

FBX-Tree: A Write Intensive Structure for Flash Disk

Yongming Luo

Harbin Institute of Technology

luoyongming@hit.edu.cn

Abstract

Recently, flash disks have not only been widely used in embedded systems, but also start to play an important role in the commercial computing environment. Due to the unbalanced read/write performances, conventional external memory algorithms have to be reexamined for flash disk. In this paper, we take the moving object database for example to illustrate the strategy on how to adjust classic algorithms to the flash disk cost model. We propose a moving objects indexing method for flash disk called FBX-Tree. By using B^x -Tree as its query processing front-end, the method maintains a Node Translation Table (NTT) and a set of log lists in memory. Through the system, all updates of the objects become the append operations to the database file. We also present two on-line algorithms for making the log clean decision. Analysis and experiment results show that our approach keeps the balance among the data file size, main memory footprint and system efficiency, meanwhile makes a longer life-span for flash disk.

Keywords: Flash Disk, Storage System, Spatial Index, Moving Object Database, B^x -Tree

1 Introduction

Because of the shock resistance, low access latency and low energy consumption, flash disks have been widely used in the mobile computing environment. Lately, people begin to use flash disks in the commercial computing environment. Manufacturing companies equip flash disks in their desktops and laptops. Famous search engine companies (Google[1], Baidu, etc.) use flash disks as their second storage layer devices to accelerate the search performances. With the growing of flash industry, flash disks are expected to completely replace the magnetic disks in the near future.

Recently, the data management issues over flash disk become a hot spot in database research[2, 3, 4, 5, 6]. Previous works mainly pay their attention to the index structure design and the query processing techniques on flash. But few work exists with the topic of how to design structures for a write intensive environment on flash. There are two challenges for this problem: (1) due to the slow random write speed, flash disk would perform unpleasantly in a write intensive environment; (2) random write will trigger more erase operations to the disk which will reduce the device's lifetime.

In this paper, we take the moving object database (MOD) for example to introduce a design strategy for the write intensive algorithms on flash disk. Our method get over both of the above challenges. First, through the experiments, we present the deficiency of the existing B^+ -tree based methods in MOD. On the knowledge of flash disk, we design an indexing method called FBX-Tree. By using B^x -Tree as its front-end and a sophisticated structure in memory, all updates to the system become append operations to the data file. Last, We build a cost model for flash disk and analyze our method on it.

Our main contributions are:

- We propose the method of organizing moving objects over flash disks. Though optimized for the MOD area, the philosophy can be well integrated with other block based structures.
- We study the log clean problem (i.e. when to clean the log list during the operation sequence). We present two on-line algorithms for this problem, and prove their competitiveness.
- We implement all proposed algorithms and conducted an extensive experimental study. The experiments show that our structure not only improve the throughput of the system, but also produces less erase operations to the disk, which makes the working life of the disk much longer.

The rest of the paper is organized as follows. Section 2 reviews some existing B^+ -Tree based methods in MOD and explains why they cannot be applied on SSD directly. Section 3 gives a very naive solution to the problem, then presents the design goals of our system. Section 4 introduces the system framework and discusses about the log clean problem. Section 5 shows the experiment results and Section 6 concludes the paper.

2 Related Work

2.1 B^+ -Tree based methods in MOD

In the moving object database, a large set of moving objects report their geographic information to the system continuously, which is too large to hold in the main memory. Therefore, how to store and retrieve such huge amount of data becomes a serious problem. There are generally two partitioning methods to index objects, object-based partitioning and spatial partitioning. In the spatial partitioning approach, space is partitioned with a grid. Objects are indexed by the cell information they belong to.

The B^x -Tree[7] is a space-partitioned method. It adopts B^+ -Tree for indexing moving objects for the first time. In B^x -Tree, the whole geographic space is partitioned into cells of the same size, and filled with one dimension space filling curve (e.g. Hilbert Curve). Therefore, each cell is assigned by a unique id. Object in the cell uses that id as a part of the key to be indexed on a B^+ -Tree. In B^x -Tree, time is divided into equal length time phases. Each phase has a reference time stamp, and each phase is mapped with a B^+ -Tree. When a new record arrives, the system first decides which phase this data belongs to, then insert the data to the related tree. The inserted record is not intact but has to be calculated to fit the reference time by the object's location and velocity. When the time passes, old trees expire and new trees build. The system always maintains a fix number of trees, and the system performance depends on the trees. Figure 1a shows how the trees work with the phases.

