

A New Dynamic Hash Index for Flash-based Storage*

Li Xiang^{1,2}, Da Zhou^{1,2}, Xiaofeng Meng^{1,2}

¹ School of Information, Renmin University of China

² Key Laboratory of Data Engineering and Knowledge Engineering, MOE

{xiangli, cadizhou, xfmeng}@ruc.edu.cn

Abstract Compared with traditional magnetic disks, Flash memory has many advantages and has been used as external storage media for a wide spectrum of electronic devices (such as PDA, MP3, Digital Camera and Mobile Phone) in recent years. As the capacity increases and price drops, it looks like a perfect alternative for magnetic disks. However, due to hardware limitations of flash memory, techniques including storage subsystem and indexing originally designed for magnetic disks can not run smoothly in a flash memory without any modification. In this paper we explore problems of indexing flash-resided data and present a new dynamical hash index for flash memory in two schemas. The analysis and experimental results validate the efficiency of our design.

I. INTRODUCTION

Flash memory is a type of non-volatile storage media which has been used in a wide spectrum of computing devices such as PDA, MP3 and Mobile Phone. As the capacity increases and price drops stably these years, flash memory gradually becomes a perfect alternative for traditional magnetic disks. One of the main electronic manufacturers of the world has launched a notebook computer that is equipped with a flash disk in stead of hard disk in 2006 [1]. The capacity of the flash disk doubles every year [2], which is faster than the Moore's law. It is reported that flash chip with 128G has been released into the market at the end of last year. As a result of the widely usage of flash memory, the data stored in this type of media is increasing sharply too. Then how to manage flash-resided data becomes an important problem.

Compared with traditional magnetic disk, Flash has many advantages, say, shock resistant, faster access speed, power saving, smaller size, lighter weight and less noise. So flash outperforms low-end magnetic disk under many circumstances. However, due to some other limits of flash (e.g. Erase-Before-Write and different Read/Write speed) existing software for magnetic disk could not yield the best attainable performance in flash. Many techniques (e.g. storage and indexing techniques) should be redesigned.

To address these limits, many flash-specific techniques have been developed. FTL [3] is a proper software layer which makes flash appears to upper layers like a magnetic disk and conventional disk-based applications can run on it without any modification. Another class of schemas (called native flash file systems) is designed to explore the unique

characteristics of flash memory to achieve the best attainable performance. JFFS [4], YFFS [5] and TrueFFS [6] are of this type. Native flash file systems borrow idea from LFS [7], and it treats the whole flash memory as a large log. Any file system modification is appended to the log. Reference [8] presents a new design called IPL for flash memory. IPL can be elegantly extended to support transactional database recovery.

How to index data in flash memory is another interesting problem. Hashing and tree are two main basic indexing techniques. Several recent works address indexing issue by adapting B⁺ Tree like structures to NAND Flash [9, 10]. A tree always requires several accesses to find a record, while hashing index requires about one on average. As the hardware platform moves from magnetic disk to flash memory, the performance of tree index deteriorates dramatically because that it is an additional costly operation to maintain the hierarchical structure of the tree index itself. The situation is worse considering the flash's Erase-Before-Write characteristic. Under this circumstance, we believe hashing, whose structure is very simple and the cost of maintaining the structure of the index itself is much lower than tree, is the proper technique to organize files dynamically in flash.

Reference [11] devised a hash index, which is called MicroHash, for sensor network. However, MicroHash is so special that it could not be used in other circumstances. It is not an indexing structure for general use. In this paper we design a new general dynamic hash index which meets the hardware limitations of flash memory. We show experimentally that our design can yield considerable performance benefit over traditional design for magnetic disks. What's more, our design can be elegantly extended to support turning performance dynamically between search and update.

We summarize our contributions as follows: *Firstly*, we discuss the detailed characteristics of flash memory and the challenges of running traditional indexing structures in flash memory. *Secondly*, we design a dynamic hash index which meets the limitations of flash. *Thirdly* our analysis and experiments validate the efficiency of our design.

