2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware

Lazy-Update B+-Tree for Flash Devices

Sai Tung On, Haibo Hu, Yu Li, Jianliang Xu

Department of Computer Science, Hong Kong Baptist University
Kowloon Tong, Hong Kong SAR, China
{ston, haibo, yli, xujl}@comp.hkbu.edu.hk

***Abstract*— With the rapid increasing capacity of flash chips, flash-aware indexing techniques are highly desirable for flash devices. The unique features of flash memory, such as the erase-before-write constraint and the asymmetric read/write cost, severely deteriorate the performance of the traditional B+-tree algorithm. In this paper, we propose a new indexing method, called *lazy-update* B+-tree, to overcome the limitations of flash memory. The basic idea is to defer the time of committing update requests to the B+-tree by buffering them in a segment of main memory. They are later committed in groups so that each write operation can be amortized by a bunch of update requests. We identify a victim selection problem for the lazy-update B+-tree and develop two heuristic-based commit policies to address the problem. Simulation results show that the proposed lazy-update method, along with a well-designed commit policy, greatly improves the update performance of the traditional B+-tree while preserving the query efficiency.**

I. Introduction

Flash memory has been adopted as the main storage media for a wide spectrum of mobile and embedded devices. Compared with traditional magnetic hard disks, flash memory is advantageous in various aspects: faster data access, lighter weight, smaller dimensions, better shock resistance, lower power consumption, and less noise. Furthermore, with recent technology breakthroughs in both capacity and reliability, flash-based devices become capable of supporting more complex and data-centric tasks. Therefore, more and more critical DBMS applications are expected to run on these devices.

However, flash memory exhibits a number of unique features which might have a significant impact on the efficiency of state-of-the-art database implementations. Firstly, flash memory has a restriction that an in-place update must be preceded by an erase operation. Even worse, the granularity of erase operations is a block, which is composed of a number of pages. This implies in-place updates are inefficient on flash memory. Secondly, the page write cost is much more expensive than the read cost, while the erase-before-write constraint makes the write cost even higher. Table I shows the read/write/erase speed of a Samsung flash memory chip [8]. We can observe that the ratio of write speed to read speed is 1:2.5, while the ratio of erase speed to read speed is about 1:18.7. Thirdly, each block can bear a limited number of erase cycles (typically 10,000∼100,000 times). A block will be worn out when this number is exceeded. After a significant number of blocks are worn-out, flash memory would become unstable. These features of flash memory lead to a new design principle for flash-aware data access algorithms: they should incur as few writes as possible, even at the price of introducing more reads or computational cost.

TABLE I: Access Time: Hard Disk vs. Flash Memory [8]

Media	Access time		
	Read(2KB)	Write(2KB)	Erase(128KB)
Hard Disk†	12.7ms	13.7ms	N/A
Flash Memory‡	80 μs	200 μs	1.5 ms

Hard Disk†: Seagate Barracuda 7200.7 ST380011A
Flash Memory‡: Samsung K9WAG08U1A 16 Gbits

B+-tree is the most widely-used index structure to expedite query processing. Although B+-tree can achieve high query efficiency, maintaining its structure usually requires intensive, fine-grained updates over B+-tree nodes. Obviously, the traditional B+-tree algorithm does not follow the above design principle and hence would encounter severe performance degradation on flash memory, especially when the workload is update-intensive.

In view of the asymmetric read/write cost, flash-aware indexing methods have been developed in [10], [13] to reduce the update cost of B+-tree by logging data changes on flash pages. In this paper, we suggest a different approach that buffers data updates in a segment of main memory (called *lazy-update pool*). A new indexing method, called *lazy-update* B+-tree, is then proposed. Consider an update sequence $\{q_1, q_2, q_3, q_4\}$, where q_1 and q_3 will insert keys into leaf node 1, while q_2 and q_4 will insert keys into leaf node 2. Under the traditional method, both nodes will be updated twice. In the lazy-update B+-tree, these update requests will be temporarily stored in the lazy-update pool. The benefit is two-folded. First, the buffered update requests can later be committed to the B+-tree in batch, thereby sharing some reading cost of B+-tree in locating the leaf nodes to update. Second, the update sequence can be re-ordered into groups, i.e., $\{q_1, q_3\}$ into one group, and $\{q_2, q_4\}$ into another group. Then, by group-based commitment, both nodes 1 and 2 are updated only once. That is, half of write operations can be saved. Furthermore, the proposed lazy-update B+-tree method is complementary to the aforementioned log-based indexing methods. They can be preceded by our method to group update requests so as to further improve their performance. However, the lazy-update B+-tree is not implemented without cost. A query now will have to search the lazy-update pool in addition to the B+-tree.