In B^x -Tree, when a range query comes, the system will first decide which trees are related to the query and distribute the query to them. For each tree, the query region is recomputed and expanded based on the statistic information and the reference time stamp. For example, in Figure 1b, several points are stored in tree K by the time stamp t_{ref} . The real value of the points are shown as solid points, while the computed value are shown as void points. If a range query q is executed, q must be expanded by the min and max velocity of the nodes, then the covered points in the expanded query region become the

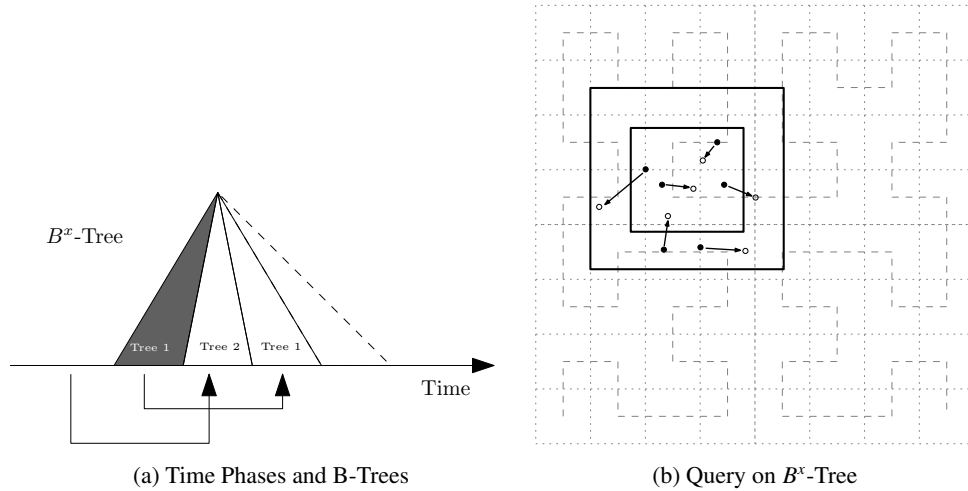


Figure 1: B^x -Tree Example

candidates of the result, waiting for the continuing operations. The B^x -Tree can be well integrated into a relational database, and it is convenient to deploy the system to a distributed environment.

Considering the variation of velocity in moving objects, the B^{dual} -Tree[8] uses space filling curve to fill both the location and velocity dimensions. In that case, objects near by which have the similar velocities are organized into the same cell.

The ST^2B -Tree[9] is another B^+ -Tree based structure, and it is the most efficient B^+ -Tree based structure in MOD to the best of our knowledge. In this paper, the authors noted that, when the data distribution varies with time, all former methods are not applicable any more. To overcome this problem, the system first partition the space based on the data density. Each partition has a different cell granularity. As the time passes, the system tunes the granularity according to the density automatically.

All of the above methods are B^+ -Tree based approaches, and they all produce a large number of random writes to the disk. When the storage system turns to be a flash disk, they not only perform a lower throughput, but also cut down the life of flash device. This offers us the opportunity: if we make the foundational B^+ -Tree work efficiently in a write intensive environment, the above methods can all be applied to flash disk.

2.2 B^+ -Tree on Flash Disk

In computer system, the most commonly used flash device is Solid State Disk (SSD). Through a mapping strategy (hardware and software, e.g. FTL[10]), operating systems access to the device with the same I/O interface as a magnetic disk. Nevertheless, from the view of the operating system, SSD performs quite different from the magnetic disk. Table 1 shows the different I/O patterns between the two types. Comparing with magnetic disk, SSD mainly has the equal access latency to any address. While the sequential write performance is comparable with read, the random write operation performs much worse than the other operations, ten to one hundred times slower in general. In SSD, in-place update for a block is not performed directly. Before that, a bigger unit called erase unit has to be erased[11]. Not only the erasing time is considerable, but also the number of times of the erase operation is limited to

$10^5 \sim 10^6$ for each inside flash chip. The flash disk will wear out if some of its chips become invalid. These characteristics require us to reconsider the external write intensive algorithms, which must take advantage of the good properties in SSD, avoid the random write as much as possible without triggering too many erase operations for a longer life-span.

Table 1: Performance Comparison Between Solid State Disk and Magnetic Disk (IO/s, 4K Page)

Operation	Segate 7200PM SATA Magnetic Disk	OCZ 64GB SSD	Intel Extreme 64GB SSD ¹
Sequential Read	10,180	18,000	55,000
Sequential Write	6,402	7,000	42,500
Random Read	2,560	17,000	35,000
Random Write	2,600	1,100	3,300

To overcome the bad random write performance, the most important philosophy in flash related field is the logging approach. By using this strategy, some of the structures are designed for embedded systems[12, 13], others are for the computer systems[2]. The main idea is to record the alternation of the existing structure instead of in-place update. In [12], the logical information of a B-Tree is maintained in the main memory. On the disk, modifications of the nodes are appended to the log file sequentially. When a read operation is invoked, system has to search in the main memory first to get the related page numbers of the nodes and rebuild them one by one. Therefore, when the write performance is very good, the query performance is so poor that in the extreme case, all the pages on the disk have to be retrieved to rebuild the nodes on the query path. This method is a write optimized approach, without considering the query performance.

The FlashDB[13] is based on [12]. The author claims that the workload of a database system cannot be predicted, and they propose a self tuning approach to let the system adapt to the workload automatically. In FlashDB, pages on the flash have two states: disk mode and log mode. When a page is in disk mode, it acts just like a traditional disk page, from which program could gain from its high read speed. When one page is in log mode, its modification log is appended to the tail of the file, which can prevent it from the bad write performance. An on-line algorithm is proposed to make a decision by which moment one page will change its state. A logical layer of the pages is built upon the system, from which a program can use the disk transparently. The length of each node's log is constricted so that there is a guarantee of the read performance. In this method, the pages are assumed to be independent to each other, so the base structure is not optimized for the appointed query. Since it is designed for the embedded system, its cost model is not suitable for a SSD based system.

