On ST, Hu H, Li Y *et al.* Flash-optimized B+-tree. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 25(3): 509–522 May 2010

Flash-Optimized B+-Tree

Sai Tung On (安世通), Haibo Hu (胡海波), Yu Li (李　宇), and Jianliang Xu (徐建良), *Senior Member*, *IEEE*

Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong, China

E-mail: {ston, haibo, yli, xujl}@comp.hkbu.edu.hk

Received June 29, 2009; revised March 3, 2010.

Abstract With the rapid increasing capacity of flash memory, 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 an optimized indexing method, called *lazy-update* B+-tree, to overcome the limitations of flash memory. The basic idea is to defer the committing of update requests to the B+-tree by buffering them in a segment of main memory. They are later committed in groups so that the cost of 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 this 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.

Keywords B+-tree, flash memory, indexing, lazy update

1 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.

However, flash memory exhibits a number of unique features which might have a significant impact on the performance of database systems. First, flash memory has a restriction that an in-place update (i.e., writing new data directly on the original page) 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. Second, 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 1 shows the read/write/erase speed of a Samsung flash memory chip[1]. 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. Third, each block can bear a limited number of erase cycles (typically 10000∼100000 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 1. Hard Disk Vs. Flash Memory[1]

Media	Access Time		
	Read	Write	Erase
Hard Disk†	12.7 ms	13.7 ms	N/A
Flash Memory‡	80 μs	200 μs	1.5 ms

†: Seagate Barracuda 7200.7 ST380011A

‡: Samsung K9WAG08U1A 12GB

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 [2-3]

Regular Paper

This work is supported by the Research Grants Council of Hong Kong under Grant No. HKBU210808 and the National Natural Science Foundation of China under Grant No. 60833005.

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*). An optimized 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. Moreover, 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. 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 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 group updates. 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 group updates and thus minimize the write cost. We will develop two heuristic-based solutions to address this problem.

The rest of the paper is organized as follows. In Section 2, we introduce the background of our research. Section 3 gives an overview of the lazy-update B+-tree method. In Section 4, we define the victim selection problem and propose two practical solutions. Section 5 shows the performance evaluation results. In Section 6, we review the related work on B+-tree algorithms and flash-based data management. Finally, Section 7 concludes the paper.

2 Background

2.1 Flash Memory

There are two major architectures in flash memory design: NOR and NAND flash memories. NOR flash memory has a faster random access speed but a lower storage capacity so it is preferred for code storage. NAND flash memory has a denser architecture and a simpler interface, and therefore it offers a much higher storage capacity and is widely used for data storage. In this paper, we focus on NAND flash memory. Hereafter, we use the term flash memory to refer to NAND-type flash memory.

A flash memory chip is organized in many blocks and each block is composed of a fixed number of pages. Each page consists of a data area and a spare area. The spare area in each page usually stores the error correction code and other management information such as LBA (logical block address). A block is the smallest unit for erase operations, while all read and write operations are at the page granularity. The typical block size and page size are 128 KB and 2 KB, respectively.

Flash memory has a number of characteristics which distinguishes itself from magnetic hard disks: 1) asymmetric read/write cost; 2) the erase-before-write constraint; 3) the endurance issue — each block can be erased for only a limited number of cycles before it is worn out; 4) no mechanical latency.

Flash devices usually access the embedded flash chips through a software layer called FTL (flash translation layer)[4-5]. The FTL provides a disk-like interface, which includes the capability to read and write a page directly without caring about the special characteristics of flash memory. It consists of components such as logical/physical address mapping, garbage collection and wear-leveling. Logical/physical address mapping can support out-of-place updates and thus overcome the erase-before-write constraint. There are several mapping strategies such as page mapping[5], block mapping[6] and hybrid mapping[7]. Garbage collection is responsible for reclaiming invalidated space due to out-of-place operations, while wear-leveling aims to lengthen the lifetime of flash memory by uniformly distributing writes/erases across the entire storage space. With the help of FTL, the file systems or DBMSs designed for disks can function adequately without any modification. However, as traditional algorithms do not

exploit the unique I/O features of flash memory (e.g., asymmetric read/write access speed), they are unlikely to yield their best attainable performance.

2.2 B+-Tree

B+-tree is the most popular index structure for accelerating queries over a large amount of data. It has been widely used in various file systems and DBMSs[8-9]. Many queries such as equality queries and range queries can be efficiently evaluated on B+-tree. It provides efficient operations to retrieve, delete, and insert data.