This work was supported by the Research Grants Council of Hong Kong (Grants HKBU210808 and HKBU211307) and Natural Science Foundation of China (Grant No. 60833005).

978-0-7695-3650-7/09 $25.00 © 2009 IEEE
DOI 10.1109/MDM.2009.48

Nonetheless, by striking a good trade-off between the saving from group updates and the overhead from increased query complexity, our approach improves the overall performance.

For the lazy-update B+-tree, when a new update request arrives and the lazy-update pool is full, a commit policy should be adopted to select a group of update requests for commitment in order to make room for the new request. An efficient commit policy is important for the lazy-update B+-tree method, as it has a great impact on the effect of update grouping. Ideally, an optimal commit policy should always select those groups which do not have any further update requests to commit. As a result, the number of write operations can be minimized. However, this is unlikely to achieve in practice due to the following two reasons. First, the groups without further update requests do not always exist. Second, future update requests are not known in advance. Therefore, an online commit policy should be carefully designed to maximize the effect of update grouping and thus minimize the write cost.

The rest of the paper is organized as follows. Section II gives an overview of the lazy-update B+-tree method. In Section III, we define the victim selection problem and propose two practical solutions. Section IV shows the performance evaluation results. In Section V, we review the related work on B+-tree algorithms and flash-based data management. Finally, Section VI concludes the paper.

II. Lazy-Update B+-Tree Overview

Normally, B+-tree nodes are stored on the secondary storage media. The main memory usually caches the nodes accessed recently to avoid retrieving them again from the secondary storage. Thanks to in-memory caching, updates on the cached nodes can be committed together. However, with limited memory, only a few nodes can be cached and hence the cached nodes are usually swapped out before they can receive adequate update requests to commit together. As a result, solely relying on such a caching mechanism is unlikely to save many write operations.

To make efficient use of the main memory resources, we propose to divide the main memory into two parts: one for caching corresponding pages of accessed B+-tree nodes as usual (known as *page cache*) and the other for buffering update requests (called *lazy-update pool*). Each update request is in the form of {*key*, *recptr*, *type*}, where *key* is the value of the key to be inserted/deleted, *recptr* is the pointer of the inserted record (null for deletion), and *type* indicates the action type (i.e., 'i' stands for insertion and 'd' for deletion). A request to modify a key is represented by an insert-type request and a delete-type request. For instance, an update request to change an entry from 5* to 6* is denoted by {5, /, d} and {6, *, i}.

Algorithm 1 gives an overview of the *lazy-update* B+-tree method. Whenever an update request arrives, instead of being committed to the B+-tree immediately, it is temporarily stored in the lazy-update pool. Inside this pool, we cancel out those pair update requests which have the same key value but opposite action types, and remove them from the pool. Furthermore, update requests are organized in *groups*. Each set of update requests which are updating the same leaf node forms a group. When the pool cannot accommodate more update requests, guided by a commit policy, one group of requests are selected as victims and committed to the B+-tree to release space. For queries, in addition to searching over the B+-tree by the traditional algorithm, an additional search of the lazy-update pool is required.

```
while a request R arriving do
    if R is an update request then
        if lazy-update pool is full then
            Use a commit policy to select a group of requests
            as victims;
            Commit victims to the B+-tree in bulk;
        Buffer R in the lazy-update pool;
        Use cancel-out policy to eliminate redundant requests;
    else
        /* R is a query request */
        Searching over lazy-update pool to get query result Q1;
        Apply traditional algorithm on B+-tree to get query
        result Q2;
        Merge Q1 and Q2 to get the final query result;
```

