Indexing Structures for flash based Solid State Drives

SeungBum Jo[1,3], Vineet Pandey[2,3], and S. Srinivasa Rao[1]

[1] School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Republic of Korea
sbcho@theory.snu.ac.kr, ssrao@cse.snu.ac.kr
[2] Computer Science & Information Systems, BITS Pilani, 333031, India
vineetp13@gmail.com
[3] Student authors

Abstract. Performance computing is fast becoming the mainstay of computer science. Despite massive increase in the processing power, performance bottlenecks exist due to much lesser throughput on the storage side. It is a matter of time before conventional magnetic disks make way for a faster technology. Flash memory devices are becoming ubiquitous and indispensable storage devices, partly replacing the traditional hard-disks. To store data efficiently on these devices, it is necessary to adapt the existing file systems and indexing structures to work well on flash memory, and a significant amount of research in this field has been devoted to designing such structures. But it is hard to do a provable comparison between these structures owing to the lack of any analytical memory models for flash memory since the existing external memory models fail to capture the full potential of flash-based storage devices. In this paper we study various index structures that have been shown to perform well on SSDs, and analyze them in the recently proposed flash memory models. This is the first such work to compare the provable performance of various flash based data structures on a common flash memory model.

Key words: Storage systems, Storage performance, Solid State Drive, flash memory, tree indexes, flash memory model.

1 Introduction

Flash memory is non-volatile computer memory which can be erased and programmed. Flash memory devices have received increasing attention amongst the storage community because they are lighter, provide greater throughput and greater shock resistance while consuming lesser power as compared to magnetic disks. Flash memory has been put to use in low power devices and it is believed that they will be used for commercial large-scale usage either alongside or by replacing traditional disk-based storage. Compared to magnetic disks, flash memory exhibits some unique characteristics such as asymmetric read and write speeds (read-write bias), erase before write, out-of-place update and weariness. Flash disks come as raw NAND memory or as commercial Solid State Disks (SSDs). The SSDs come prepackaged with a *Flash Translation Layer (FTL)* which distributes erases uniformly across the disk to prevent early failure. We consider the problem of storing indexing data structures on a flash based device where the index contains pointers to the data records which are stored on a different secondary disk. With the rapid growth of flash memory capacity, the implementation of index structures originally designed for disk-based external memory can become a bottleneck. Despite significant work in the field of flash-specific data structures, there is a lack of a central model to theoretically analyze the performance of the several proposed structures. By means of this study, we intend to develop a framework to compare various secondary indexing structures for flash memory.

2 Flash Memory Models and B$^+$-tree

In the past few years, there have been efforts to characterize the flash devices to develop efficient algorithms for them which may ultimately influence the exact architecture of future flash devices. Ajwani et al. [2] proposed the following

two computational models for analysing the performance of algorithms on flash memory.

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

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

B$^+$-tree. A *B$^+$-tree* [4] is a data structure which is used to manage databases efficiently. It maintains a collection of records in the sorted order of their keys and allows for find, insert and delete operations to be performed in time proportional to the height of the tree. Let us consider a B$^+$-tree where each node is of size B and the total number of elements in the tree is N. The number of elements in a node and hence the branching factor at every node is maintained between $B/2$ and B. Therefore, the height of a B$^+$-tree is $O(\log_B N)$ which is also the time taken to perform find, insert or delete operations.

We assume the flash disks to be of NAND type in which every available EU is cleared (set to 1) to begin with, and writing a page means changing the 1s selectively to 0s. Reads and writes can happen at a page level but rewriting a page requires erasing the entire EU which has a huge cost penalty associated with it. A page though can be invalidated and its contents can be written to a new page in the same or a different EU. All pointers using the physical address of the previous page (now invalidated) need to be changed accordingly.

N	Number of records in the indexing tree
B	Size of a node in the indexing tree
B_R	Size of a read-block in flash memory
B_W	Size of a write-block in flash memory
B_U	Size of buffer
EU	Erase Unit

Table 1: List of symbols used in the text

We analyze the worst-case complexity of search and calculate the amortized cost for update for various indexing structures based on B$^+$-tree. Based on the cost of operations, we predict theoretically which data structure works well for different conditions. Once we have the values (x and y in the models), we can obtain the performance in either the general-cost or the unit-cost model. We assume that $B_W \geq B_R$.