An in-page logging approach is proposed in [2]. In this paper, the system only stores one page's log in the erase unit it belongs to. Because the log area is so small for a write intensive environment, the overflow of the log will trigger erase unit operations continuously. What is more, it is not realistic for an operating system to control the details of the erasing strategy and garbage collection through the I/O interface. So it is not a suitable solution for our problem.

¹<http://www.intel.com/design/flash/nand/extreme/index.htm>

3 Design Principles

3.1 Naive Method

For we want to make changes on the B^x -Tree, we first study the performance of B^+ -Tree. In order to avoid the poor random write performance, the very naive thought is to never produce random write to the disk. To achieve this goal, we build a node map table in memory. The table consists of two columns: the node number and the physical address of the node. We use a file cursor to remember the end of the file. When a node needs update, the system just append the node to the file. Then the physical address in the table will be updated by the file cursor. We compare this approach with the conventional method. Figure 2 shows the elapsed time for inserting 10^6 records. The Sequential B-Tree denotes our approach. The Random B-Tree means the B-Tree without modification. From the figure we see that the performance really grows. Because an insert operation of a B-Tree needs a search and a leaf insert, the estimating cost of the Random B-Tree is $R_{rdm} \times H + W_{seq}$, while the Sequential B-Tree is $R_{rdm} \times H + W_{rdm}$. When $\frac{W_{rdm}}{W_{seq}} = 10$, $H = 4$, $W_{seq} = R_{seq}$, the cost ratio is between $2 \sim 3$, which is in accordance with our test (the notations are listed in Table 2). But the data file size is not acceptable, nearly 20X larger than the original. The big file size also demonstrates that during the tree construction, a lot of erase unit operations are triggered.

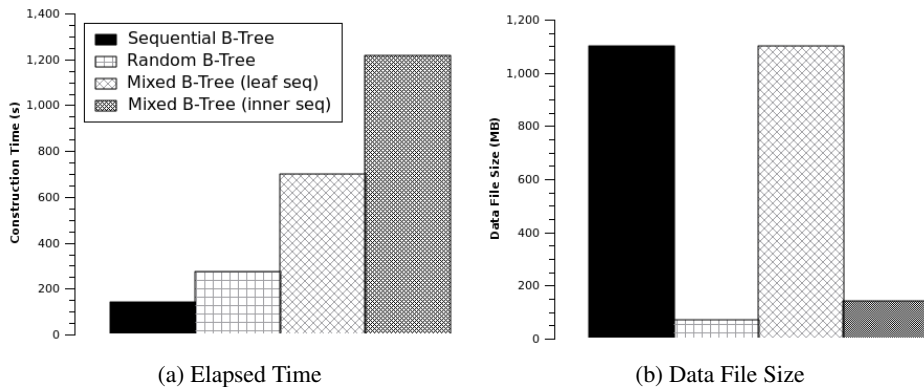


Figure 2: Sequential B-Tree and Random B-Tree

We also do some test on a mixed operation tree. We consider the inner nodes and leaf nodes separately. When inner nodes append to the data file, leaf nodes update in-place, so does the opposite. Both of the performances become so poor that they behave even worse than the Random B-Tree. We believe that this is because the appending operation harass the internal functioning of SSD’s wear leveling mechanism. But from the file size we also notice that in inner node updates are not the main part of the whole writes.

To sum up, we have the following observations from the test: (1)the in-place updates cause a lot of erase operations; (2)main updates take place in the leaf nodes; (3)mixing append and in-pace update together performs worse than either. These observations leads us to the design principles of our system.

3.2 Design Manifesto

Our main assumption is that the storage system consists of two levels: a RAM based main memory and a SSD based external memory. On the basis of the characteristics of flash disk and the experimental observations, the design manifesto is stated as follows.

- Take advantage of the good features of flash disk. The uniform access latency to any address on the disk makes it possible to scatter information all over the disk.
- Produce fewer erase operations. For erase operation not only makes the system pay extra cost, but also shorten the lifetime of the flash device, we should try our best to avoid it.
- Do not mix the append and in-place update together. For one file we only choose either of the operations but not mix them together, in case it may bring a worse performance.
- A guarantee of the system performance. Though the system is designed for a write intensive environment, its query performance should have a theoretical and experimental guarantee.

4 FBX-Tree

In this section, we first introduce the structure and basic query algorithms of the FBX-Tree. Then we raise the log clean problem and give two on-line algorithms to solve it. We use the notations in Table 2 to estimate the I/O cost of the structure.

Table 2: Notations for Cost Analysis

Symbol	Description
R_{seq}	sequential read cost per block
R_{rdm}	random read cost per block
W_{seq}	sequential write cost per block
W_{rdm}	random write cost per block
H	tree height
$Root$	root of the tree
D	leaf order
L_i	log list length of node i

4.1 Index Structure

FBX-Tree mainly use B^x -Tree as its front-end query processing part. By using a space filling curve, record insertions and queries are mapped to several B^+ -Trees to execute. Each logical B^+ -Tree in the system consists of three parts: a Node Translation Table (NTT), a Node Manager and a Buffer Manager.