B+-tree is a balanced tree in which every path from the root to a leaf has the same length. There are two types of nodes, namely, leaf nodes and non-leaf nodes. All keys to be indexed are stored in leaf nodes, while other keys residing in non-leaf nodes are used for directing searches to the proper leaf nodes. Fig.1(a) shows an exemplary B+-tree of order 1. For ease of presentation, without loss of generality, we assume each B+-tree node has the same size as a flash page in this paper. B+-tree answers range query $[s, t]$ as follows: first, the leaf page that contains the starting key s is located by following a search from the root to the leaf level; then, the sibling leaf pages are retrieved (by traversing the links among them) until the one that contains the ending key t has been reached. For example in Fig.1(a), the query [5, 80] will visit the nodes on pages A, B, D, E, F and G. Insertion/deletion algorithms on B+-tree are more complicated than queries. Assume that we want to insert a new entry with key 6 into the tree in Fig.1(a). The target leaf page is first located by following the root-to-leaf path (hence, pages A, B and D are accessed). In a normal case, the entry is directly inserted into that leaf page. However, *overflow* occurs when the leaf page (i.e., D in the example) is full (i.e., it already contains the maximum number of entries). In this case, node splitting is performed (i.e., entries 5* and 6* are kept in the original page and entry 20* forms a new leaf node). Along with the creation of the new node, an entry pointing to this newly created node must be inserted into its parent page (i.e., B). Node splitting might propagate to upper levels if *overflow* also occurs in the parent page. For a deletion, the leaf node that contains the entry to be removed is first identified. If the node does not generate an *underflow* (i.e., it does not violate the minimum node utilization), the deletion terminates. Otherwise, the algorithm attempts to avoid a merge if some entries can be borrowed from sibling nodes (known as redistributing). For example, redistributing occurs when deleting entry 40* in Fig.1(a). Specifically, the entry 45* in page F can be redistributed to page E so that both pages can have sufficient entries. Node merging is performed when redistributing is not allowed. Similar to node splitting, merging might also propagate to upper levels if *underflow* also occurs in the parent page.

As can be observed from the above, maintaining the B+-tree structure for data insertions/deletions involves intensive small updates (in several bytes). As a page is the basic unit for read/write operations on flash memory, any small update on a node will lead to writing the entire content of its residing page to some clean page (due to out-of-place updates). Intensive small updates would consume up storage space quickly and shorten the lifetime of flash memory. Moreover, these updates will eventually incur erase operations whose cost is very expensive in terms of both access time and energy consumption. As such, these intensive small updates are extremely costly on flash memory, which explains why the performance of the traditional B+-tree algorithm severely deteriorates on flash memory.

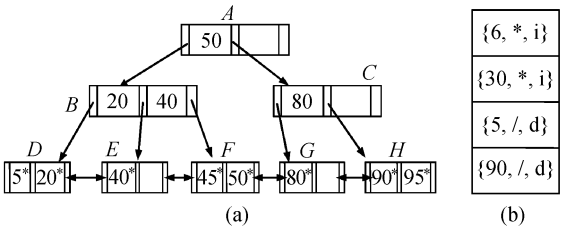


Fig.1. Lazy-update B+-Tree example. (a) B+-tree (order = 1). (b) Lazy-update pool (size = 4).

3 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. Lazy-Update B+-Tree Overview

```
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;
    end
    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;
  end
end
```

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.

The proposed method groups small updates for the same B+-tree node, thereby reducing the number of write operations. Consider the B+-tree in Fig.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 Fig.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 pages 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 Fig.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.

4 Commit Policies

4.1 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 (Victim Selection Problem). *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 $P_1 = \varnothing$ and $V_1 = \varnothing$. For all P_i $(i = 2, \ldots, m, m+1)$

$$P_i = \begin{cases} P_{i-1} \cup \{\sigma_{i-1}\}, V_{i-1} = \varnothing, \text{ if } |P_{i-1}| < N, \\ P_{i-1} \cup \{\sigma_{i-1}\} - V_{i-1}, \quad V_{i-1} \subseteq P_{i-1}, \\ \qquad\qquad\qquad\qquad \text{if } |P_{i-1}| = N. \end{cases} \quad (1)$$

The *Victim Selection Problem* is to find a sequence $V = V_1 V_2 \cdots V_m$ which satisfies (1) and minimizes the following cost function:

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

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.

4.2 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.

4.3 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:

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

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:

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

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:

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

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, in Subsection 4.4.2, we will propose some pruning techniques to avoid calculating the exact gain value for every group. As shown in Subsections 5.5, by adopting pruning techniques, such additional read operations can be greatly reduced.

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

4.4 Implementation Issues

4.4.1 Data Structure for Pool Management

The update requests in the lazy-update pool should be well organized to support efficient search and fast victim identification. We propose to organize them with an orthogonal list. Fig.2(a) illustrates how update requests in Fig.1 are organized. As shown, requests are hierarchically clustered to enable efficient searching and traversing.

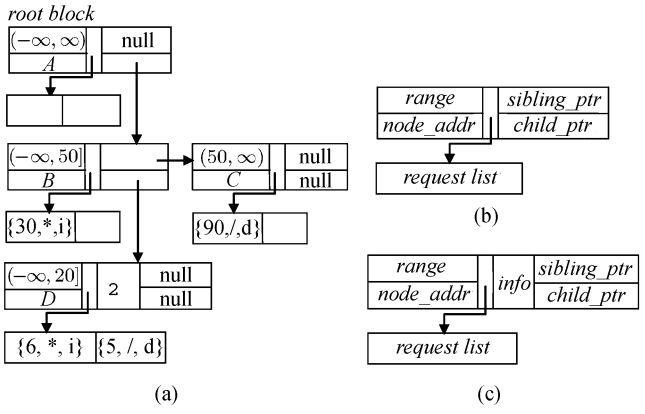


Fig.2. Proposed orthogonal list. (a) An orthogonal list example. (b) Index block structure. (c) Group block structure.