3 Proposed variants of B$^+$-tree for flash memory

Various modifications have been proposed to B$^+$-tree to reduce the number of writes, such as using a main memory buffer to apply a group of updates together and varying the size of the node according to the level in the tree.

3.1 Flash Translation Layer (FTL)

Since in-place update is not allowed by flash memory, updating a page requires one to invalidate that page and write the contents (along with modifications) to a different page altogether. Flash Translation Layer (FTL) [3] solves the problem of updating multiple nodes for one update by using logical addresses instead of physical addresses for pointers and maintaining a logical to physical address map. FTL enables a *wear-levelling* policy to be used, which distributes

erases uniformly across EUs. It can also provide a *sector* based access to directly implement existing magnetic-disk based algorithms on the flash disk. Using FTL along with a standard B$^+$-tree brings down the update cost from $O(\log_{B_W} N)$ write I/Os to $O(1)$ write I/Os while maintaining the cost of find operation as $O(\log_{B_R} N)$ read I/Os,

3.2 Lazy Adaptive tree

The Lazy Adaptive tree proposed by Agrawal et al. [1] has a flash-resident buffer at every k^{th} level from the root to avoid high update cost of flash memory by grouping together update operations in buffers. An optimal online algorithm named ADAPT dynamically determines whether to empty the buffer and update the contents of the descendant nodes or append the update request to the buffer. The ADAPT algorithm estimates benefit of emptying buffer using buffer size and buffer scan cost at each lookup and if this benefit is larger than the cost of emptying the buffer, then the ADAPT algorithm empties the buffer at that lookup. We assume that LA-tree is used over FTL, the effective buffer size B_U is same at every level and it is emptied only for insertion.

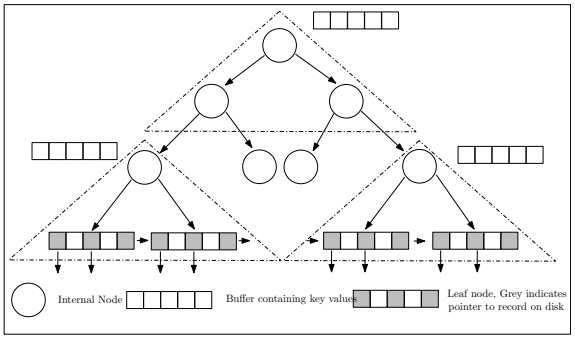

Fig. 1: Lazy-Adaptive tree

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

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

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

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

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

The search and update costs of LA tree are both inversely proportional to the parameter k. Hence, it would seem to make sense to choose

k as large as possible (i.e., equal to the height of the tree), in which case, the performance matches that of B$^+$-tree. But the actual performance of LA tree is better than that of B$^+$-tree in practice as the buffers are flushed adaptively during both searches and updates, and buffer sizes are also not the same for all the nodes.

3.3 μ-tree: minimally updated tree

The *minimally updated tree* or μ-tree proposed by Kang et al. [5] is a balanced tree similar to B$^+$-tree, which reduces the number of pages written by using varying sizes for nodes depending on their distance from the root. It requires a single flash write operation to perform an update to the tree, if no nodes are split during the update.

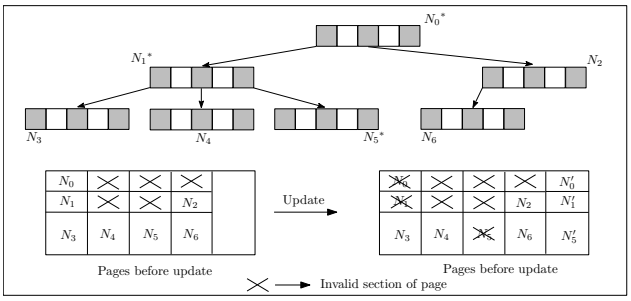

Fig. 2: Update in a μ-tree. Updated nodes and new nodes are marked with * and $'$ resp. The pages are shown before and after the update.

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

Update. Update in a μ-tree involves $O(h)$ read I/Os to perform search and 1 write I/O, since entire path is contained in one read-block. μ-tree provides the same search and update costs as B$^+$-tree with FTL without the overheads.