Node Manager rebuild the nodes and do the maintenance to the NTT. Buffer Manager controls the write and read buffer according to the buffer replacement policy. It also maintains a log buffer to cache

the log information. The NTT is the most important part of the system. It contains metadata of the logical nodes (e.g. node state, entry numbers) and logs (e.g. log list length).

Nodes in the system have three physical states: IN_MEM, BLOCK, LOG. IN_MEM means the node is in the buffer. BLOCK indicates the node occupies one block on the disk. LOG means the node consists of logs spreaded on the disk, while the pointers to the log pages are stored in the NTT. One log entry is formatted as (key, record, node number it belongs to, time stamp, etc). From the logs, one node can be rebuilt easily. The inner node change states between IN_MEM and BLOCK, so that the search operation preforms as good as B-Tree. The leaf node change states between IN_MEM and LOG, in order to exploit the fast random access pattern of flash disk.

The write to the file is always performed as an append operation in FBX-Tree. When an inner node update comes, it would be just a write to the tail and an update in the NTT. When a log page comes, it is also appended to the file, meanwhile add entries to the related nodes' log lists.

To insert one record, the system searches the tree from the root to the parent of the leaf, and do the insertion to the leaf. Since the insertion is buffered by log buffer, and would not be flushed till it's full, the insertion cost will be amortized as $\frac{W_{seq}}{D}$. Therefore, insert one record will cost $R_{rdm} \times (H - 1) + \frac{W_{seq}}{D}$. Comparing with Random B-Tree, the speedup ratio will be nearly 4X. When we turn on the node buffer pool for the system, we can always keep the root node in memory, which leads to a higher ratio.

To search one record, the system has to retrieve the nodes from the root to the leaf. We let L_i denotes the log list length of node i (i is a leaf node), then we get the estimated cost $R_{rdm} \times ((H - 1) + L_i)$. We see that the L_i determines the search performance of the tree. We will discuss this issue in section 4.4.

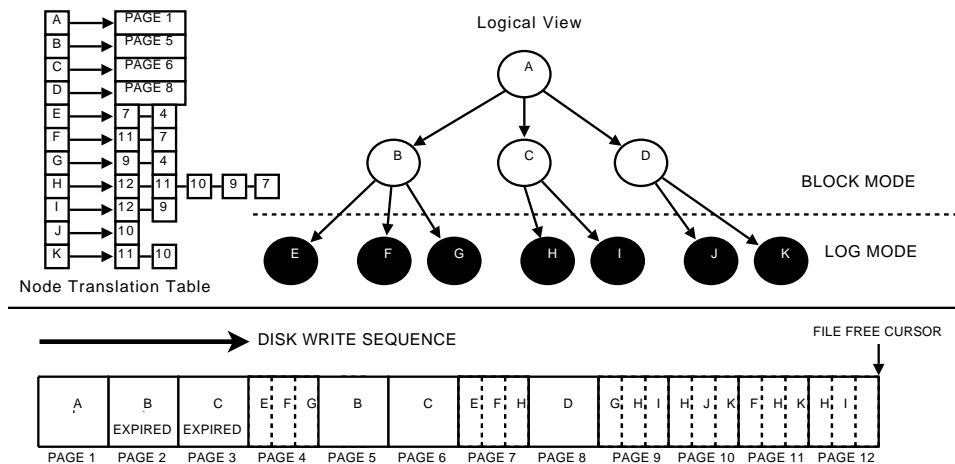


Figure 3: FBX Index Structure

Example. Figure 3 shows a snapshot of one tree in FBX-Tree. Since node A to D are inner nodes, their physical address are stored in the NTT. When an update comes, the old page is abandoned, and a new page is appended to the file. Page 2 and Page 3 are in this case. Node E to F are leaf nodes, so their updates are recorded as logs in the log buffer, then flush to the disk when the buffer is full. Page 7 is made up of 3 log entries, which belong to node E, F and G. When we write this page to the disk, log lists in the NTT are updated as well.

4.2 Buffer Management

Besides node buffer, the system keeps a log buffer in the system. The log buffer is used to cache the log entries to the leaf nodes. Because the log information for one node is scattered on several pages, if we reserve more object's cluster property through the log buffer, the log list would be shorter and the query would be executed more efficiently.

Since the cells in the space are numbered by the space filling curve, the locality of the cells can be evaluated by the cell numbers. In FBX-Tree, we set the log buffer for 1M space, which can contain 2^8 log pages. When a log entry comes to the buffer, the first 8 bit of cell number is used to decide which buffer it will go to.

