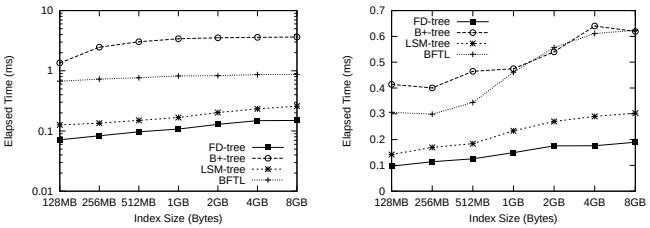
(a) On Mtron SSD (b) On Intel SSD

Figure 8: Search performance comparison varying index size



(a) On Mtron SSD (b) On Intel SSD

Figure 9: Insertion performance comparison varying index size



(a) On Mtron SSD (b) On Intel SSD

Figure 10: Overall performance comparison varying index size

5.3 Insertion and deletion performance

Since insertions and deletions of FD-tree are based on logarithmic methods, we study their performance trends in long-running experiments. We insert (or delete) 50 million entries into an 8GB FD-tree. The total number of inserted (deleted) entries is sufficient to make the lowest two levels be merged for three times. The average elapsed time of each 100 thousand operations is shown in Figure 6.

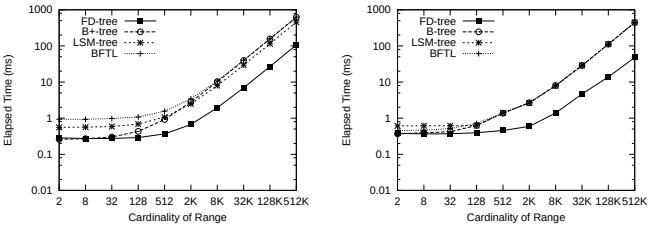
The average elapsed time of insertions varies significantly, from 0.01 to 5 milliseconds, due to the sizes of levels where merge operations occur. The three spikes in the figure indicate the three merges between the lowest two levels. Between every two spikes, the average elapsed time fluctuates. As the levels gradually become larger along with insertions, the average insertion time increases. Once all insertions migrate to the lowest level, the elapsed time reduces to a small value.

The average elapsed time of deletions is greater than those of insertions, because a search is invoked before inserting the filter entries. The elapsed time remains relatively steady except the three spikes. We also find that the three spikes of deletions appear slightly later than that of insertions. The reason is that some filter entries encounter their corresponding phantom entries and are absorbed before migrating to the lowest level.

5.4 Performance of deamortization

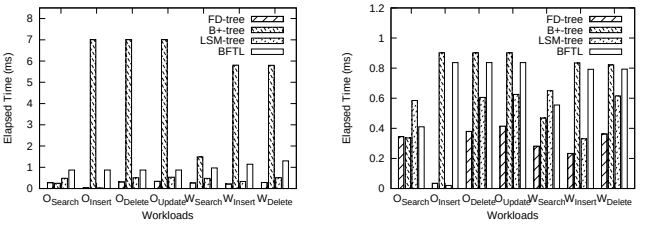
Figure 7(a) demonstrates the accumulated elapsed time of over one million queries consisting of 50% searches and 50% insertions, with and without deamortized execution. We separately show the accumulated time for searches and insertions, as well as the accumulated total time for both of operations. In the query execution without deamortization, we observe a sharp increase in both of the accumulated overall time and insertion time (also implied by the spikes in Figure 6). The high elapsed time (around 10 seconds) of a particular insertion operation is caused by a merge operation. With the deamortization, the accumulated time increases smoothly when a merge starts, because the high merge cost is amortized to thousands of insertions afterwards. Nevertheless, the search time is slightly increased by the deamortization before the merge completes. The reason is that FD-tree maintains two head trees to overlap the executions of insertions and the merge operation, which results in extra searches on the temporary head tree.

The average and worst execution times with and without deamortization are illustrated in Figure 7(b). Deamortization significantly reduces the worst elapsed time (by 80–2500X) while introducing a slight overhead on the average elapsed time (less than 5%). Since deamortized deletion is implemented with a deamortized search and a deamortized insertion, the performance is similar to the overall performance of the 50% search 50% insertion workload.