Each block corresponds to a node in the B+-tree. The one corresponding to the root node is called *root block*. A block b is called *child block* of b' if b's corresponding node is the child of b''s. Moreover, b and b' are called *sibling blocks* if their corresponding nodes are siblings. There are two types of blocks in the proposed orthogonal list: *index block* and *group block*.

• *index block* corresponds to a non-leaf node. As shown in Fig.2(b), it has several components including *range*, *request_list*, *child_ptr*, *node_addr* and *sibling_ptr*, where *range* and *node_addr* are the value range and page address of its corresponding node respectively, *request_list* is the array of update requests buffered in the block, *child_ptr* represents a pointer to the head of the chain of its child blocks and *sibling_ptr* represents a pointer to its sibling block. Note that an update request is buffered in an index block if and only if its key is in the range of this block but is not in the range of any of its child blocks.

• *group block* corresponds to a leaf node. In addition to the components in *index block*, each group block has an extra component called *info* to store the information for victim selection (e.g., the node size). Fig.2(c) shows its structure.

Algorithm 2 shows the pseudo code upon an insertion request. To insert a request into the orthogonal list, the proper block is first located by searching from *root block* to the bottom. Then, the request is directly inserted into the *request_list* of that block.

Algorithm 2. Insertion Algorithm

Parameter: the update request q, the root block *root*
block = *root*, *temp* = *root*;
while true **do**
  while *temp* ≠ null **and** q.key **not in** *temp*.range **do**
    temp = *temp.sibling_ptr*;
    if *temp* = null **then** break;
    else *block* = *temp*, *temp* = *block.child_ptr*;
end
if there **exists** a matched pair p of q **in**
block.request_list **then**
  use the cancel-out policy to eliminate p from the pool;
else
  insert q into *block.request_list*;
end

Initially, there is only a *root block* in the orthogonal list. All update requests are buffered in this block. When it is time to select victims, these update requests are sorted by their key values and divided into several parts according to the entries of the root node. Then associated child blocks are built and added to the

orthogonal list to accommodate each part of update requests. Such process continues recursively in every block until all update requests have formed their groups for victim selection. Similarly, when searching the B+-tree for query results, the accessed nodes are used to assist in creating new blocks and thus requests can be clustered at a more fine-grained level. By adopting the above approaches to form groups lazily, some savings on the cost of accessing nodes for victim selection can be obtained. We detail them in Algorithms 3 and 4, respectively. Note that, although Algorithm 4 is based on equality queries, other types of queries such as range queries can be handled with the similar rationale.

When a group block is selected as victim, it is removed from the orthogonal list.① An index block is also removed if it does not contain update requests or child blocks. In other words, a block is kept only when there are update requests within their value ranges.

Algorithm 3. Group Forming
Parameter: the root block *root*
push *root* into stack, *bk* = null;
while *stack* **not** *empty* **do**
  bk = POP(stack);
  get the chain of its child blocks *chain* by *bk.child_ptr*;
  if *bk* **is not** *a group block* **and** *bk.request_list* **is not** *empty* **then**
    get the corresponding node *node* by *bk.node_addr*;
    divide *bk.request_list* into $r[1], \ldots, r[m]$ by *node*'s entries;
    for $i \leftarrow 1$ **to** m **do**
      create block *chd*[*i*] with $r[i]$;
      insert *chd*[*i*] into *chain* and keep *chain* sorted, update *bk.child_ptr* if necessary;
    end
  end
  foreach child block *child* in *chain* **do** PUSH(*child*);
end

Algorithm 4. Handling Equality Queries
Input: the query request *q*, the root block *root*
Output: query result *R*
follow the method in Algorithm 2 to locate the proper block *bk* using parameter *q*.key and *root*;
temp = null;
foreach request *uq* in *bk.request_list* **do**
  if *uq.key* = *q.key* **then** *temp* = *uq*, break;
end
if *temp* ≠ null **then** *R* = *temp.recptr*;
  /* no need to query on B+-tree */
else
  retrieve the corresponding node *startnode* by *bk.node_ptr*;
  search from *startnode* to leaf on B+-tree to get result *R*;
  let *nodes* be the set of nodes accessed in the above searching;
  foreach B+-tree node *node* in *nodes* **do**
    get the block *b* which corresponds to *node*;
    try to redistribute *b.request_list* at a fine-grained level; /* see Algorithm 3 lines 6∼11 */
  end
end
return *R*;