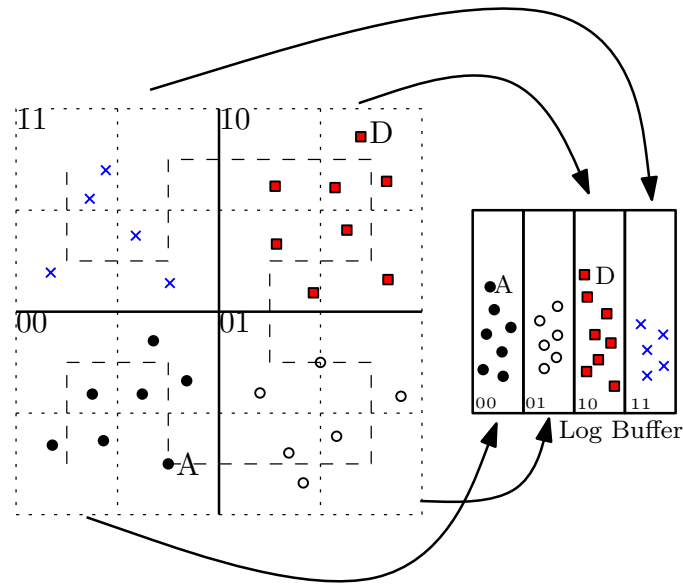


Figure 4: Log Buffer Cluster Property

Example. In Figure 4, we set a 4 page log buffer for one tree. Assume that there are 16 cells in the map. The cells can be parted by the first two bits of their numbers. Then objects in the different parts are arranged to the different buffers. When object A and D are inserted into the map, they go to buffer 00 and buffer 10.

4.3 Query Execution

In FBX-Tree, we use the same logic as B^x -Tree to execute the range query and KNN query. Since the index structure changes, we will make use of the good features of the structure to accelerate the query performances.

Range Query. In B^x -Tree, range query is transformed to the range queries on B^+ -Tree. Using the sibling pointers in the leaf nodes, we can traverse to the related nodes and get the result. If we use this method in our solution, we have to rebuild the leaves one by one, without considering the cluster

property of the logs, and one page may be accessed for many times. In FBX-Tree, we first get several corresponding log lists, then we merge the lists to one, in order to remove duplicate access to pages.

Algorithm 1 Range Query

```

set A;
for all node  $i$  such that node  $i$  in the range do
    get node  $i$ 's log list;
    add the addresses in the list to set  $A$ ;
end for
for all address  $a$  such that  $a$  in set  $A$  do
    read page by  $a$ ;
    add node information to related nodes;
end for
return range result;

```

KNN Query. To deal with the KNN query, we put entry numbers of each node to the NTT. Through that information we can estimate how many node should be expanded and how many pages should have to be accessed. Then using the same strategy as Range Query, we get the results.

4.4 Log Clean Strategy

According to the above statement, leaf node change states between IN_MEM and LOG in the system. There are three operation that can be used to leaf nodes: read, write and clean. The write is performed as a log insert. The read is performed as node rebuild, which is to read all the related pages on the basis of the NTT log list. When a clean operation is executed, system first rebuild the node, then reset the log list length to one, meanwhile put the page to the node buffer pool. To assure the query performance, the in memory log list cannot be too long. However, considering the life-span of the flash disk and the footprint of the data file, the clean operation cannot be triggered too often. So the decision-making in log clean is a key problem to guarantee the query performance of the system, which is a trade-off between efficiency and the disk lifetime.

Example. Given an operation sequence for one node: $\{w, r, w, r, r, w, r, w, w, r, r, w, r\}, c$. w represents write, c is short for clean, and r represents read. The overall cost consists of the three types of operations. Since the write operation is atomic, it remains the same when clean takes place. Figure 5a and 5b represents two different cost when we do the clean operation at different time. When the operation sequence is determined, we can calculate the optimal time to do the clean operation. Figure 5c shows all the possible savings of the operation clean, and the best time to do the clean operation is after the fifth read. The operation sequence, however, is unpredicted when the system starts, so the this problem needs an on line algorithm to solve. Firstly, we give the formal definition to the problem.

Definition 1 Given an operation sequence: $O = (O_1, O_2, \dots, O_n)$, where $O_i \in \{read, write\}$, and a

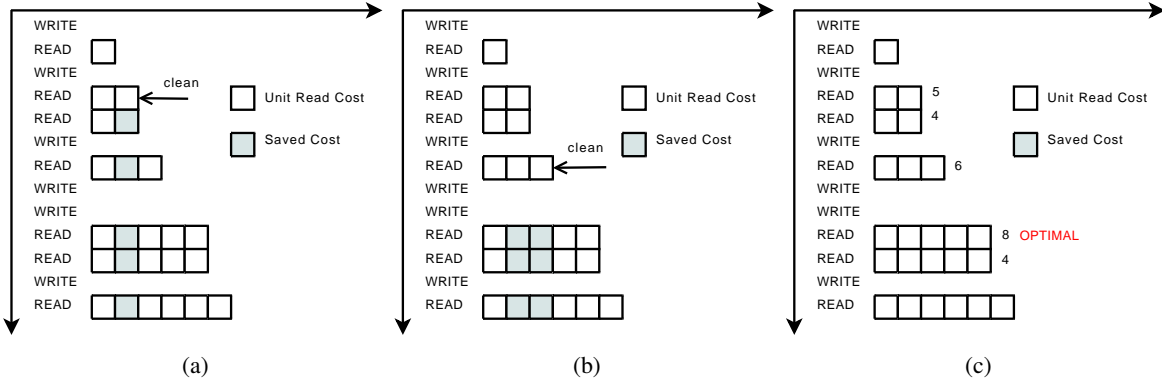


Figure 5: log clean example

set of clean operations C , where $|C| = m$. Insert C to O , and get a new operation sequence $P = (P_1, P_2, \dots, P_{n+m})$, where $P_i \in \{\text{read}, \text{write}, \text{clean}\}$. Cost (i) denotes the cost of the i th operation. Our goal is to minimize the overall operation cost $\sum_{i=1}^{m+n} \text{Cost}(i)$. Here the cost of a write operation is a constant number; the cost of read and clean operations depend on the currently length of the log list. For in our solution the write cost for one record is small, in the following proof we ignore the write cost for simplicity.