(a) On Mtron SSD (b) On Intel SSD

Figure 11: Search performance comparison varying selectivity



(a) On Mtron SSD (b) On Intel SSD

Figure 12: Performance comparison on flash SSDs

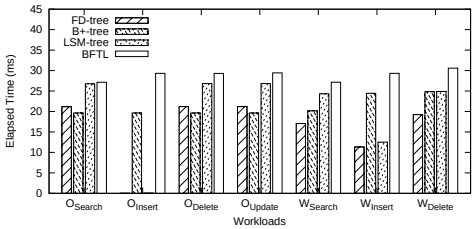


Figure 13: Performance comparison on hard disk

5.5 Performance comparison on flash SSDs

Figures 8 shows the performance comparison for the search-only workload. On the Mtron SSD, 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 has a performance similar to B+-tree on small indexes. When the index size exceeds 2GB, FD-tree is slightly slower than B+-tree, since FD-tree is taller than B+-tree. LSM-tree is slower than both B+-tree and FD-tree, because a

single search on LSM-tree requires searching on multiple B+-trees. On the Intel SSD, the search performance comparison is similar to that on Mtron SSD, except that BFTL outperforms LSM-tree.

Figure 9 shows the insertion performance of the four indexes. LSM-tree and FD-tree are over an order of magnitude faster than the other two indexes due to their logarithmic structures with multi-page I/O optimization. Specifically, when the index size is 8GB, FD-tree is around 35X and 280X faster than BFTL and B+-tree on the Mtron SSD, respectively. FD-tree is 10-50% slower than LSM-tree due to its fence structure.

The overall performance of the four indexes is shown in Figure 10. On the Mtron SSD, the gap of overall performance among the four indexes is very large. B+-tree is the slowest because its overall performance is dominated by the poor insertion performance. BFTL reduces the insertion cost by degrading search performance, and achieves a balanced performance between search and insertion. It has a better overall performance than B+-tree but is still much worse than FD-tree and LSM-tree. FD-tree outperforms all other three indexes for all index sizes. Specifically, when the index size is 8GB, the speedup of FD-tree is around 24.2X, 5.8X, and 1.8X over B+-tree, BFTL and LSM-tree, respectively. As for the overall performance comparison on Intel SSD, the speedup of FD-tree over other competitors is not as significant as on Mtron SSD. Specifically, when the index size is 8GB, the speedup of FD-tree over B+-tree and BFTL is 3.3X, and the speedup of FD-tree over LSM-tree is 1.6X.

Figure 11 shows the performance comparison for the range search varying the number of entries in the search range. The elapsed time of all four indexes gradually increases when the matching entries occupy more than one page. With the knowledge about the number of pages to be retrieved in the next level and the structure of sorted run, FD-tree exploits the multi-page I/O technique. On a B-tree and BFTL, sibling nodes may not be placed on consecutive physical pages, and lose the opportunity of multi-page I/O optimization. Therefore, search on FD-tree is 6–10X and 6–9X faster than B+-tree and BFTL for large search ranges, respectively.

Figure 12(a) shows the elapsed time for different workloads on Mtron SSD. The workloads include W-Search, W-Insert and W-Delete, and four workloads with only searches, insertions, deletions and updates, (denoted by O-Search, O-Insert, O-Delete, and O-Update, respectively). We performed 10 million queries for each workload. The deletions and updates on FD-tree are 20.6-22.9X, 1.6-1.7X, 2.6-2.9X faster than those on B+-tree, LSM-tree and BFTL, respectively. For W-Search on Mtron SSD, the speedups of FD-tree over B+-tree, LSM-tree and BFTL are 5.7X, 1.6X, 3.7X, respectively. For W-Insert and W-Delete, FD-tree is over 20.5-27.9X, 1.4X, 4.6-5.5X faster than B+-tree, LSM-tree and BFTL. The results on Intel SSD are shown in Figure 12(b). Due to the higher speed on random write of Intel SSD, the performance of both B+-tree and BFTL are improved significantly. In specific, B+-tree outperforms LSM-tree on the search-intensive workload. While FD-tree has a smaller speedup on Intel SSD than on Mtron SSD, it exhibits the best performance for the mixed workloads.