The rest of this paper is organized as follows: in section II we discuss the characteristics of flash memory and give preliminary knowledge of dynamic hash index. In section III we present the basic algorithms of our design. We analyse the performance of Lazy-Split schema and Eager-Split schema, and give the tuning method in section IV. Section V shows the experiment results. In section VI we conclude the whole paper.

*This research was partially supported by the grants of Natural Science Foundation of China (No.60573091); China 863 High-Tech Program (No.2007AA01Z155); China National Basic Research and Development Program's Semantic Grid Project (No.2003CB317000); Program for New Century Excellent Talents in University (NCET).

II. FLASH CHARACTERISTICS AND DYNAMIC HASH INDEX

Flash is an electronic device which has no mechanical latency and this is very different from traditional magnetic disk. We discuss the key characteristics of flash and their impacts on developing new techniques and then give preliminary knowledge background of dynamic index techniques in this section. TABLE I shows the main parameters of flash memory and magnetic disk [12].

TABLE I
PARAMETERS OF FLASH AND DISK

| | Samsung's 64G Flash-SSD | 80G HDD |
|--------------------------------|-------------------------------|----------------|
| Read/Write speed (MB/s) | R:64 W:45 | R:15 W:7 |
| Weight (g) | 15 | 61 |
| Power Consumption (watt) | O:0.5 I:0.1 | O:1.5 I:1.5 |

A. Flash memory Characteristics

Flash memory has a hierarchical structure [13]. A flash chip is consisted of many blocks (typically 128 Kbytes each block), and a block is consisted of many pages (typically 2 Kbytes each page). The Read/Write granularity of flash memory is page, and it requires erasing operation before overwriting. The erasing granularity is block, that is to say, it requires erasing the whole block containing the page if you want to update data item in the page. Reading (typically 100 μ s/page) is the fastest operation. Writing (typically 200 μ s/page) is much more time-consuming than reading, and erasing (typically 2ms/block) is the most costly operation among all the three operations.

In this section, we describe the key characteristics of flash memory that distinguish itself from magnetic disk, and discuss their impacts on designing new index structures.

1) *Erase-Before-Write*: In magnetic disk, we assume that the update of a data item can be overwritten in the same physical address where the older version inhabits. We call this update pattern in magnetic disk *in-place update*. But this assumption doesn't work in flash memory. In a flash memory, the update can't be written to the same physical address until the whole block containing the data item has been erased. We call this characteristic of flash memory *Erase-Before-Write*. Considering that erasing the whole block is a time-consuming operation, the cost of update is too big to afford and the system's performance degrade significantly when meeting update-intensive workloads. Consequently, it requires us to rethink the index techniques running well on magnetic disks and find ways to reduce or avoid update to maintain reasonable performance in flash.

2) *Asymmetric Read/Write Speed*: Another important characteristic of flash is different Read/Write speed, while in a magnetic disk they are the same. The reason is simple that flash is an electronic device and it takes longer time to discharge (when writing) than to read the status (when reading). Obviously, all the existing works of index for magnetic disk treat the read speed as the same as writing,

so there are still a lot of work to do when taking this characteristic of flash memory into consideration. One possible way to upgrade the overall performance of indexes in flash is trying to reduce writing times even in the expense of more reads. Another advantage of reducing writing times is that the erasing times are reduced indirectly. The performance may be upgraded much considering that writing and erasing are more costly operations than reading in a flash memory.

3) *Finite Erasure Times*: The erasing granularity of a flash memory is block, and the erasing times of a block are not infinite. Typically, a block can be erased 100,000 times before its status becomes uncertain. This limitation emphasizes the necessity of distributing writing operations evenly on the whole chip and reducing erasing times by processing update subtly. One possible way is to reduce intermediate results written to flash.

B. Dynamic Hash Index

Main idea of Dynamic has is increasing or shrinking the address space and altering the hash function dynamically in the course of insertion and deletion. In fact, hashing indexes comes in two flavors, Extendible Hash [14] and Linear Hash [15]. The main difference between Extensible Hash and Linear Hash is how many additional buckets should be appended when it comes to increase the number of buckets. Extensible Hash doubles its buckets every time, while Linear Hash increases one bucket at one time.