We have several observations for this problem.

Observation 1 The optimal clean operation must take place just after a read operation.

If not, there would be two choice for the clean operation: (1) During the read operation; (2) After the write operation. For (1), the clean can be delayed to the end of the read operation, and the overall cost will be smaller. For (2), the clean can be moved to the next nearest read operation tail, the overall cost will be smaller.

Observation 2 The optimal clean operation must take place when the read operation is after a write operation.

If not, it means there are one or more read operations before this read, then the clean operation can be shifted to the upper read operation, the overall cost will be smaller.

Observation 3 If the sequence has i reads and j writes, and ends with a read operation, the minimum cost is no less than $i + j - 1$ (1 cost unit for 1 read block operation).

For the sequence ends with a read, all logs must be read at least once, which makes the cost j . Besides read the new added page, the system would take at least 1 cost at each read. Since we would compute the first added record twice, we remove 1 from the result and get $i + j - 1$.

Naive Method. In the previous work, the log clean problem is mainly despised. They use a counter to count the read operations for the node. When it goes bigger than a given threshold, the clean operation will take place. When we let the threshold to be K , it is easy to show that the competitive ratio of this

method is K , which is far away from the optimal solution. So we raise two algorithms for this problem, both of which have a better competitive ratio.

The Longer Than Next Algorithm. This algorithm's idea is mainly from the naive solution for Ski Rental Problem[14]. The accumulated cost increases by the log length then. When the cost is larger than the present log list length. If the cost is larger, then the clean operation is triggered. This algorithm is sensitive to the workload. When a series of read operations come into the system, the clean function is called frequently. In a write intensive environment, the algorithm would work well.

Algorithm 2 The Longer Than Next Algorithm

```

acc = 0;
if operation == write then
    insert the entry to the log buffer;
else if operation == read then
    rebuild the node;
    if acc >  $L_i$  then
         $L_i = 1$ ;
        acc = 0;
        put node i to the buffer pool;
    end if;
    acc = acc +  $L_i$ ;
end if

```

Theorem 1 *The Longer Than Next Algorithm's competitive ratio is 3.*

Considering about a read sequence (R_1, R_2, \dots, R_n) , in which R_i denotes the cost for the i th operation, and one clean is triggered at R_n . According to the algorithm, $\sum_{i=1}^{n-1} R_i > R_n$, $\sum_{i=1}^{n-2} R_i < R_{n-1}$. Then we get $\sum_{i=1}^{n-1} R_i = \sum_{i=1}^{n-2} R_i + R_{n-1} < 2R_{n-1}$. For $R_{n-1} < R_n$, we finally get the cost of the algorithm $\sum_{i=1}^n R_i < 3R_n$. From Observation 3, we have that the cost of the operation sequence is no less than R_n . Therefore, the competitive ratio is 3.

Accumulate and Clean Algorithm. In this algorithm, we also define a constant threshold K . When a read is performed to the node, like the Longer Than Next Algorithm, the accumulated cost will increase by the present log length. When the cost is bigger than the threshold, it would trigger a clean operation. This algorithm's competitive ratio is $\frac{\sqrt{K}}{2} + \frac{1}{3}$.

Theorem 2 *The Accumulate and Clean Algorithm's competitive is $\frac{\sqrt{K}}{2} + \frac{1}{3}$, when the constant in the algorithm is K .*

Assume that for an operation sequence S , we do N clean operations. By the algorithm we can get the overall cost is NK . From Observation 3, we know there is a lower bound for the optimal cost. We can

Algorithm 3 The Accumulate and Clean Algorithm

```
acc = 0;
if operation == write then
    insert the entry to the log buffer;
else if operation == read then
    rebuild the node;
    acc = acc + Li;
    if acc > threshold then
        Li = 1;
        acc = 0;
        put node i to the buffer pool;
    end if;
end if
```

construct S to achieve the bound meanwhile take N clean operations for both of the optimal and on-line methods. The sequence is as the form in Figure 6. We can also show that when each operation block is organized as a square, the optimal cost reaches the minimum, which is $2\sqrt{K} - 1$. Therefore the overall optimal cost is $N(2\sqrt{K} - 1)$, and we get the competitive ratio $\frac{K}{2\sqrt{K}-1} = \frac{\sqrt{K}}{2} + \frac{\sqrt{K}}{4\sqrt{K}-2} = \frac{\sqrt{K}}{2} + \frac{1}{4-\frac{2}{\sqrt{K}}} < \frac{\sqrt{K}}{2} + \frac{1}{3}$.