5.6 Performance comparison on hard disk

We also study the performance of FD-tree on hard disk (Figure 13). We performed 1,000,000 queries for each workload. FD-tree and LSM-tree have a superior insertion performance by adopting the logarithmic method, but their overall performance is significantly limited by the search efficiency due to the poor random read speed on hard disk. As a result, FD-tree has a similar performance to the competitors under search- and deletion-intensive workloads, and have a 1.1-2.6X speedup over other competitors under insertion-intensive workloads.

6. CONCLUSIONS

Due to the asymmetric speeds of reads and writes of the flash disk, data structures and algorithms originally designed for the hard disk require a careful adaptation or even redesign to suit the flash disk. In this paper, we 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 the 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.

7. REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *PVLDB*, 2(1):361–372, 2009.

[2] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *WADS*, 1995.

[3] J. L. Bentley. Decomposable searching problems. *Inf. Process. Lett.*, 8(5), 1979.

[4] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. In *CIDR*, 2009.

[5] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2), 1986.

[6] F. Chen, D. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS*, 2009.

[7] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD Conference*, 2009.

[8] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2), 1979.

[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2), 2005.

[10] G. Graefe. Write-optimized b-trees. In *VLDB*, 2004.

[11] G. Graefe. The five-minute rule 20 years later: and how flash memory changes the rules. *ACM Queue*, 6(4):40–52, 2008.

[12] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *ACM Queue*, 6(4):18–23, 2008.

[13] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2), 2007.

[14] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, 1997.

[15] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data warehouse insertion. In *VLDB*, 1999.

[16] K. Kimura and T. Kobayashi. Trends in high-density flash memory technologies. In *IEEE Conference on Electron Devices and Solid-State Circuits*, 2003.

[17] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD Conference*, 2007.

[18] 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.

[19] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE*, 2009.

[20] C. Manning. Yaffs: the nand-specific flash file system. 2002.

[21] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the lham log-structured history data access method. In *VLDB*, 1998.

[22] D. Myers. On the use of nand flash memory in high-performance relational databases. *MIT Msc Thesis*, 2008.

[23] S. Nath and A. Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN*, 2007.

[24] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4), 1996.

[25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1), 1992.

[26] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD Conference*, 2009.

[27] D. Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*, 2001.

[28] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. In *RTCSA*, 2003.

APPENDIX

A. PRELIMINARY ON FLASH SSD

This section introduces the preliminary on flash SSDs for reference. The flash memory has been the main-stream storage in mobile devices and embedded systems due to its superior characteristics. Recently, many manufactures pack flash memory into Solid State Disks (SSDs) with the same interface as the magnetic disks. As the capacity increases, SSDs have become attractive for personal computers and high-end servers, because of its extremely low access latency and power consumption. The most common type of flash memory in SSDs is NAND flash. In the paper, we use flash SSDs to denote NAND-flash based SSDs.

Flash memory is a non-volatile storage media with unique characteristics. Both reads and writes of NAND flash memory are at the granularity of flash pages. A typical size of flash page is between 512B to 2KB. Due to the physical characteristics of flash memory, writes are only able to change bits from 1 to 0. Thus, an erase operation that sets all bits to 1 must be performed before rewriting. However, the unit of erase operations is *block*, which typically contains 16-64 pages. Moreover, the latency of erase operations is far higher than reads or writes. As a result, this *erase-before-write* mechanism causes inferior write, especially random write performance of flash memory. In addition, each flash block can only be erased by a finite number of times before wearing out. Once a flash block wears out, it cannot be reused any more.

Flash memory used in flash SSDs can be categorized into two types. High-end flash disks use Single Level Cell (SLC) flash memory that stores one bit of data per cell. The alternative is Multi Level Cell (MLC) flash memory that uses four voltage levels and can thus store two bits of data per cell. While MLC flash has twice the density of SLC, it has an inferior read/write performance and fewer erase cycles before wearing out.