Algorithm 1: Overview of Lazy-update B+-tree

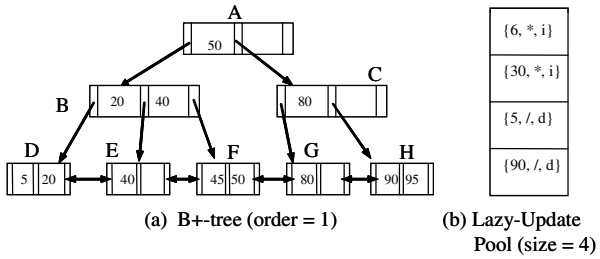

Fig. 1: Lazy-update B+-tree Example

The proposed method groups small updates for the same B+-tree node, thereby reducing the number of write operations. Consider the B+-tree in Figure 1(a), where four update requests are issued, i.e., inserting keys 6 and 30, and then deleting keys 5 and 90. Under the traditional method, these updates are committed in their arrival order. First, the insertion of key 6 will trigger a split for leaf page D, which propagates the update further to pages E, B and A. Next, key 30 will be stored on leaf page E. Finally, the deletions of keys 5 and 90 will incur updates on pages D and H, respectively. As a result, seven pages (i.e., D, E, B, A, E, D, H) will be updated sequentially. Under the proposed method, the update requests will be stored in the lazy-update pool first (see Figure 1(b)). Later on, these requests can be propagated to the B+-tree in groups. First, {5, /, d} and {6, *, i} will be committed to page D together, and then {30, *, i} and {90, /, d} will be committed to page E and H, respectively. As no rebalancing operations (i.e., splitting/merging/redistributing) are required, only three pages (i.e., D, E, H) will be updated — a cost saving of 57% compared to the traditional method.

In the lazy-update B+-tree method, how to select victims when the lazy-update pool is full is a critical issue, because

it affects the number of small updates that can be gathered. Continue with the example in Figure 1. Suppose that besides those four update requests, we have another sequence of four update requests – deleting keys 40 and 95 and then inserting keys 99 and 100. As the lazy-update pool can hold only four requests, this request sequence will be committed in several batches. Consider a commit policy which always selects all in-pool requests as victims. As a result, these requests are committed in two batches, i.e., {6, *, i}, {30, *, i}, {5, /, d} and {90, /, d} in the first batch, while {40, /, d}, {95, /, d}, {99, *, i} and {100, *, i} in the second batch. As described previously, pages D, E and H will be updated in the first batch. In the second batch, {40, /, d} is committed to page E, while {95, /, d}, {99, *, i} and {100, *, i} are committed to page H together, Therefore, there are totally five pages (i.e., D, E, H, E, H) updated using this policy. However, if we choose another policy which commits the requests in three batches: batch 1 — {6, *, i} and {5, /, d}, batch 2 — {30, *, i} and {40, /, d} and batch 3 — {90, /, d}, {95, /, d}, {99, *, i} and {100, *, i}, then in batch 1 only page D is updated; similarly in batches 2 and 3, only pages E and H are updated, respectively. That is, only three pages are updated using this policy. As the commit policy has a great impact on the performance of the lazy-update B+-tree method, in the next section, we will develop two policies which differ in how the victim is selected.

III. Commit Policies

A. Victim Selection Problem

The victim selection problem is to schedule an optimal committing sequence of the lazy-update requests with minimum I/O cost. In detail, we formulate it as follows:

Definition 1: Given a request sequence $S = \sigma_1\sigma_2\cdots\sigma_m$, each of which represents a key insertion/deletion on the B+-tree. Consider a lazy-update pool which can hold up to N update requests. Let P_i be the set of update requests residing in the pool when σ_i arrives, and V_i be the victim group selected to release space. Initially, the pool is empty, thus $P_1 = \phi$ and $V_1 = \phi$. For all P_i $(i = 2, \cdots, m, m+1)$

$$P_i = \begin{cases} P_{i-1} \cup \sigma_{i-1}, & V_{i-1} = \phi \quad \text{if } |P_{i-1}| < N \\ P_{i-1} \cup \sigma_{i-1} - V_{i-1}, & V_{i-1} \subseteq P_{i-1} \quad \text{if } |P_{i-1}| = N \end{cases} \tag{1}$$

The *Victim Selection Problem* is to find a sequence $V = V_1V_2\cdots V_m$ which satisfies (1) and minimizes the following cost function:

$$F(S) = \sum_{i=1}^{m} cost(V_i) + cost(P_{m+1}), \tag{2}$$

where $cost(V_i)$ and $cost(P_{m+1})$ are the costs of committing update requests in V_i and P_{m+1}, respectively.

In this paper, we focus on the online case when the request sequence is unknown in advance. Hence, the selection of victims can only base on the knowledge of past update requests. It is without doubt that an optimal commit policy is hard to obtain in such cases. In the following, we propose two heuristic-based solutions.