Algorithm 5. Victim Selection with Pruning
Input: the root block *root*
Output: the victim group block *v*
push *root* into stack;
bk = null, *v* = null; /* *bk* is a block */
while stack **not** empty **do**
  bk = POP(stack);
  get the chain of its child blocks *chain* by *bk.child_ptr*;
  if *bk* **is not** a group block **and** *bk.request_list* **is not** empty **then**
    let *size* = the number of requests in *bk.request_list*;
  if *v* = *null* **or** $H(v) \cdot size \geqslant 1$ **then**
    get the corresponding node *n* by *bk.node_addr*;
    try to redistribute *bk.request_list* at a fine-grained level; /* see Algorithm 3 lines 6∼11 */
    end
  end
  else if *bk* is a group block **and** (*v* = null **or** $H(v) > H(bk)$) **then**
    if *bk.node_size is unknown* **then**
      get the corresponding node *l* by *bk.node_addr*;
      bk.node_size = *l.node_size*;
    end
    if *v* = null **or** $H(v) > H(bk)$ **then**
    v = *bk*;
    else if $H(v) = H(bk)$ **and** *v.order* > *bk.order* **then** *v* = *bk*;
  end
  foreach child block *child* in *chain* **do**
  PUSH(*child*);
end
return *v*;

①If rebalancing operations are involved when committing the victim, the orthogonal list might be updated correspondingly.

4.4.2 Performance Enhancement

4.4.2.1 Pruning Technique

Both group forming and heuristic value calculating require accessing tree nodes, which might degrade the performance of the proposed lazy-update method. In order to minimize the number of read operations, we propose the following pruning technique.

In Algorithm 3, when the pool is full, update requests residing in upper blocks are moved into lower levels to form group blocks for victim selection. Then the biggest size policy or cost-based policy is adopted to select victims. We propose to merge these two phases, and prune those update requests or group blocks which are impossible to be selected as victims. Specifically, for each index block in the orthogonal list, it is firstly regarded as a group block and a bound of its heuristic value is calculated. If the bound value indicates its update requests might be selected as victims, then its corresponding node is retrieved to create child blocks and its update requests are moved into these child blocks. Otherwise, this index block is pruned from the victim selection process. For each group block, if calculating its heuristic value requires accessing its corresponding leaf node[②], then the lower bound value is first computed by assuming its gain value to be one page write. Only when the bound value indicates the group block is possible to be selected as a victim, would the exact heuristic value be calculated by acquiring the size of its corresponding leaf node. We detail this pruning technique in Algorithm 5 based on the cost-based policy. For the biggest size policy, the only difference is the pruning condition.

4.4.2.2 Efficient Use of Page Cache

Besides the pruning technique for read operations, we also make efficient use of the page cache for write operations. As the page cache is for B+-tree nodes, it is profitable to commit those requests whose corresponding leaf nodes are cached and dirty (i.e., the leaf node is updated previously but not yet written back to flash memory). To implement this idea, when a dirty leaf node is about to be swapped out from the page cache, we verify if there are in-pool requests updating this node. If so, these requests are committed immediately.

4.4.2.3 Super Group

Within the proposed policies, a group is the basic unit for victim selection. However, extra cost savings might be obtained if a group of requests are committed together with the requests residing in its sibling groups. Generally, we propose to evict a set of groups G as a whole if the following condition holds.

Condition 1. *$\forall g \in G$, $\exists g' \in G$, $cost(g)+cost(g') > cost(g \cup g')$, where $cost(g)$ and $cost(g')$ are the write costs of committing g and g', respectively, while $cost(g \cup g')$ is the write cost of committing g and g' together.*

Each group in the pool may belong to several such sets, the largest one among which is called *super group*, whose definition is presented below.

Definition 2 (Super Group). *Let U be the universe of groups in the pool. The Super Group S is defined as a set of groups which satisfies Condition* 1 *and the following condition*

$$\forall g \in U - S, \forall g' \in S, cost(g) + cost(g') = cost(g \cup g'), \tag{6}$$

where the meanings of those cost functions are the same as Condition 1.

For simplicity, we only consider the cost savings due to updating leaf nodes and their parent nodes. This is reasonable because cost savings on higher levels of the tree seldom happen. As a result, a super group is simplified as a set of sibling groups who share the same parent node in the B+-tree. In what follows, we enumerate the cases when two sibling groups g and g' satisfy the inequality shown in Condition 1. Fig.3 shows examples of these cases.

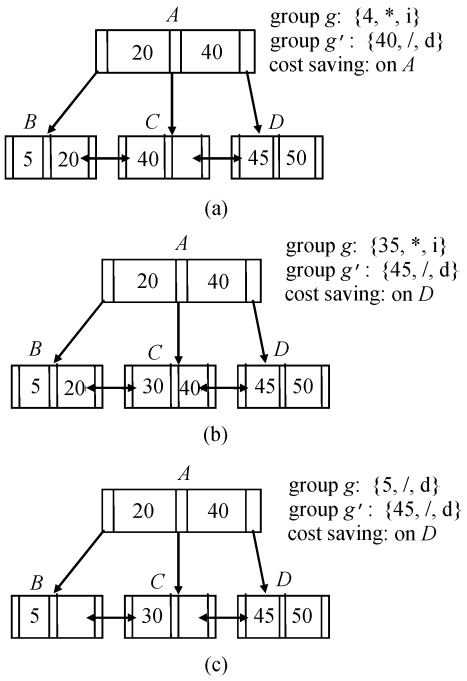


Fig.3. Cost saving cases. (a) Example of case 1. (b) Example of case 2. (c) Example of case 3.