Flash SSDs are built on an array of flash memory chips. A logical page might span on multiple flash memory chips, and thus create the potential for leveraging parallelism within drives. Those drives provide a disk-like bus interface on top of the flash memory chips. To emulate a traditional hard disk interface that has no erase operation, flash SSDs employ a firmware layer, called the *flash translation layer* (FTL), to implement an out-place update strategy by maintaining a mapping table between the logical and physical pages. As writes in flash memory cannot be performed in place, each write of a logic page is actually performed on a different physical page. On a write request of a page, a block with the size of mapping granularity is rewritten to another place, and the corresponding entry in the mapping table is updated to reflect the new physical address. The mapping table is maintained in persistent flash memory and rebuilt in a volatile RAM buffer at startup time. Besides the address mapping, the FTL takes the responsibilities of *garbage collection* and *wear leveling*. The garbage collector copies the valid pages into a free area and erases the old area for future use. Wear-leveling is a technique that prolongs the life time of the flash disk by evenly distributing the writes across the entire flash disk. In addition, Flash SSDs are usually equipped with an on-drive RAM cache for improving the performance of writes with high locality.

B. ALGORITHM PSEUDOCODE

The algorithm pseudocode of search, insertion, merge, deletion is given in Algorithms 1, 2, 3 and 4, respectively.

Algorithm 1 Search(K)

Parameter: K: the search key

F = NULL, $R = \emptyset$; //F: the filter entry, R: the result set
Search L_0;
Let pid be the id of the page containing the entry whose key value is the greatest among those equal to or smaller than K in L_0;
for each level L_i in FD-tree **do**
  Perform a binary search on the page whose id is pid;
  let e be the largest entry that is equal or smaller than K;
  while $e.type \neq External$ and $e.type \neq Internal$ **do**
    if $e.type$ = Filter **then**
      $F = e$;
    else
      if $F.key \neq e.key$ or $F.rid \neq e.rid$ **then**
        return e;
    Let e be the previous entry in L_i;
  $pid = e.pid$; /*the next-to-go page in L_{i+1}*/
return R

Algorithm 2 Insert(e)

Parameter: e, the entry to be inserted into the FD-tree.

Insert e into L_0 ;
if L_0 reaches its level capacity **then**
  Merge(L_0, L_1); //See Algorithm 3

Algorithm 3 Merge(L_{i-1}, L_i)

Parameter: L_{i-1},L_i: the levels to be merged

Let e_{i-1} and e_i be the first entry in L_{i-1} and L_i, respectively;
while $e_{i-1} \neq null$ and $e_i \neq null$ **do**
  while $e_{i-1}.type =$ Fence **do**
    Let e_{i-1} be the next entry of L_{i-1};
  while $e_i.type =$ Internal Fence **do**
    Let e_i be the next entry of L_i;
  if $e_i.type$ = Normal and $e_{i-1}.type$ = Filter and $e_i.key = e_{i-1}.key$ and $e_i.rid = e_{i-1}.rid$ **then**
    Let e_{i-1} and e_i be the next entry of L_{i-1} and L_i, respectively;
  if $e_{i-1}.key \leq e_i.key$ **then**
    $entryToInsert = e_{i-1}$;
    Let e_{i-1} be the next entry of L_{i-1};
  else
    $entryToInsert = e_i$;
    Let e_i be the next entry of L_i;
    if $entryToInsert.type$ = Fence **then**
      $lastFence = entryToInsert$;
  if the current page in L'_i is empty **then**
    if $entryToInsert.type \neq$ Fence **then**
      $internalFence.key = entryToInsert.key$;
      $internalFence.rid = lastFence.rid$;
      Write $internalFence$ to L'_i;
    Write $entryToInsert$ to L'_i;
    $externalFence.key = entryToInsert.key$;
    $externalFence.rid =$ ID of current page in L'_i;
    Write $externalFence$ to L_{i-1}; //This may invokes writes external fences to the higher levels;
  else
    Write $entryToInsert$ to L'_i;
if L'_i reaches its level capacity **then**
  Merge(L'_i, L_{i+1});
Replace L_i by L'_i;

Algorithm 4 Delete(q)

Parameter: q, the predicate of the deletion query.

Perform q on the head tree;
Search q on FD-tree, let the result entry be e;
$e.type = Filter$;
Insert(e); //See Algorithm 2

C. COST ANALYSIS AND COST MODEL

This section presents the details on the cost analysis and cost model of FD-tree.