Because Extensible Hash doubles its buckets every time, the efficiency of the space utilization is very low and the response time is very long in the worst situation [16]. We believe Linear Hash is more proper to be used to index data in flash. In the next section we discuss Linear Hash index in details.

C. Linear Hash Index

Linear Hash index is consisted of three components, that is, *hash function*, *bucket list* and *split control method*. The first two parts of Linear Hash is dynamically adjusted according to the number of records it indexed. If we want to find a record, we compute the bucket number by the hash function, then, find the actual address of the corresponding bucket containing the record in the bucket list. Finally, we scan that bucket. In the course of insertion/deletion, it may be necessary to increase/decrease the number of buckets. We don't consider shrinking because the cost is too high and it isn't worthy to do so especially in flash. The component of determining when/how to increase buckets is called the *split control method*. TABLE II shows the tokens and process procedures that will be used in the remains of this paper.

To be simply, we use the following setting to show the mechanism of Linear Hash. It can be extended easily.

- we use the value of the last i bits of key K_j to be the hash value of record R_j (K_j is key of R_j)
- if $h(K_j) \geq b$, we convert the highest bit of $h(K_j)$ to zero, and let the new value $h(h(K_j), \text{Bit}(h(K_j)) - 1)$ to be the hash value of record R_j
- if the load factor, lf , exceeds the upper bound, I , we increase the bucket number and append a new bucket to the bucket list

Figure 1.a shows the status of Linear Hash after insertion of three records, 8, 7 and 3. Figure 1.b shows the status after inserting 10. Reference [16] discuss the algorithms of linear hash in details.

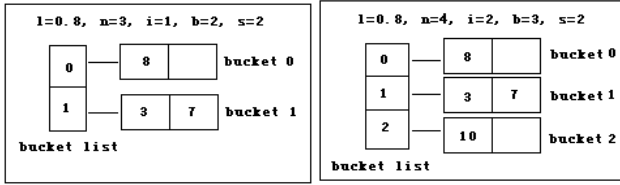


Figure 1.a

Figure 1.b

III. OUR DESIGN

The Linear Hash index depicted in section II is originally designed for magnetic disk (referenced as *Disk Hash* below). Disk Hash is not suitable for direct use in flash, and the problem arises from the Erase-Before-Write limitation. The challenges of designing index in flash are of two types. One is how to deal with the update/deletion of indexed data item. The other is how to dynamically manage the index structure itself.

TABLE II
TOKENS AND PROCESS PROCEDURES

| Token | Description |
|--------------------|----------------------------------------------------------------------------------------|
| R_i | record j |
| K_i | key of record j |
| n | total record number |
| b | total bucket number |
| s | bucket size (records per bucket) |
| lf | load factor, $lf = (n/b)/s$ |
| Query ratio | search operations/total operations |
| l | upper bound of load factor |
| $h(K_i)$ | hash function |
| i | bit number of hash value in use |
| $h(x, j)$ | get the last j bits of $h(x)$ |
| $Bkt[j]$ | bucket j |
| $Bit(x)$ | compute the bit number of binary value of x (e.g. $Bit(7)=3, Bit(1)=1$) |
| $split(j)$ | split bucket j , insert the records to the corresponding bucket properly |
| $ist(k, Bkt[j])$ | insert k to bucket j , if bucket j is full, add a overflow page before insertion |
| $scan(k, Bkt[j])$ | scan bucket j to look for k , return the records whose key equal k |

Consequently, our first design manifesto here is to insert a minus record to avoid deleting records in-place. Of course, we should get rid of the minus records and their corresponding records before return the results of a searching operation. Our second design manifesto here is to reduce writing times in the expense of more reads.