[②]Such a case occurs when applying the cost-based policy.

Case 1: Both of g and g' incur rebalancing operations. As both of them involve a write on the parent node, cost saving exists.

Case 2: g incurs a rebalancing operation while g' does not. Furthermore, g is the immediate sibling (the left/right sibling) of g'. In this case, there exists cost saving as both groups involve a write on the corresponding leaf node of g'.

Case 3: g incurs a merging operation while g' does not involve any rebalancing operation. Furthermore, g' is the indirect sibling of g. In this case, since the merge operation leads to updating the forward/backward pointer of the corresponding leaf node of g', cost saving exists if they are committed together.

To summarize, the operation type (normal/splitting/merging/redistributing) is the most important factor to decide whether g and g' belong to the same super group. Fortunately, the operation type for a group can be easily detected by its size and the size of its corresponding leaf node. To find the super group for a group, its sibling groups are retrieved and verified by Condition 1 iteratively until no more groups can be added into the super group. We detail it in Algorithm 6.

Algorithm 6. Forming Super Group
Input: an in-pool group g
Output: the super group S containing g
$S = \{g\}$;
get g's sibling groups and store them in *candidate*;
flag = false; /∗indicates whether the process finishes∗/
while *flag* **not** true **do**
  flag = true;
  foreach group p in *candidate* **do**
    if $\exists g' \in S$, g' and p are in cases 1, 2 or 3 **then**
      $S = S \cup \{p\}$;
      flag = false;
      candidate = *candidate* $-\{p\}$;
    end
  end
end
return S;

There are two approaches to apply the super group technique to the biggest size and cost-based policies. One approach is to select victim based on groups as usual, then after a victim group is selected, we get its corresponding super group and commit all of its group members together. Another approach is to select victim based on super groups. Specifically, for each group, we get its corresponding super group. Then among all super groups, their heuristic values (by $H(S) = \sum_{g_k \in S} gain(g_k)$) are calculated for victim selection. Finally, update requests residing in the groups of the victim super group are committed together. For the second approach, each in-pool group needs to form its super group for victim selection, while for the first approach, only one super group is formed. As forming a super group requires reading a number of leaf nodes, the second approach surely incurs much more read operations than the first approach. On the other hand, as the second approach strictly adopts the super group as the basic unit for victim selection, it is expected to incur fewer write operations than the first approach. In Section 5, we will conduct a simulation study to compare their performance.

5 Performance Evaluation

5.1 Simulation Setup and Performance Metrics

We conducted a simulation study on a PC running Windows XP SP2 with an Intel Quad 2.4 GHz CPU and 4 GB memory. We implemented an FTL module[5] to emulate a 2 GB 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 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 method.

For a fair comparison, the total main memory allocated for each algorithm was 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 were 12 bytes (8 bytes for the key value and 4 bytes for the page address). The order of B+-tree was set to 85 so that each node could exactly fit in a page. By default, the buffer pool of the lazy-update B+-tree algorithms was configured to be 50% of the assigned memory. We summarize the default parameter settings in Table 2.

In our evaluation, we constructed a dataset from DBLP, which contains 610 907 distinct authors③. Each author name was normalized to a floating number in

③The DBLP data is available from http://www.informatik.uni-trier.de/∼ley/db/.

Table 2. Default Simulation Parameter Settings

Parameter	Setting
Page size/Block size	2 KB/128 KB
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

the domain of [0, 1], on which a B+-tree index was built. 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), W-Update(Insert), W-Query(Mix) and W-Update(Mix). In the former two workloads, the authors who appeared after year 2007 were inserted into the index. In the latter two, 60% of updates were of insert-type, while 40% were of delete-type. The performance metrics to be evaluated included the number of page reads/writes and the CPU time of the algorithms. We also reported the overall I/O cost based on the write/read speed given in Table 1. The number of blocks erased was omitted as erase operations seldom happened during our simulation.

5.2 Overall Evaluation

Figs. 4∼7 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. For the computational cost, the lazy-update algorithms require less time. This can be interpreted as follows. As the update requests are well organized by the orthogonal list to support efficient search and fast victim identification, the extra cost of searching over the lazy-update pool for query processing becomes trivial. Also, the computational cost to update a node is amortized by a group of requests.

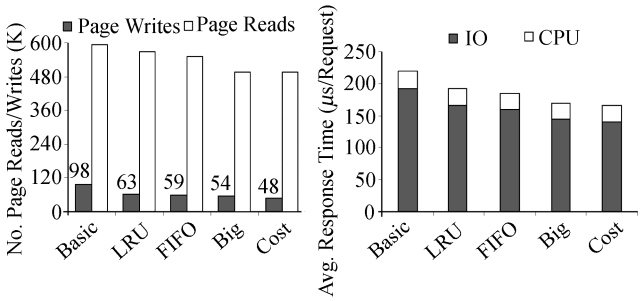


Fig.4. Performance under W-Query (Insert).