B. Biggest Size Policy

A hit occurs if a newly arrived update request has an existing group in the pool to join. In order to increase the hit ratio, we should keep as many groups as possible in the pool. Therefore, it is more profitable to evict one large group than to evict a bunch of small ones to reclaim the same amount of space. Moreover, as a large group has more update requests, the amortized update cost for each request is usually low enough for commitment. This motivates us to propose the *biggest size* policy. Here, the size of a group is defined as the number of update requests residing in the group. This strategy is simple and is easy to implement — among all groups of requests, select the one with the largest size as the victim group, breaking ties by choosing the least-recently-hit group.

C. Cost-based Policy

While the biggest size policy aims to maximize the hit ratio, the objective of the cost-based policy is to minimize the price resulting from evicting victim groups, which is defined as follows. Intuitively, a group gradually expands as long as it stays in the pool to receive new requests. In other words, keeping a group is profitable as it can gather more update requests so that the update cost can be amortized by more requests. A gain function is defined to quantify that profit for each group g:

$$gain(g) = cost(R) + cost(R') - cost(R \cup R'), \tag{3}$$

where R is the set of update requests residing in g, R' is the set of new update requests issued in some future period T , the first two items are the write costs of committing R and R' separately, and the last item is the write cost of committing them together. In essence, the gain value of a group is the saving of write operations which can be obtained if this group is kept in the pool during the period T. We define the price for evicting a group g as its gain value.

In the following, we will discuss how to compute the gain value. To facilitate our analysis, we further define the following notations:

- D: the set of leaf nodes to be updated if R is committed.
- d_k: the k-th leaf node in D.
- $range_k$: the value range of d_k.
- $f(k; r; t)$: the probability of having k update requests whose key values are in the range r during the period t.

For each $range_k$, if $\exists r' \in R'$,$r'.key \in range_k$, then as both of $cost(R)$ and $cost(R')$ include a write on the leaf node d_k, we can save one page write if R and R' are committed together. Otherwise, there is no saving on write operations. Therefore, the saving on d_k is $(1 - f(0; range_k; T)) \cdot C_w$, where C_w is the cost of one page writing. By adding up the cost savings on each leaf node in D, we can get the value of $gain(g)$. That is, we have the following formula:

$$gain(g) \simeq \sum_{d_k \in D} ((1 - f(0; range_k; T)) \cdot C_w). \tag{4}$$

The approximation is due to the omission of some cost savings which do not frequently happen (e.g., all requests in

R are cancelled out due to matched pairs in R', which results in more cost savings; also, cost savings due to updating the same non-leaf nodes are omitted). In order to calculate the gain value for a group in (4), we must first identify a suitable period T and the probability function. In order to reduce the calculation overhead, we set $T = \infty$. Since $f(0; r; \infty) = 0$, (4) can be simplified as:

$$gain(g) \simeq \sum_{d_k \in D} C_w. \quad (5)$$

Thus, the gain value is linear to the number of leaf nodes that are updated if R is committed. Note that although the update requests in a group are applied on a single leaf node, neighboring leaf nodes will also be updated when rebalancing operations are involved (in that case, $|D| > 1$).

The amount of space reclaimed by evicting a group is proportional to its size. In order to minimize the total price for evicting groups, a heuristic solution is to spend the lowest price for each unit of reclaimed space, i.e., to evict the group with the minimum value given by $H(g) = gain(g)/g.size$ whenever the pool is full. We call this *cost-based* policy. Similar to the biggest size policy, the cost-based policy breaks ties by choosing the least-recently-hit group.

To compute the gain value for a group, we need to access its corresponding leaf node to know the node size (i.e., the number of entries in the node). This leads to many additional read operations. However, as we only need to find out the group with the lowest heuristic value during each victim selection process, we propose some pruning techniques to avoid calculating the exact gain value for every group. Due to space limitations, the detailed pruning algorithms are omitted here. Interested readers are referred to [12] for details.

It is easy to see that the cost-based policy degenerates to the biggest size policy when no rebalancing operations are involved in committing the victims, because under this circumstance $|D| = 1$ for each group and thus each group has an equal evicting price (i.e., C_w).

IV. Performance Evaluation

A. Simulation Setup & Performance Metrics

We conducted a simulation study on a PC running Windows XP SP2 with an Intel Quad 2.4GHz CPU and 4GB memory. We implemented an FTL module [5] to emulate a 2GB flash memory whose block size and page size are 128 KB and 2 KB, respectively.