In this section we present our design in two flavors: *Eager-Split* schema and *Lazy-Split* schema. Both of our designs have implemented the first design manifesto, but only the *Lazy-Split* schema implements the second one. The main difference of the two schemas is that in *Eager-Split* schema there is a bucket that should be split when a new bucket is appended to the bucket list according to the Split Control Method. In the *Lazy-Split* schema, the splitting is delayed and processed in batches. We append a bucket to the bucket list directly and the splitting operation

is controlled by an additional procedure.

In the remains of this section, we present the basic algorithms of our design firstly. Then we give algorithms of *Eager-Split* and *Lazy-Split* schema. The setting is the same as in section II if not pointed out specially. Noteworthily, in our design a record $R_i(k_i, p_i, \text{mark})$ is consisted of three fields: k_i is the key of record R_i , p_i is the pointer of record R_i . If record i is a minus record mark is 1, otherwise, mark is 0. For example, the minus record of $(353, 13430, 0)$ is $(353, 13430, 1)$.

A. Eager-Split Schema

In *Eager-Split* schema, the algorithms are almost the same as *Disk Hash* index. We give the pseudo-codes of insertion, deletion and search below.

Insertion: The algorithm of insertion of *Eager-Split* schema is listed in Algorithm 1. In the course of insertion, if the load factor exceeds the threshold l , we append a new bucket to bucket list, and then update other parameters properly. Before inserting the record we trigger the splitting operation to split a bucket and reinsert records in it correctly. At last we insert the record.

Algorithm 1

Parameters:

k , the key to be inserted

- (1) $n++$;
- (2) **If** $lf > l$
- (3) $b++$;
- (4) $i = \text{Bit}(b)$;
- (5) add $Bkt[b-1]$ to bucket list;
- (6) $split(h(b-1, \text{Bit}(b-1)-1))$;
- (7) $tmp = h(k, i)$;
- (8) **If** $h(k, i) \geq b$
- (9) $tmp = h(h(k, i), \text{Bit}(h(k, i))-1)$;
- (10) $ist(k, Bkt[tmp])$;

Search: The algorithm of search of *Eager-Split* schema is as the same as *Disk Hash*'s. Firstly, we compute the bucket number by the hash function, then, we find the actual address of the corresponding bucket containing the record. Finally, we scan the bucket to find the record. Of course we should get rid of the minus records before return the results. Pseudo-code of search is listed in Algorithm 2 below.

Algorithm 2

Parameters:

k , the key to be searched

Ra , the result array

- (1) $tmp = h(k, i)$;
- (2) **If** $h(k, i) \geq b$
- (3) $tmp = h(h(k, i), \text{Bit}(h(k, i))-1)$;
- (4) scan bucket $Bkt[tmp]$, add the matched record to Ra ;
- (5) delete minus records and their corresponding records from Ra
- (6) **Return** Ra

Deletion: One point of the algorithm of deletion that worthy mentioned is that if the record existed, we don't delete record from its bucket directly, but insert its corresponding minus record in order to avoid erasing the whole block. For example, in order to delete $R(100, 13342, 0)$, we insert its minus record $R(100, 13342, 1)$

to hash index. Algorithm 3 shows the process procedure of deletion.

Algorithm 3

Parameters:

R, the record to be deleted

R', the minus record of R

Ra, the result array of search k

-
- (1) **Ra** = search (k);
 - (2) **For** every record **r** in **Ra**
 - (3) **If** the key and pointer of **r** equal the key and pointer of **R** **And** **r** isn't a minus record
 - (4) insert record **R'**; **break**
-

B. Lazy-Split Schema

Considering the asymmetric read/write speed of flash memory, we believe that it upgrade overall performance if we can improve the writing efficiency even in the expense of more reads. Because splitting is the most costly operation in linear hash, we develop Lazy-Split schema to process splitting in batches. The main idea here is reducing splitting times to reduce intermediate result written to flash. Therefore, it reduces writing times and erasing times, and upgrades the overall performance and prolongs the lifetime of flash in the same time.

Insertion: The insertion procedure of Lazy-Split schema is more or less the same as Eager-Split schema. The only difference is that it doesn't trigger the splitting when there is a new bucket appended to the bucket list. The splitting is delayed and will be processed in batches. Consequentially, the response of insertion of Lazy-Split is quick at most of the time. When and how the splitting operation is triggered is controlled by a new separate operation called splitting, which will be showed later.