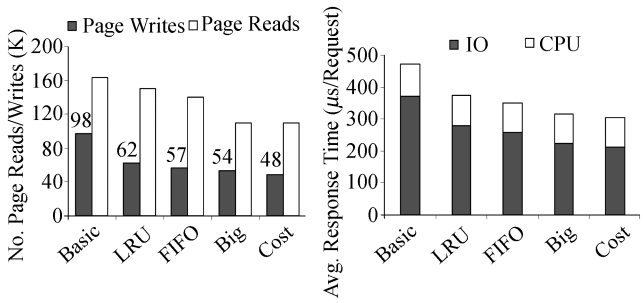


Fig.5. Performance under W-Update (Insert).

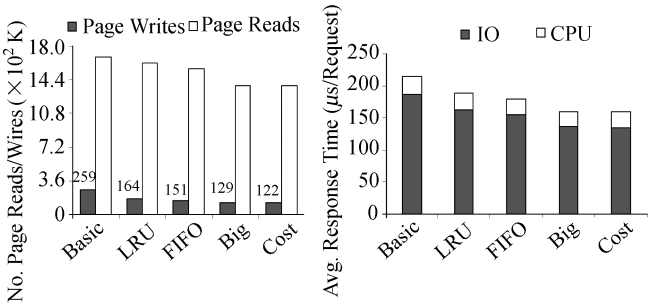


Fig.6. Performance under W-Query (Mix).

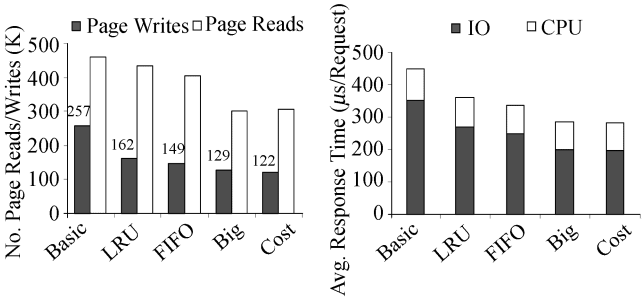


Fig.7. Performance under W-Update (Mix).

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

in terms of 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 into account the rebalancing cases, 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.

5.3 Effect of Buffer Pool Size

In this set of the experiment, we evaluated the performance of lazy-update algorithms by running the workload W-Query (Mix) under different buffer pool sizes. Fig.8 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 at least hold two 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 the buffer pool, the more the update requests can 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.

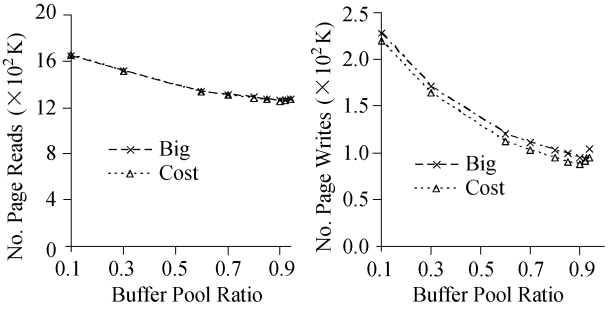


Fig.8. Effect of buffer pool size.

5.4 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. Fig.9 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 Fig.9(b) that the performance improvement of Cost over Big becomes smaller when the B+-tree order increases. Specifically, compared with Big, Cost can save 16 454 (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.

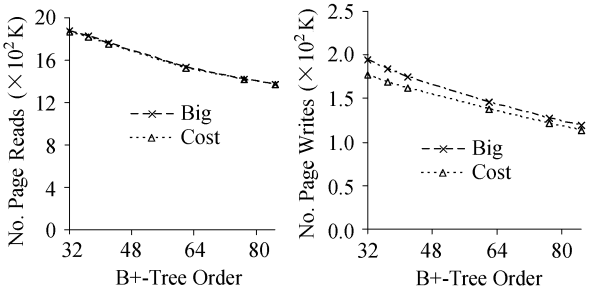


Fig.9. Effect of B+-tree order.

5.5 Effect of the Pruning Technique

Fig.10 shows the performance of lazy-update algorithms based on the biggest size and cost-based policies with and without the pruning technique (Subsection 4.4.2.1) under different workloads. We can see that the biggest size policy can save 4.5%∼15% read operations with the pruning technique, while the saving is much bigger (9.8%∼33%) for the cost-based policy. This is because with pruning, the update requests are moved to form groups only when they might be selected as victims and therefore the cost of creating groups can be amortized by more requests. Furthermore, since the exact heuristic value of each group is computed only when necessary, pruning also reduces additional read

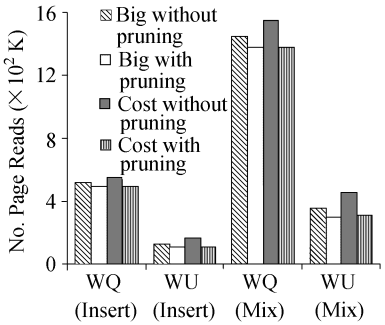


Fig.10. Impact of pruning.

operations due to accessing corresponding leaf nodes for calculating the gain value.