We implemented both the lazy-update B+-tree method and the traditional B+-tree method upon the FTL for performance comparison. Specifically, the algorithms under evaluation include: traditional B+-tree (called *Basic*), lazy-update B+-tree with the biggest size policy (called *Big*), and lazy-update B+-tree with the cost-based policy (called *Cost*). In order to verify the effectiveness of the proposed commit policies, we also implemented the lazy-update method with the LRU policy (called *LRU*) and the FIFO policy (called *FIFO*) for comparison. As mentioned earlier, we do not compare the lazy-update method with the existing flash-aware B+-tree algorithms (e.g., BFTL) because they are complementary to our proposed method.

TABLE II: Default Simulation Parameter Settings

Parameter	Setting
Page size/Block size	2 KB/128KB
Key entry /Update request size	12 bytes
Index size	610,907 key entries
B+-tree order	85 by default
Memory size	1% of index size
Buffer Pool size	50% memory space by default

For a fair comparison, the total main memory allocated for each algorithm is the same (1% of the index size). LRU was used as the replacement policy of the page cache. Both the size of a key entry in each node and the size of an update request are 12 bytes (8 bytes for the key value and 4 bytes for the page address). The order of B+-tree is set to 85 so that each node can exactly fit in a page. By default, the buffer pool of the lazy-update B+-tree algorithms is configured to be 50% of the assigned memory. We summarize the fault parameter settings in Table II.

In our evaluation, we constructed a dataset from DBLP,[1] which contains 610,907 distinct authors. A B+-tree index was built on their names. The authors who appeared in DBLP before year 2007 were used to build the initial B+-tree index (with 540,936 entries). Then, each algorithm was tested by running the following workloads:

- **W-Query (query-intensive workload)**: contains 80% queries, 20% updates.
- **W-Update (update-intensive workload)**: contains 20% queries, 80% updates.

In order to evaluate the performance with different delete/insert ratios, we subdivided the above workloads into W-Query(Insert-only),W-Update(Insert-only),W-Query(Mix) and W-Update(Mix). In the former two workloads, the authors who appeared after year 2007 are inserted into the index. In the latter two, 60% of updates are of insert-type, while 40% are of delete-type. The performance metrics include the numbers of page reads/writes and the CPU time of the algorithms. We also report the overall I/O cost based on the write/read speed given in Table I. The number of blocks erased is omitted as erase operations seldom happen during our simulation.

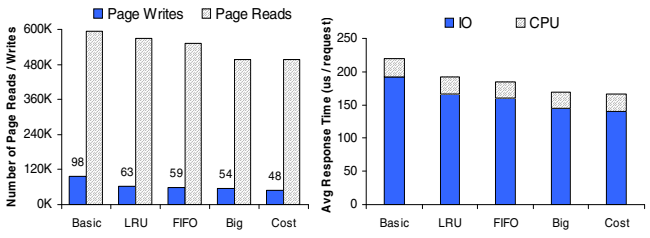

Fig. 2: Performance under W-Query(Insert-only)

[1]DBLP database: http://www.informatik.uni-trier.de/∼ley/db/.

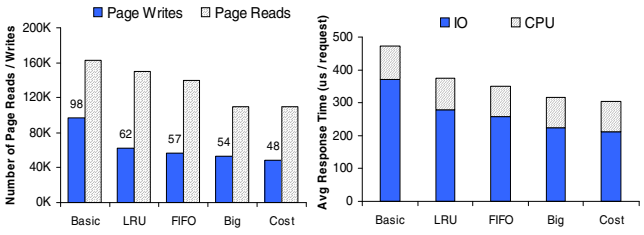

Fig. 3: Performance under W-Update(Insert-only)

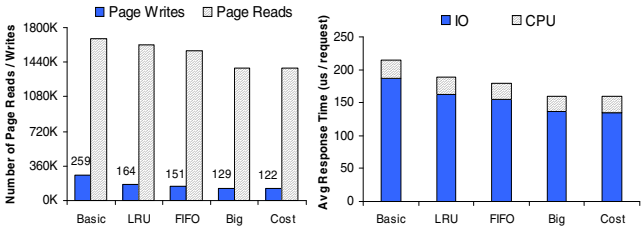

Fig. 4: Performance under W-Query(Mix)

B. Overall Evaluation

Figures 2 through 5 show the performance of all algorithms under different workloads. We can observe that the lazy-update algorithms greatly outperform the traditional B+-tree algorithm: the number of page writes is reduced by half for both query-intensive and update-intensive workloads. Moreover, the number of page reads is about 16% less for the query-intensive workload, while it is over 33% less for the update-intensive workload. This is because with the lazy-update method, the update cost of each B+-tree node is amortized by a group of requests and thus considerable write/read operations can be saved. In addition, the overall computational cost of lazy-update algorithms is not hindered by the extra cost of searching over the lazy-update pool for query processing, as their computational cost to update a node is also amortized by a group of requests.