Splitting: The splitting operation is the most time-consuming procedure in linear hash. In Lazy-Split schema we redesign it to reduce splitting times, consequentially, to reduce intermediate result written to flash and balancing the performance of updating and searching. Detailed analysis of Lazy-splitting efficiency and tuning method will be presented in section IV.

We introduce a parameter *Split-cursor* here to denote which bucket the splitting operation will end at, that is to say, the splitting operation will split the buckets from 0 to Split-cursor (Obviously Split-cursor is between 0 and the total bucket number **b**). This parameter is very important in the tuning procedure as we will see later. We list the pseudo-code of lazy splitting operation simply in algorithm 4 below. In algorithm 4, there are many choices in determining (Step 1) whether it is necessary to split. We can assume here that splitting is evoked at the moment when searching efficiency is not tolerable any more or when update is inessential. More details about the proper tuning time would be given in section IV.

Algorithm 4

Parameters:

Split-cursor, the last bucket number to be split

-
- (1) determine whether necessary to split
 - (2) **If** it is necessary now
 - (3) tmp = 0;
 - (4) Split-cursor = b-1;
 - (5) **While**(tmp < H(Split-cursor))
-

(6) split (tmp);

In step 5, **H()** is a function used to compute the max number of the bucket that should be split. Its detailed analysis is presented in section IV (proposition 2).

Search: With the introduction of Split-cursor, the buckets of Lazy Hash are divided into two categories: buckets from Bkt [0] to Bkt[Split-cursor], and buckets from Bkt[Split-cursor +1] to Bkt[b-1]. Consequently, in the course of searching key k, we should take it into consideration. If hash (k) is in range of 0 and Split-cursor, we scan the corresponding bucket Bkt[hash(k)] and return the results. If hash (k) is in range of (Split-cursor+1) and (b-1), the situation is a little more complex, because key k may in the buckets that aren't split yet. We should search every bucket may containing key k that exceeds Split-cursor until the target bucket to be searched drops into the first category that has already been split. The algorithm of searching is presented in algorithm 5.

Algorithm 5

Parameters:

k, the key to be searched

Ra, the result array

-
- (1) tmp = h(k, i);
 - (2) **If** h(k, i) >= b
 - (3) tmp = h(h(k,i), Bit(h(k, i))-1);
 - (4) **While**(tmp > Split-cursor)
 - (5) scan bucket Bkt[tmp], add the matched record to **Ra**;
 - (6) tmp = h(h(k,i), Bit(tmp)-1);
 - (7) scan bucket Bkt[tmp], add the matched record to **Ra**;
 - (8) delete minus records and their corresponding records from **Ra**;
 - (9) **Return Ra**;
-

Deletion: Lazy Split schema processes insertions by inserting minus records as Eager Split does (Algorithm 3).

IV. PERFORMANCE ANALYSIS

In this section we discuss the performance of lazy hash in details and give the idea of tuning index performance.

A. Efficiency of Lazy-Split Schema

In Disk Hash and Eager-Split a bucket is split whenever a new bucket is appended to the bucket list. We therefore conclude that if there are 2^i ($i = \log_2 b$) buckets it should have split $2^i - 1$ times. In Lazy-split splitting is delayed and processed in batches. If we let Split-Cursor equals the total bucket number **b**, about half of the total buckets should be split in splitting. We therefore get the first proposition.