C.1 Search Time Analysis

The cost of a lookup consists of two parts: the search cost on head tree, and the search cost on lower sorted runs. For the first part, $\lceil \log_f |L_0| \rceil$ pages are retrieved, similar to a lookup on B+-tree. For the second part, the lookup operation is performed by retrieving a page at each level and finding a fence within this page. At a certain level, the I/O cost to find the fence is one, because each page has at least one matching fence (the first entry in the page). With the number of levels in the index $l = \lceil \log_k n/|L_0| \rceil + 1$, we have the estimated time of a search:

$$t_{search} = \frac{B \cdot (\lceil \log_f |L_0| \rceil + \lceil \log_k n/|L_0| \rceil)}{W_{rrnd}(B)} \tag{1}$$

Since $\lceil \log_f |L_0| \rceil + \lceil \log_k n/|L_0| \rceil = O(\log_k n)$, we show that FD-tree serves a lookup in $O(\log_k n)$ random reads.

C.2 Insertion Time Analysis

Each insertion on FD-tree causes O(1) random write on the head tree, which is small and of a high locality. However, some insertions may invoke expensive merge operations. In this subsection, we will show that an insertion on FD-tree amortizedly requires $O(\frac{k}{f-k} \log_f n)$ sequential reads and writes. Moreover, we firstly relax our assumption that the size ratios between adjacent levels must all be equal for an optimal insertion performance, i.e., we define $k_i = |T_i|/|T_{i-1}|$ $(1 \le i < l)$, and later prove that FD-tree has a minimal amortized merge cost when $k_1 = k_2 = \cdots = k_{l-1}$ in Theorem 1.

The time cost of an insertion consists both of insertion cost on the head tree and the merge cost on lower sorted runs, i.e. $t_{insert} = t_{headtree} + t_{merge}$. Following the design principle $\mathcal{P}2$, those random writes on the head tree have a similar performance with sequential ones. Thus, we use the bandwidth of sequential write to calculate the time cost of random writes on the head tree, i.e. we have $t_{headtree} = \frac{1}{W_{wseq}}$. Next, we will show the amortized merge cost per insertion t_{merge} by deriving the total merge time during n continuous insertions.

Firstly, we focus on the merges between two levels L_{i-1} and L_i $(1 \le i < l)$. Let $\mathcal{M}_i$ denote the set of merges occurred between the two levels when performing n insertions, and m_i being the number of merges in $\mathcal{M}_i$. Since the sizes of level L_{i-1} and L_i are changed as the merges in $\mathcal{M}_i$ occur, we use $|L_{i-1}^j|$ and $|L_i^j|$ to denote the current number of entries in the level L_{i-1} and L_i immediately after the completion of the j^{th} merge between them. According to the insertion and merge algorithms we described in Section 3.3, FD-tree has the following two properties.

Property 1. When the j-th merge in $\mathcal{M}_i$ is completed, the upper level, L_{i-1}, contains only external fences. Thus, we have

$$|L_{i-1}^j| = |L_i^j|/f \tag{2}$$

Property 2. Since all n inserted entries will go through level L_{i-1} and will be moved into level L_i, we have

$$n = \sum_{j=1}^{m_i} (|L_{i-1}| - |L_{i-1}^j|) = m_i \cdot |L_{i-1}| - \sum_{j=1}^{m_i} |L_{i-1}^j| \tag{3}$$

By substituting Eq. (2) and Eq. (3), we derive the number of entries to be written and read, N_{write}^i and N_{read}^i respectively, for all m_i merges in $\mathcal{M}_i$.

$$\begin{aligned} N_{write}^i &= \sum_{j=1}^{m_i} (|L_i^j| + |L_{i-1}^j|) = \sum_{j=1}^{m_i} (1+f) \cdot |L_{i-1}^j| \\ &= (1+f)(m_i \cdot |L_{i-1}| - n) \end{aligned} \tag{4}$$

$$\begin{aligned} N_{read}^i &= \sum_{j=1}^{m_i} (|L_i^{j-1}| + |L_{i-1}|) = N_{write}^i + \sum_{j=1}^{m_i} (|L_{i-1}^{j-1}| - |L_{i-1}^j|) \\ &= N_{write}^i - n/(f-1) \end{aligned} \tag{5}$$

Given the maximum number of external fences $|L_i|/f$, and interval fences $|L_{i-1}/f|$ on level L_{i-1}, we have the upper bound for m_i $(1 \le i < l)$

$$m_i < \frac{n}{|L_{i-1}| - |L_{i-1}|/f - |L_i|/f} = \frac{f}{f - k_i - 1} \cdot \frac{n}{|L_{i-1}|}$$