Among different commit policies, the proposed biggest size policy and cost-based policy outperform conventional replacement policies (i.e., LRU and FIFO) on both read and write costs. Specifically, the number of page writes is 16%∼20% less than that of conventional replacement policies, while it is 10%∼26% less for the number of page reads. In general, the cost-based policy incurs the fewest write operations and achieves the best overall performance, whereas the biggest size policy requires the fewest read operations and least computational cost. This can be explained as follows. Since the cost-based policy takes the rebalancing cases into account, it can save more write operations. However, as the victim selection of the cost-based policy requires accessing the leaf nodes for calculating gain values, additional read operations and computational cost are incurred.

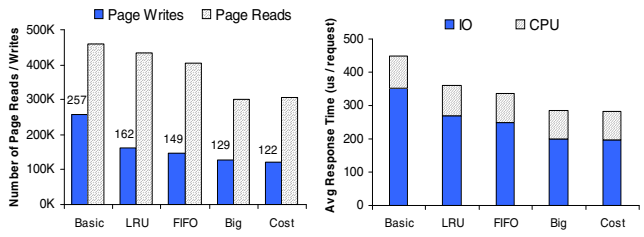

Fig. 5: Performance under W-Update(Mix)

C. Effect of Buffer Pool Size

In this set of the experiment, we evaluate the performance of lazy-update algorithms by running the workload W-Query(Mix) under different buffer pool sizes. Figure 6 shows the results while the buffer pool ratio is varied from 0.1 to 0.94. Note that 0.94 is the maximum ratio we can set as the page cache should hold at least 2 pages (one for caching the root node, and the other for caching the current processing node). When the ratio increases from 0.1 to 0.9, the number of page reads/writes decreases. This can be explained as follows. The bigger is the buffer pool, the more can update requests be buffered. As a result, it is more likely to group update requests and hence each update cost can be amortized by more requests. When the ratio is higher than 0.9, however, as there is little space for caching nodes, frequent page swappings in the page cache are incurred and hence the performance becomes worse.

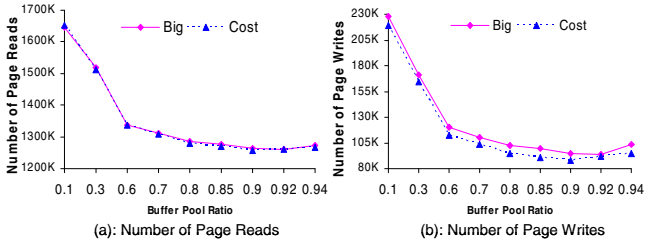

Fig. 6: Effect of Buffer Pool Size

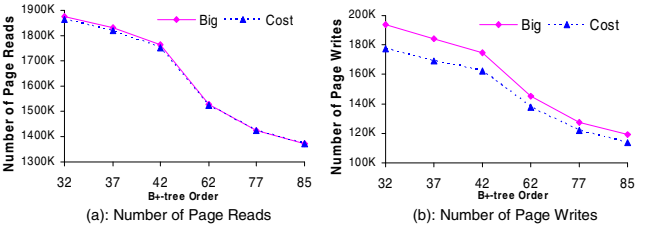

Fig. 7: Effect of B+-tree Order

D. Effect of B+-Tree Order

We conducted a study on the impact of B+-tree order on the performance of lazy-update algorithms with the biggest size and cost-based policies. Figure 7 shows the W-Query(Mix) results when the order is varied from 32 to 85. The number of page reads/writes decreases when the B+-tree order increases.