Proposition 1:

$$\begin{aligned} \text{Splitting Ratio} &= \frac{\text{SplitTimesOfEager}}{\text{SplitTimesOfLazy}} \\ &= \frac{2^i - 1}{2^{i-1}} \approx 200\% \end{aligned}$$

TABLE III shows the footprint of splitting. What will happen if we let the Split-cursor wandering between 0 and b in the Lazy-Split schema? We find the assertion that half of the buckets would be split is not accurate, and it could be improved to get a much better result. More accurately, in Lazy-split schema if Split-cursor wandering between 0 and total bucket number **b** we get the following result. In advance we assume:

$$f(x) = x - 2^{\text{Bit}(x)-1}$$

the max number of the bucket that should be split when Split-cursor equals x can be denote in Proposition 2

Proposition 2:

$$H(x) = \text{Max} \{f(x), f(2^{\text{Bit}(x)-1}-1)\}$$

For example, if Split-cursor equals 4, the max number of the bucket should be split is 1, that is to say, bucket 0 and bucket 1 should be split in the splitting operation. If Split-cursor equals 23, the max number of the bucket should be split is 7, that is to say, bucket 0 to bucket 7 should be split in splitting. There are 8 buckets should be split. It is much less than half of 24. Figure 2 gives an intuitive impression of the results.

TABLE III
SPLITTING SITUATION

| New Bucket (decimal) | New Bucket (binary) | Split bucket | Max bucket no. has been split |
|----------------------|---------------------|--------------|-------------------------------|
| 0 | 0000 | | |
| 1 | 0001 | | |
| 2 | 0010 | 0 | 0 |
| 3 | 0011 | 1 | 1 |
| 4 | 0100 | 0 | 1 |
| 5 | 0101 | 1 | 1 |
| 6 | 0110 | 2 | 2 |
| 7 | 0111 | 3 | 3 |
| 8 | 1000 | 0 | 3 |
| 9 | 1001 | 1 | 3 |
| 10 | 1010 | 2 | 3 |
| 11 | 1011 | 3 | 3 |
| 12 | 1100 | 4 | 4 |
| 13 | 1101 | 5 | 5 |
| 14 | 1110 | 6 | 6 |
| 15 | 1111 | 7 | 7 |
| 16 | 10000 | 0 | 7 |
| 17 | 10001 | 1 | 7 |
| ... | ... | ... | ... |

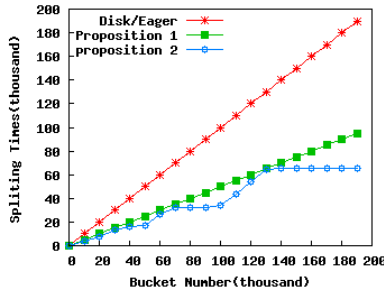


Figure 2 Splitting Times

Obviously the splitting times we gain from Lazy-Split can be denoted in proposition 3

Proposition 3:

$$G(x) = x/2 - H(x) \\ = x/2 - \text{Max} \{f(x), f(2^{\text{Bit}(x)-1}-1)\}$$

B. Tuning the Performance of Search and Update in Lazy-Split schema

There are actually two classes of operations in Eager Split and Lazy Split (search and update). Index may undergo different workloads at different time. Under some circumstances, the operations on index may be dominated by searching, and it also may be mainly updates sometimes. Consequently, we must trade off between search efficiency and update efficiency to achieve a reasonable performance.

Our Lazy-split schema can be elegantly extended to support tuning index performance dynamically.

In the algorithms of search and update, Split-cursor is a very important parameter that can make the procedure very differently at different setting. Especially, if the Split-cursor equals 0, there is no splitting operation in Lazy-Split forever and the efficiency of update is very good in the expense of deterioration of search performance. If the Split-cursor equals b (the number of the total buckets), it requires scanning about only one bucket to find a data item on average, but the performance of update is degraded to the extreme that whenever there is a new bucket appended there is a splitting operation. Lazy-Split schema becomes Eager-Split schema at this time.

In short, our Lazy-Split can trade off between search and update subtly by increasing or decreasing Split-cursor. More details about tuning method should be study in our future work.

V. EXPERIMENTS

In this section we design experiments to validate the efficiency of our design. Especially, the splitting operation is set to be triggered when total bucket number b is four times of Split-cursor in the Lazy-Split schema.