The upper bound of the amortized merge cost per insertion operation is given in Eq. (7).

$$\begin{aligned} t_{merge} &= \frac{1}{n} \cdot \sum_{i=1}^{l-1} \left(\frac{R \cdot N_{write}^i}{W_{wseq}} + \frac{R \cdot N_{read}^i}{W_{rseq}}\right) \\ &< \frac{W_{wseq} + W_{rseq}}{W_{wseq} \cdot W_{rseq}} \cdot B \cdot \sum_{i=1}^{l-1} \left(\frac{m_i \cdot |L_{i-1}|}{n} - 1\right) \qquad (6) \\ &< \frac{W_{wseq} + W_{rseq}}{W_{wseq} \cdot W_{rseq}} \cdot B \cdot \sum_{i=1}^{l-1} \left(\frac{f}{f - k_i - 1} - 1\right) \qquad (7) \end{aligned}$$

THEOREM 1. *Given an FD-tree with l levels, the amortized cost of insertion is minimized when $k_1 = k_2 = \cdots = k_{l-1}$.*

PROOF. Since $t_{insert} = t_{headtree} + t_{merge}$ and $t_{headtree}$ is a constant here, we will prove that the amortized merge cost is minimized when $k_1 = k_2 = \cdots = k_{l-1}$. We firstly rewrite Eq. (7) in form of $t_{merge} < \Phi \cdot \sum_{i=1}^{l-1} (\frac{f}{f-k_i-1} - 1)$, where $\Phi = B \cdot \frac{W_{wseq}+W_{rseq}}{W_{wseq} \cdot W_{rseq}}$.

By the design of FD-tree, we have

$$\prod_{i=1}^{l-1} k_i = \frac{n}{|L_0|} \tag{8}$$

Next, we apply the Geometric Mean inequality three times:

$$\begin{aligned} \Phi \cdot \sum_{i=1}^{l-1} \left(\frac{f}{f-k_i} - 1\right) &= \Phi \cdot \sum_{i=1}^{l-1} \frac{k_i}{f-k_i} \\ &\ge \Phi \cdot (l-1) \cdot \sqrt[l-1]{\prod_{i=1}^{l-1} \frac{k_i}{f-k_i}} \\ &\ge \Phi \cdot (l-1) \cdot \sqrt[l-1]{\frac{n}{|L_0|}} \cdot \frac{l-1}{\sum_{i=1}^{l-1} (f-k_i)} \\ &\ge \Phi \cdot (l-1) \cdot \sqrt[l-1]{\frac{n}{|L_0|}} \cdot \frac{1}{f - \sqrt[l-1]{\prod_{i=1}^{l-1} k_i}} \\ &= \Phi \cdot (l-1) \cdot \sqrt[l-1]{\frac{n}{|L_0|}} \cdot \frac{1}{f - \sqrt[l-1]{\frac{n}{|L_0|}}} \end{aligned}$$

All the three equalities hold if and only if all k_i are equal, proving the theorem. □

In the rest of the paper, we assume $k_1 = k_2 = \cdots = k_{l-1}$, and thus use k to represent k_i for simplicity. Then, Eq. (7) can be

rewritten into Eq. (9), which clearly shows that the amrotized time cost of an insertion on FD-tree is the time of performing $O(\frac{k}{f-k}\log_k n)$ sequential reads and $O(\frac{k}{f-k}\log_k n)$ sequential writes.

$$t_{merge} < \frac{k+1}{f-k-1} \cdot \frac{W_{wseq}+W_{rseq}}{W_{wseq}\cdot W_{rseq}} \cdot B \cdot \lceil \log_k n/|L_0| \rceil \quad (9)$$

C.3 Cost Model for Parameter Setting

We analytically develop a cost model to determine the optimal k value in order to achieve the optimal overall performance given the characteristics of both workload and flash SSDs.