5.6 Effect of the Super Group

In order to investigate the effectiveness of the proposed super group technique, we evaluated the performance of lazy-update method under the cost-based policy with/without adopting the super group approach (Subsection 4.4.2.3). The results shown in Fig.11(a) demonstrate that the number of write operations can be reduced by 2.5%∼10.4% with the first super group approach, while the improvement increases to 6.8%∼12.7% with the second approach. However, as can be observed from Fig.11(b), adopting the second approach would incur 11%∼48% additional read operations, while such overhead is trivial when using the first approach. This is because more page accesses are required to form super groups for victim selection under the second approach. In addition, as the second approach insists in adopting the super group as the basic unit for victim selection, it obviously incurs less write operations. Overall, the first super group approach achieves the best performance.

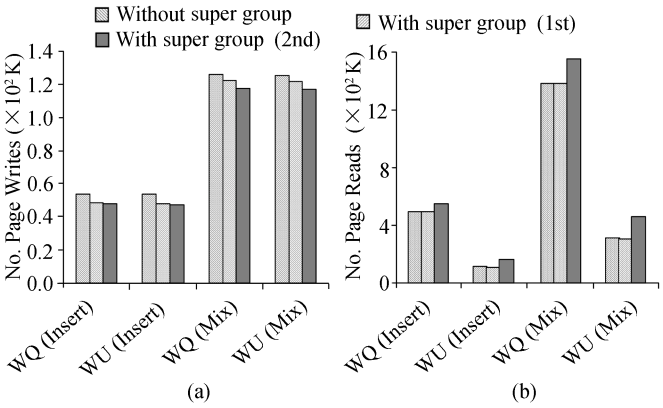


Fig.11. Impact of super group. (a) Number of page writes. (b) Number of page reads.

5.6.1 *Effect of Flash I/O Feature*

In previous experiments, we use the read/write speed presented in Table 1 (i.e., 80μs/page and 200μs/page) to calculate the overall I/O cost. As different flash storage media may have different ratios of read/write speed, in this subsection, we investigate the impact of read/write speed on the performance of the lazy-update algorithms.

We use λ to denote the ratio of read speed to write speed. The read speed and write speed are set at 80μs/page and 80λ μs/page, respectively. Based on the data in Fig.7, we plot the average I/O cost per request under different settings of λ in Fig.12. We observe that when the ratio of read/write speed is low, the performance of Big is slightly better than that of Cost. This is because Big incurs fewer read operations during the victim selection and therefore it performs relatively better when the read cost tends to dominate the overall I/O cost. On the other hand, when the ratio becomes higher, Cost outperforms Big. Moreover, their performance gap becomes larger when λ grows. The reason is that Cost incurs fewer write operations by taking into account rebalancing cases. When the write cost becomes more expensive, its advantage becomes more obvious.

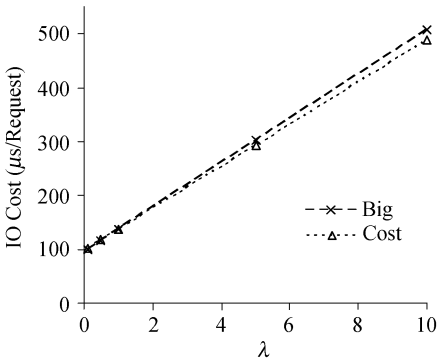


Fig.12. Impact of flash I/O feature.

6 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[5,10-11] 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 [12-13].

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[14]

and JFFS[15] were proposed. For DBMSs running on flash-based media, Lee and Moon[1] presented a novel design of data logging called in-page logging (IPL) to further improve the logging performance. Lee *et al.*[16] conducted a case study to investigate how the performance of conventional database applications is affected by the new flash-based disk.

Buffer management is an important research topic in database systems. As pointed out in [17], database processes must be cache-efficient to effectively utilize modern hardwares. Many flash-aware buffer management schemes have been proposed to exploit the inherent read-write asymmetry of flash devices. FAB (Flash-Aware Buffer)[18] groups cached pages located in the same erasure block, and adopts LRU replacement policy to maintain these groups. To enhance the random write performance, Kim and Ahn[19] proposed to use the in-device RAM to reorder the write requests into a desirable write pattern for the flash storage. CFLRU (Clean-First LRU)[20] is another flash-aware replacement policy based on the LRU algorithm. It addresses the asymmetry of flash I/Os by maintaining the LRU list into two regions (namely, working region and clean-first region), and giving priority to replacing clean pages over dirty pages. CFDC (Clean-First Dirty-Clustered)[21] further splits the clean-first region into a clean queue and a dirty queue, which avoids scanning extra dirty pages in the clean-first region of CFLRU. In contrast, this paper employs a lazy-update pool to reduce the number of write operations incurred by index maintenance. We note that the aforementioned flash-aware policies cannot be used to manage our lazy-update pool, since the lazy-update pool contains only update requests, which, once evicted, are to update the index nodes in the page cache (but not flushed to the flash storage). On the other hand, they can be employed for the management of the page cache, and hence are complementary to our work.