We count the page numbers of read, writing and erasing in stead of reading the clock time directly to get rid of the impact of buffer size. To understand the experiment results clearly, we use the following formula to estimate the time elapsed in our simulations.

$$\text{Time} = \text{read page number} \times 100\mu\text{s} \\ + \text{writing page number} \times 200\mu\text{s} \\ + \text{erasing block number} \times 20\text{ms}$$

A. Eager-Split Vs. Lazy-Split

We examine the efficiency of Lazy-Split and Eager-Split schema by simulating the process of inserting 10,000 records. Figure 3 shows the time efficiency of different splitting schema under different load factors. It is obviously that the Lazy-Split schema outperforms the Eager-Split schema under all load factors. This result is identical to our analysis in section IV.

B. Disk Hash Vs. Our Design

In order to validate our design under real workloads, we run 100,000 operations both on Disk Hash, Eager-Split schema and Lazy-Split schema. Load factor ($= n / b / s$) and Query ratio ($= \text{search operations} / \text{total operations}$) are two parameters in hashing determining the performance. In the first experiment, the *query ratio* is set at 70%, which is the normal query ratio of OLAP services, and the load factor is wandering from 0.1 to 1.0 (Deletion and insertion each takes up half of the 30% update operations). Figure 4.a shows the result. Not surprisingly, Eager Split/Lazy Split schema beat the Disk Hash almost all of the time.

In the second experiment we set the load factor at 0.8, and change the query ration from 0.1 to 0.9. Figure 4.b shows the result. Our design outperforms Disk Hash again.

C. Impact of Workloads

Workloads have significant impact in determining the performance of index. Our Lazy-Split schema is more sensitive to workloads changing. Because it is upgrade update performance in expense of searching.

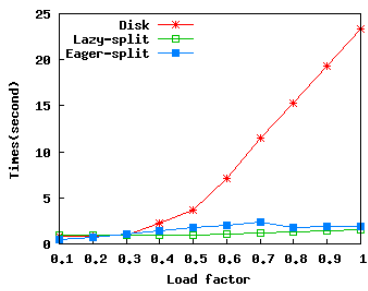


Figure 3 Splitting Performance

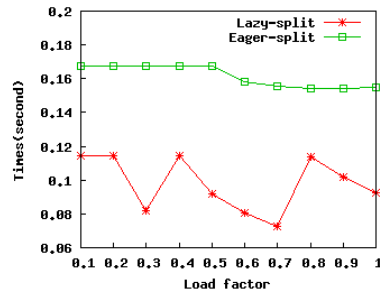


Figure 4.a Query Ratio = 70%

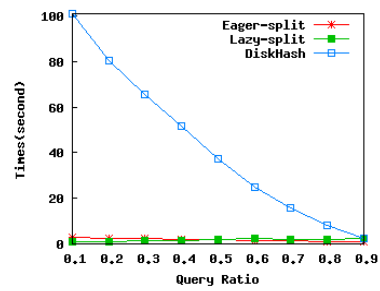


Figure 4.b Load Factor = 0.8

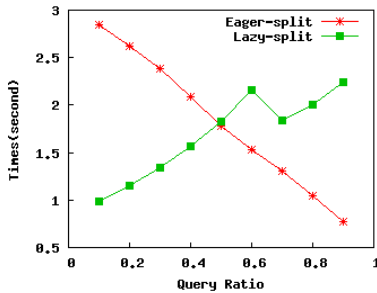


Figure 5 Load Factor = 0.8

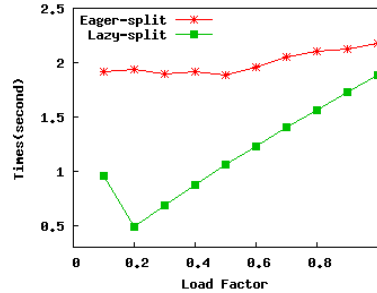


Figure 6.a Query Ratio = 40%

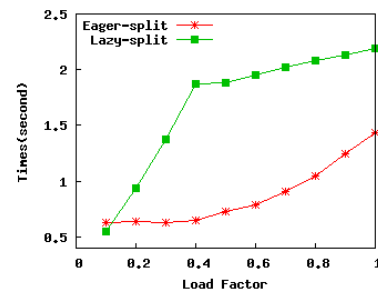


Figure 6.b Query Ratio = 60%

We then design experiments to examine the impact of different workloads. In this experiment we set the load factor at 0.8 and let the query ration change from 0.1 to 0.9. Figure 5 shows the result we have got. It is obviously that Lazy-Split schema outperforms Eager-Split schema when the query ration is below about 50%. However, the situation is reversed when query ratio exceeds 50%.