The search cost analysis (Eq. (1)) does not take the buffer pool into consideration, which is widely employed and plays a key role in real systems. To estimate the optimal configuration, we develop a cost model with the buffer pool considered. Based on the access path on tree indexes, it is commonly held that the nodes at a higher level have a larger possibility that they reside in the buffer pool. Since the head tree is so small that it is very likely to fit into memory, we omit the cost of a lookup on the head tree. We model that the top $log_k(M/|L_0|)$ levels could reside in a buffer pool of size M. By extending Eq. (1), the estimated search cost with the buffer pool considered, $\bar{t}_{search}$, is given in Eq. (10).

$$\bar{t}_{search} = \frac{B \cdot (\lceil \log_k n/|L_0| \rceil - \log_k M/|L_0|)}{W_{rrnd}(B)} \quad (10)$$

While the amortized insertion cost is bounded by Eq. (9), that equation is an over-estimation. Instead of using an analytical model, we use an estimation model that simulates the amortization. Algorithm 5 simulates the procedure that sufficient entries are inserted into an FD-tree to make the lowest two levels merge. In order to calculate the amortized insertion time, we count the number of entries that have been read and written, as well as the number of insertions.

Let $|L_i|$, $|L'_i|$ denote the capacity, and the current cardinality of level i, respectively. In the outer loop of Algorithm 5 (Lines 3-6), we count the number of insertions occurred between two consecutive merges. After the first merge operation, L_0 contains $|L'_0|$ entries. Thus, $|L_0| - |L'_0|$ insertions are performed before the next merge operation occurs. The number of insertions are accumulated into a variable $numInsert$ (Line 4). The merge operations are simulated in the inner loop (Lines 7-12). In each iteration of the inner loop, two adjacent levels are merged. This process continues from the top to the bottom levels until a level does not exceed its capacity. The numbers of entries read and written are maintained in variables $numRead$ and $numWrite$, respectively. The current cardinality of L_{i+1} advances by the number of non-fence entries in L_i (Line 9). The cardinality of L_i is then set to L'_{i+1}/f (Line 10), which can be derived from Eq. (2). Finally, we include the time for sequential reads an writes, and return the average cost (Line 13).

Algorithm 5 Insertion Cost Estimation

```
1: i = 0;
2: numInsert = numRead = numWrite = 0;
3: while i < l − 1 do
4:     numInsert = numInsert + (|L_0| − |L'_0|);
5:     |L'_0| = |L_0|;
6:     i = 0;
7:     while |L'_i| ≥ |L_i| do
8:         numRead = numRead + (|L'_i| + |L'_{i+1}|);
9:         |L'_{i+1}| = |L'_{i+1}| + (|L_i| − |L'_i|);
10:        |L'_i| = |L'_{i+1}|/f;
11:        numWrite = numWrite + (|L'_i| + |L'_{i+1}|);
12:        i = i + 1;
13: return (numRead·E/W_rseq + numWrite·E/W_wseq) / numInsert;
```

The computation cost of Algorithm 5 is $O(\frac{n}{k\cdot|L_0|})$. This cost is low, e.g., the computation on an 8GB FD-tree completes in 50ns on our experimental platform.

By now, we have both of the estimated search time cost $\bar{t}_{search}$ and insertion time cost $\bar{t}_{insert}$. Next, we develop a model to estimate the query time under a certain workload. Suppose the ratios of search, deletion, insertion, and update operations in the workload are p_{search}, $p_{deletion}$, p_{insert} and p_{update}, respectively. Deletion and update operations are implemented using search and insertions. In particular, the deletion is implemented by a search followed by an insertion. The update is performed as a deletion and an insertion. Thus, we can perform an update by a search and two insertions. We define p_s as the normalized percentage of search operations in the workload, as given in Eq. (11). We further define the normalized percentage of insertion to be $p_i = 1 - p_s$.

$$p_s = \frac{p_{search} + p_{deletion} + p_{update}}{p_{search} + p_{insert} + 2p_{deletion} + 3p_{update}} \quad (11)$$

Given a workload of the search and the insertion rates being p_s and p_i, respectively, the total execution time T_{tot} of the workload is given in the equation $T_{tot} = p_s \cdot \bar{t}_{search} + p_i \cdot \bar{t}_{insert}$. We can enumerate the k values, calculate T_{tot} for each value, and determine the suitable k value that minimizes T_{tot}.