3.4 FD-tree

The FD tree proposed by Li et al. [7] consist of multiple levels, denoted as $L_0, L_1, \ldots, L_{h-1}$. At the top level, L_0, it has a *head tree* which is a small (i.e., constant height) B$^+$-tree with node size equal to the read-block size. Each of the other levels, $L_1, \ldots, L_{h-1}$, is a sorted run of key values stored in contiguous pages. Each level of the tree has a capacity which is the maximum number of elements that can be stored in that level. The ratio of capacities between any two adjacent levels is same, and is equal to r_c. i.e., for $0 \leq i \leq h-2$, $|L_{i+1}| = r_c \cdot |L_i|$. If the FD-tree contains N keys, then the height h (i.e., the number of levels) of the FD-tree is $O(\log_{r_c} N)$.

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

To search for a given key x, we first search in the head tree, and then follow the appropriate fence pointers to find the target pages at each level, and search in those target pages. To perform an insertion, we first insert the element into the head tree. If at any time, the number of elements in any level L_i, for $0 \leq i < h-1$, exceeds its capacity, the FD-tree merges the elements of L_i with the elements in the adjacent lower level L_{i+1} into a single sorted run (stored in contiguous pages). In the merge process, the tree uses only sequential writes and random writes occur only in head tree. FD-tree performs better thanl B$^+$-tree for update operation in flash memory by converting random writes (typically slow) to sequential writes.

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

els. Thus the search cost is $O(\log_{r_c} N)$ read I/Os.

Update. Li et al. [7] show that the update cost of FD-tree is amortized $O(\frac{r_c}{f-r_c} \log_{r_c} N)$ sequential I/Os, where f is the size of the read-block. But this cost is in terms of read-blocks. Thus to obtain the actual update cost of FD-tree, we need to divide this by B_W/B_R. By choosing $r_c = \Theta(B_R)$, we get the update cost to be $O((B_R/B_W) \log_{B_R} N)$ sequential write I/Os.

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

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

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

4 Discussion and Future work

Data Structure	Search cost (read I/Os)	Update cost (write I/Os)	Reference
B$^+$-tree	$O(\log_{B_R} N)$	$O(\log_{B_W} N)$	[4]
B$^+$-tree (w/ FTL)	$O(\log_{B_R} N)$	$O(1)$	[3]
LA-tree	$O((1 + \frac{B_U}{k \cdot B_R}) \log_{B_R} N)$	$O((1/k) \log_{B_R} N)$	[1]
FD-tree	$O(\log_{B_R} N)$	$O((B_R/B_W) \log_{B_R} N)$	[7]
μ-tree	$O(h)$	$O(1)$	[5]

Table 2: Complexity of operations in various data structures in the general-cost model where the actual update cost also includes read I/Os for search. h = height of the tree.

We discussed the various B$^+$-tree based indexing data structures which have been designed specifically for flash disks and analyzed the cost of performing search and update operations in the general-cost flash memory model. Our further plan is to analyze other indexing data structures proposed for flash memory, namely Inpage logging [6], BFTL (B-tree flash translation layer) [10], FlashDB [8] and Lazy Update tree [9].

Table 2 summarizes the performance of the index structures that we have analyzed in terms of the read and write I/Os. The search cost for all the structures is essentially the same. The update cost of μ-tree and B$^+$ with FTL is better than that LA-tree and the standard B$^+$-tree without FTL. For most practical values of B_R, B_W and N, the FD-tree outperforms all the other structures, as its amortized update cost is less than 1 (for most practical values of the parameters), and also since it has very few random writes.

References

1. Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2:361–372, 2009.
2. Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In Jan Vahrenhold, editor, *SEA*, volume 5526 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2009.
3. Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *J. Syst. Archit.*, 55:332–343, 2009.
4. Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.
5. Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree: an ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 144–153. ACM, 2007.
6. Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, 2007.
7. Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3:1195–1206, 2010.
8. Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 410–419. ACM, 2007.
9. Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update b+-tree for flash devices. In *Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, MDM '09, pages 323–328. IEEE Computer Society, 2009.
10. Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6, 2007.