In Figure 6.a we set the query ratio at 40% and at this time Lazy-Split schema outperforms Eager-Split schema. Figure 6.b shows the reverse situation when we set the query ratio at 60%.

Consequently, we can conclude that Lazy-Split schema is more suitable for update-intensive workloads, and Eager-Split schema performs better for query-intensive workloads just as our analysis shows.

VI. CONCLUSIONS

In this page we have discussed the challenges arised by flash memory and present a new design of dynamic linear hash index. Our design comes in two flavors, Eager-Split schema and Lazy-Split schema. They are designed to serve for different workloads. Our analysis and experiments shows that both of the two schemas outperform Disk hash. Especially, we find that Eager-Split schema is more suitable for search-intensive workloads and Lazy-Split schema performs better for update-intensive workloads.

With the introduction of Split-cursor, our Lazy-split schema can adapt itself dynamically to different query ratios. We give a simple splitting control method (The splitting operation is triggered when the total bucket number is four times of the Split-cursor) in this paper to show the efficiency of Lazy-split scheme. More details about tuning method should be study in our future work.

REFERENCES

[1] (2006) The Sony website. [Online]. Available: http://www.sonymstyle.com.cn/vaio/products/ux/ux_developer/index_08.htm

[2] C.-G. Hwang, "Nanotechnology Enables a New Memory Growth Model", Proceedings of The IEEE, vol. 91, issue 11, pp. 1765 – 1771, Nov. 2003

[3] Intel. Understanding the Flash Translation Layer (FTL) Specification. Application Note AP-684, Intel Corporation, December 1998.

[4] (2004) AXIS COMMUNICATIONS. JFFS home page. Lund, Sweden. Available: <http://developer.axis.com/software/jffs/>

[5] (2002) ALEPHONE, YAFFS: Yet another flash filing system. Cambridge, UK. Available: <http://www.aleph1.co.uk/yaffs/index.html>.

[6] R. Dan, J. WILLIAMS, "A TrueFFS and FLite technical overview of M-Systems' flash file systems", Tech. rep. 80-SR-002-00-6L Rev. 1.30, M-Systems. 1997

[7] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", Proceedings of the 13th ACM Symposium on Operating Systems Principles and the February 1992 ACM Transactions on Computer Systems

[8] S-W. Lee, B. Moon, "Design of flash-based DBMS : an in-page logging approach", Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp.55-66

[9] S. Nath, and A. Kansal, "FlashDB: Dynamic Self-tuning Database for NAND Flash", Int. Conf. on Information Processing in Sensor Networks (IPSN), 2007, pp. 410-419.

[10] C. Wu, L. Chang, and T. Kuo, "An Efficient B-Tree Laye tems and Applications (RTCSA 2003), 2003, pp. 409-430.

[11] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar, "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices", Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, Dec. 2005

[12] (2007) The Samsung website. [Online]. Available: http://www.samsung.com/PressCenter/PressRelease/PressRelease.asp?seq=20070327_0000332936

[13] Samsung data sheet. 1G x 8 Bits / 2G x 8 Bits / 4G x 8 Bits NAND Flash Memory. Nov, 2006.

[14] R. Fagin, J. Nievergelt, N. Pippenger, H. R. Strong, "Extendible Hashing – A Fast Access Method for Dynamic Files", ACM Transactions on Database Systems (TODS) vol. 4, No. 3, pp.315-344, Sep. 1979

[15] W. Litwin, "Linear hashing: A new tool for file and table addressing", Proceedings of the 6th Conference on Very Large Database 1980

[16] H. Garcia-Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice Hall