D. DETAILED EXPERIMENTAL SETUP

We present the detailed experimental setup to help understanding the experiment. We select three commodity SSDs by three major SSD manufacturers: Samsung MCBOE32G8APR-0XA00 32GB, Mtron MSD-SATA3035 64GB, and Intel X25-M 80GB. Some features of the three SSDs are summarized in Table 3. The basic I/O cost metrics for Mtron and Intel SSDs are shown in Figure 1.

Implementation details. All implementation is written in C language, and is compiled with MSVC 8.0 with full optimizations on.

We have implemented a storage manager that uses standard OS file system facilities. The components of index are stored in large files in the file system, which are treated as linear arrays of disk-resident pages. We adopt the fixed-size page format in our storage manager. Each page has a page header containing multiple fields such as the number of entries in the page. The rest of the page is for the entries and is organized as the slotted page layout for fixed-length records. We set the size of index node(page) to be consistent with the physical page size of the disk, e.g. $B = 2K$ bytes. We understand that the consecutive pages in a large file may not be placed entirely consecutively on the physical device, but it was shown that the negative impact was insignificant [13].

An LRU buffer manager is implemented for caching pages recently read and written. A written page in the buffer pool is firstly marked as a dirty page, and will later be written to disk when it is evicted by the replacement policy. To avoid the interference between the virtual memory of the operating system and our buffer manager, we disabled the buffering functionality of the operating system using Windows APIs.

In the implementation of FD-tree, the $type$ and pid fields of entries are packed into one integer: 30 bits for pid and 2 bits for $type$. The size of the head tree is fixed to 512KB. The merge operation on FD-tree can skip the buffer layer and directly read or write pages from or to the storage layer in order to exploit the multi-page I/O optimization and prevent hot pages from being evicted from the buffer pool. The access unit size of sequential I/Os is set to 512KB.

The B+-tree implementation follows the previous study [8]. If a node is full, an insertion causes a split on the node. If a node is less than half full, we combine it with its sibling node.

Table 3: Specifications of flash SSDs

	Samsung SSD	Mtron SSD	Intel SSD
Model	MCBOE32G8APR	MSD-SATA3035	X25-M
Mapping Table in FTL	Block-Level Mapping	Block-Level Mapping	Page-Level Mapping
Capacity	32GB	64GB	80GB
Memory	MLC	SLC	MLC
Interface	ATA	SATA	SATA

LSM-tree is implemented as a forest consisting of multiple B+-tree components. Similar to FD-tree, the tree components are designed to be of stepped sizes. We set the size ratio between two tree components to be the same as the k value, and perform the same tuning as that on FD-tree. The size of the smallest tree is set to the locality area size, the same size as the head tree in FD-tree. The multi-page I/O optimization is also applied to the merge operation between tree components.

We implemented BFTL at the application level and tuned its performance on the flash SSD. One important parameter for BFTL is c, the maximum number of pages per tree node scattered over the disk. This parameter is a factor to balance the asymmetric performance of search and updates. Since BFTL is originally designed for embedded systems with small flash cards, the recommended c value is not suitable for the large flash SSDs. We varied the parameter and found that $c = 16$, $c = 11$, and $c = 2$ is the best configuration for balance this asymmetry on Samsung SSD, Mtron SSD and Intel SSD respectively. Since an index node may scatter over multiple pages, BFTL needs around $nEC/2f$ memory for storing the mapping table, for example, 256MB for an 8GB index on Mtron SSD. We separate this memory area from the buffer pool.

Workload Design. We used the workloads with search only or update only, as well as mixed ones with different operations. In particular, we use a workload of 80% searches, 10% insertions, 5% deletions and 5% updates to simulate a workload dominated by reads, denoted as W-Search. We define W-Insert as a workload consisting 20% searches, 50% insertions, 20% deletions and 10% updates to simulate a workload dominated by insertions. We define W-Delete as a workload consisting 20% searches, 20% insertions, 50% deletions and 10% updates to simulate a workload dominated by deletions.

All experiments focus on evaluating long running indexes. At the beginning of all experiments, B+-tree is built by bulk loading with the load factor of 0.7. Then we insert entries to make 20% leaf nodes split. We build FD-tree and LSM-tree in a similar process. The index is first built by bulk loading and then we keep inserting entries into the index until each level of FD-tree or each tree component of LSM-tree is at least half full. For BFTL, after bulk loading, we keep inserting entries into the index until each node scatters over multiple pages.