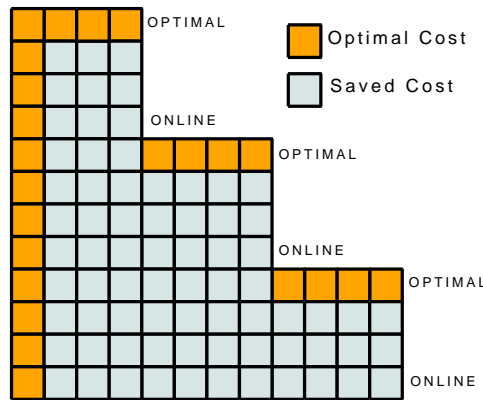


Figure 6: Constructed Optimal Cost

5 Experiment

Our experiments are implemented in C++ and running on a PC with an Intel P4 2.8G CPU, 512M main memory. The SSD's capacity is 64G, from OCZ Technology^{footnote{\url{http://www.ocztechnology.com/}}}. The NAND chip type in the SSD is Multi-Level Cell (MLC), which is in common use in mainstream SSD. When we use the SSD on a Linux system, it would halt for no reason. So we run the experiments on Windows XP SP3. In order to reduce the effect of the operating system, we close the system buffer and use the disk as a raw device. One entry of the index contains a key (4 bits) and a record (20 bits).

We set the page size to 4KB. The order of the inner node is 500, while the order of the leaf node is 100. The page replacement algorithm for the buffer pool is LRU.

Metadata in memory. The system maintains the metadata of each node in NTT. It contains the node number (32 bit), the node state (16 bit), the read count (32 bit), and the address of the node (32 bit). For a 1G data file, the footprint in memory will be $\frac{1000000}{4} \times (4 + 2 + 4 + 4) = 3.5M$, which is a reasonable size for a standard compute system.

5.1 Comparison between FBX-Tree and B^+ -Tree

To show how unsuitable a B-Tree could be on the flash disk, we run the original B-Tree and FBX-Tree on the SSD and compare their performances. We continuously insert 10^6 records to the tree then search 10^6 records from it. From Figure 7, we can see that the insertion performance of FBX-Tree is much better than B-Tree, about 20X faster in general. The trend of the B-Tree insertion also shows that when the record number continues to grow, the performance would be even worse. Due to the fast random read speed of SSD, the read performance of B-Tree shows big potential, but FBX-Tree also keeps an acceptable behavior, mainly 3 times slower as in B-Tree.

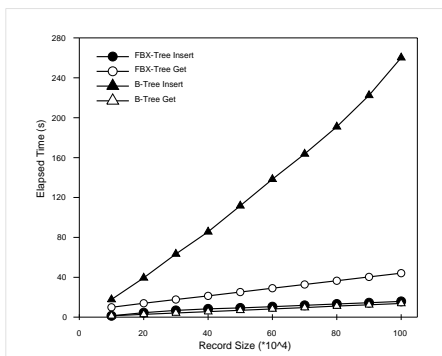


Figure 7: Insertion and Search

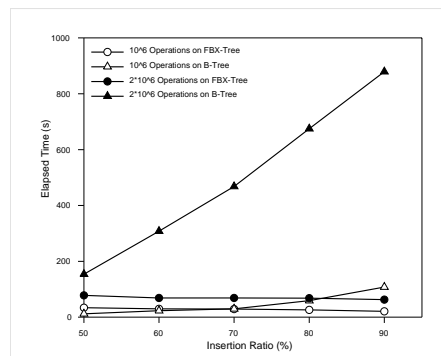


Figure 8: Mixed Workload

To simulate a real workload, we generate a series of mixing operation test files to test the performance of FBX-Tree and B-Tree. For we use FBX-Tree in a write intensive environment, the insertion ratio of the operations changes from 50% to 90%, and the record size is 10^6 and 2×10^6 . From Figure 8 we find that the FBX-Tree is not sensitive to the workload. It performs steadily to every ratio. But the B-Tree's performance is restricted by the write operations to the tree. This is more obvious when the record size is 2×10^6 .

5.2 Buffer Issues

Since the FBX-Tree needs NTT to store the metadata and log list in memory, one question is raised: how about use the same footprint to make a larger buffer for B-Tree? Our experiment shows that in a write intensive workload, a big buffer (8M) does not help to change the performance.

We also test about the affection of the log buffer size to the system performance. For a larger log buffer reserves more locality for the objects, it will lead to a shorter log list in memory, and objects in

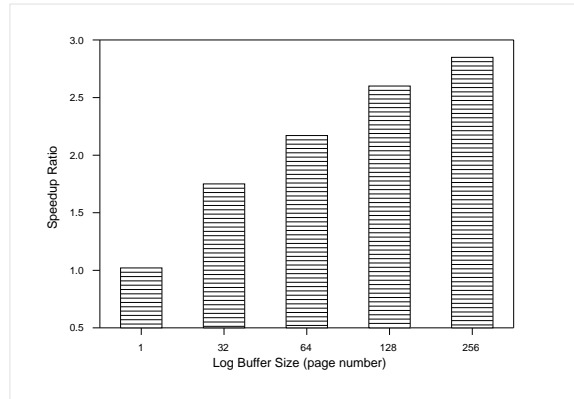


Figure 9: Log Buffer Size and Speedup Ratio

a log page are more likely to be retrieved when a range query comes. Figure 9 shows that by using our log buffer management and query execution methods, the query can perform as 3 times better than the original.