This is due to the following two reasons. First, as each node can hold more entries, the total number of B+-tree nodes is decreased. As a result, the probability of update requests being applied on the same leaf nodes is greatly increased and hence more cost savings can be achieved. Second, when the order increases, the frequency of rebalancing occurrences is reduced as well. We can further observe from Figure 7(b) that the performance improvement of *Cost* over *Big* becomes smaller when the B+-tree order increases. Specifically, compared with *Big*, *Cost* can save 16454 (i.e., 8.5%) write operations when the order is 32, but it is reduced to 5433 (i.e., 4.3%) when the order is 85. This is because the cost-based policy outperforms the biggest size policy by taking into account rebalancing cases, which seldom happen when the tree order is large.

V. Related Work

Data management on flash-based media has received much attention from research community in recent years. To enable a quick deployment of flash-memory technology, early work attempted to hide the unique characteristics of flash memory. They focused on simulating traditional magnetic disks by flash memory chips. Kawaguchi et al. [5] proposed a software module called flash translation layer (FTL) to transparently access flash memory, so that conventional disk-based algorithms and access methods can work as usual. To overcome the erase-before-write constraint, an out-of-place update scheme was adopted and various garbage collection mechanisms [3], [5], [7] were proposed to reclaim invalidated space. To lengthen the lifetime of flash memory, wear-leveling algorithms that attempted to evenly distribute writes/erases across all pages were developed in [2], [4].

Besides these fundamental achievements, recent work shifted to exploit the characteristics of flash memory to enhance the performance of file systems and DBMSs. In view of the slow write speed on flash memory, the log structure was adopted to reduce the number of write operations. Along this direction, some flash-aware log-based file systems like YAFFS [1] and JFFS [11] were proposed. For DBMSs running on flash-based media, Lee and Moon [8] presented a novel design of data logging called in-page logging (IPL) to further improve the logging performance. Kim and Ahn [6] proposed to use the in-device write buffer to improve the random write performance of flash storage. Lee et al. [9] conducted a case study to investigate how the performance of conventional database applications is affected by the new flash-based disk.

Research efforts have also been put into optimizing B+-tree algorithms. To overcome the asymmetric read/write speed and the erase-before-write limitation on flash-based media, some flash-aware B+/B-tree algorithms were developed. Wu et al. [13] introduced BFTL, an optimized B-tree layer for flash memory. In BFTL, all changes are written on log pages and therefore expensive update cost for each node is avoided. As a side effect, an in-memory Node Translation Table (NTT) is required to maintain the list of log pages for each node. Observing that the log-based indexing scheme is not suitable for read-intensive workload on some flash devices, Nath and Kansal [10] developed FlashDB, which uses a self-tuning B+-tree that dynamically adapts its storage structure to the workloads and storage devices. Although these indexing methods degrade the query performance due to the need of accessing multiple log pages when searching a single node, the update cost is successfully reduced.

VI. Conclusion

In this paper, we discussed the challenges of maintaining B+-tree on flash memory. To overcome the asymmetric read/write limitation, we proposed a new indexing method, called lazy-update B+-tree, to group update requests in order to reduce the number of write operations. For the lazy-update B+-tree, we identified a critical problem of victim selection, and proposed two commit policies. Simulation results show that the lazy-update B+-tree significantly improves the update performance of the traditional B+-tree while still preserving the query efficiency.

References

[1] Aleph one ltd., embedded debian, yaffs: A nand-flash file system. http://www.aleph1.co.ukyaffs, 2002.

[2] L.-P. Chang and T.-W. Kuo. An efficient management scheme for large-scale flash-memory storage systems. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 862–868, New York, NY, USA, 2004. ACM.

[3] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *Trans. on Embedded Computing Sys.*, 3(4):837–863, 2004.

[4] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 212–217, New York, NY, USA, 2007. ACM.

[5] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[6] H. Kim and S. Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[7] H.-j. Kim and S.-g. Lee. A new flash memory management for flash storage system. In *COMPSAC '99: 23rd International Computer Software and Applications Conference*, page 284, Washington, DC, USA, 1999. IEEE Computer Society.

[8] S.-W. Lee and B. 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.

[9] 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 '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086, New York, NY, USA, 2008. ACM.

[10] S. Nath and A. 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.

[11] D. Woodhouse. Jffs: The journalling flash file system. In *Proceedings of the of the Ottawa Linux Symposium*, pages 177–182, 2001.

[12] S. T. On, H. Hu, Y. Li, and J. Xu. Lazy-update B+-tree for flash devices. Technical Report, Hong Kong Baptist Univeristy, 2009.

[13] C. Wu, T. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.