5.3 Log Clean Strategy

Besides analysis, we experiment on the three algorithms of our log clean strategy. We test the three strategies in a mixing workload. The result is shown in Figure 10. We notice that the naive method is not

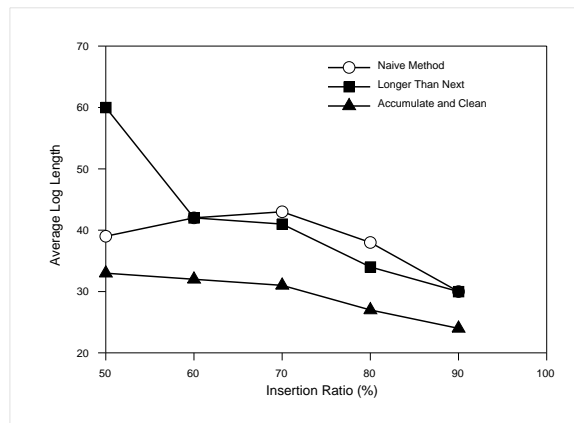


Figure 10: Log Clean Strategy Comparison

as bad as we imagine. But according to the competitive analysis, we can design a workload to make this method slower. Other than that, the average log length for this method is longer, which means to occupy more space in memory. The Longer than Next Algorithm performs well when the workload is write intensive, but it goes bad quickly when there are more queries. The Accumulate and Clean Algorithm performs very good in our experiment. For each work load, it is always the best one.

Since in the Accumulate and Clean Algorithm we have to set a constant as the threshold, we also do some test on how big the number should be. From Figure 11 we see that for each workload a constant between 60 ~ 70 remains at the bottom, which is our choice for the system.

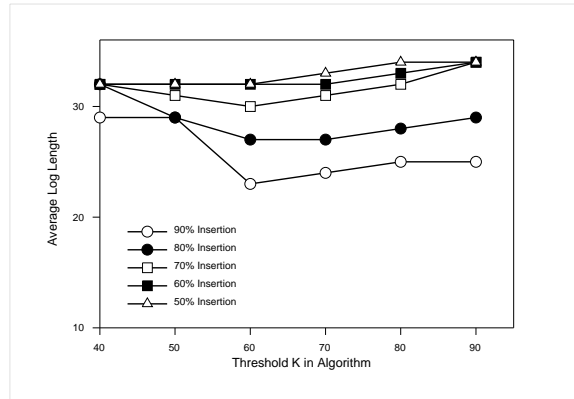


Figure 11: Selection for Threshold

6 Conclusion

In this paper, we give a moving objects indexing method for flash disk. Two on-line algorithms are proposed to solve the log clean problem in the method. Extensive experiments and analysis show that our solution maintains the balance among the throughput, footprint, device life and query performance.

In the future work, we will investigate the object partitioning method on flash disk, and we want to take advantage of the FBX-Tree to answer the uncertain queries incrementally as well.

References

- [1] Justin Mann. Google to utilize intel ssds in servers. Website, 2008. www.techspot.com/news/30003-Google-to-utilize-Intel-SSDs-in-servers.html.
- [2] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, New York, NY, USA, 2007. ACM.
- [3] S. Nath and P.B. Gibbons. Online maintenance of very large random samples on flash storage. *Proceedings of the VLDB Endowment archive*, 1(1):970–983, 2008.
- [4] I. Koltsidas and S.D. Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment archive*, 1(1):514–525, 2008.
- [5] M.A. Shah, S. Harizopoulos, J.L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *Proceedings of the 4th international workshop on Data management on new hardware*, pages 17–24. ACM New York, NY, USA, 2008.
- [6] 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 *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM New York, NY, USA, 2008.

- [7] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b^+ -tree based indexing of moving objects. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 768–779. VLDB Endowment, 2004.
- [8] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. The b^{dual} -tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal*, 17(3):379–400, 2008.
- [9] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. St^2b -tree: a self-tunable spatio-temporal b^+ -tree index for moving objects. In *SIGMOD Conference*, pages 29–42, 2008.
- [10] Intel-Corporation. Understanding the flash translation layer (ftl) specification. Website, 1998. www.embeddedfreebsd.org/Documents/Intel-FTL.pdf.
- [11] Marcus Schneider. Ssds vs. hdds: Ten considerations. Website, 2009. www.infostor.com/index/articles/display/6131087070/s-articles/s-infostor/s-volume-13/s-Issue_4/s-features/s-SSDs_vs_HDDs_Ten_considerations.html.
- [12] C.H. WU, L.P. CHANG, and T.W. KUO. An efficient b-tree layer for flash-memory storage systems. *Lecture notes in computer science*, 2968:409–430, 2004.
- [13] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, New York, NY, USA, 2007. ACM.
- [14] Maurice Queyranne. An introduction to competitive analysis for online optimization. Lecture note, 2002. http://www.cam.wits.ac.za/~mali/Online_Brown-Bag_Slides.pdf.