Research efforts have also been put into optimizing indexing 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 have been developed. Wu *et al.*[3] 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. To make BFTL work, 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[2] developed FlashDB, which uses a self-tuning B+-tree that dynamically adapts its storage structure to the workloads and storage devices. Lee *et al.*[22] introduced another flash-aware indexing scheme called IBSF to address the drawbacks of BFTL. The key ideas of IBSF are as follows: 1) all logs of index changes associated with a B-tree node are stored into a single page so that the in-memory NTT is no longer needed in IBSF; 2) redundant logs are eliminated from the buffer pool, which efficiently reduces the frequency of flushing the logs to the flash storage. We note that our proposed lazy-update method differs significantly from IBSF. Our method defers and groups update requests to the index, whereas IBSF does not. Instead, they update the index immediately upon receiving a request, but log the changes of index nodes for possible merge at a later time. As such, the design issues of IBSF and our method are totally different: while IBSF focuses on how to manage the logged changes, we focus on the commit policies of the lazy-update pool.

7 Conclusion

In this paper, we discussed the challenges of maintaining B+-tree on flash memory. To overcome the asymmetric read/write limitation, we proposed an optimized 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] Lee S W, Moon B. Design of flash-based DBMS: An in-page logging approach. In *Proc. the 2007 ACM SIGMOD International Conference on Management of Data* (*SIGMOD 2007*), Beijing, China, June 11-14, 2007, pp.55-66.

[2] Nath S, Kansal A. FlashDB: Dynamic self-tuning database for NAND flash. In *Proc. the 6th International Conference on Information Processing in Sensor Networks* (*IPSN 2007*), New York, USA, 2007, pp.410-419.

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

[4] Understanding the flash translation layer (FTL) specification. Technical Report, Intel Corporation, 1998, http://developer.intel.com.

[5] Kawaguchi A, Nishioka S, Motoda H. A flash-memory based file system. In *Proc. USENIX 1995*, New Orleans, USA, Jan. 16-20, 1995, pp.155-164.

[6] SmartMedia specification. SSFDC Forum, http://www.ssfdc.or.jp.

[7] Kim B, Lee G. Method of driving remapping in flash memory and flash memory architecture. United States Patent, No.6 381176, 2002.

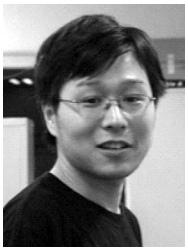
[8] Reiser H. Reiser file system. http://www.namesys.com, 1997.

[9] Oracle Corporation. Oracle 10g. http://www.oracle.com/database/index.html, 2003.

[10] Chang L P, Kuo T W, Lo S W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *Trans. Embedded Computing Sys.*, 2004, 3(4): 837-863.
[11] Kim H J, Lee S G. A new flash memory management for flash storage system. In *Proc. the 23rd International Computer Software and Applications Conference* (*COMPSAC 1999*), Phoenix, USA, Oct. 19-26, 1999, p.284.
[12] Chang L P, Kuo T W. An efficient management scheme for large-scale flash-memory storage systems. In *Proc. the 2004 ACM Symposium on Applied Computing* (*SAC 2004*), Nicosia, Cyprus, Mar. 14-17, 2004, pp.862-868.
[13] Chang Y H, Hsieh J W, Kuo T W. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proc. the 44th Annual Conference on Design Automation* (*DAC 2007*), San Diego, USA, June 4-8, 2007, pp.212-217.
[14] Embedded Debian, YAFFS: A nand-flash file system. Aleph one ltd., http://www.aleph1.co.uk/yaffs, 2002.
[15] Woodhouse D. JFFS: The journalling flash file system. In *Proc. the Ottawa Linux Symposium*, Ottawa, Canada, July 25-28, 2001, pp.177-182.
[16] Lee S W, Moon B, Park C, Kim J M, Kim S W. A case for flash memory SSD in enterprise database applications. In *Proc. the 2008 ACM SIGMOD International Conference on Management of Data* (*SIGMOD 2008*), Vancouver, Canada, June 10-12, 2008, pp.1075-1086.
[17] Cieslewicz J, Mee W, Ross K A. Cache-conscious buffering for database operators with state. In *Proc. the Fifth International Workshop on Data Management on New Hardware* (*DaMoN 2009*), Rhode Island, USA, June 28, 2009, pp.43-51.
[18] Jo H, Kang J, Park S, Kim J, Lee J. FAB: Flash-aware buffer management policy for portable media players. *Trans. Consumer Electronics*, 2006, 52(2): 485-493.
[19] Kim H, Ahn S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proc. the 6th USENIX Conference on File and Storage Technologies* (*FAST 2008*), Berkeley, USA, 2008, pp.1-14.
[20] Park S, Jung D, Kang J, Kim J, Lee J. CFLRU: A replacement algorithm for flash memory. In *Proc. the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (*CASES 2006*), Seoul, Korea, Oct. 23-25, 2006, pp.234-241.
[21] Ou Y, Härder T, Jin P. CFDC: A flash-aware replacement policy for database buffer management. In *Proc. the Fifth International Workshop on Data Management on New Hardware* (*DaMoN 2009*), Rhode Island, USA, June 28, 2009, pp.15-20.
[22] Lee H S, Park S, Song H J, Lee D H. An efficient buffer management scheme for implementing a B-tree on NAND flash memory. In *Proc. the 3rd International Conference on Embedded Software and Systems* (*ICESS 2007*), Berlin, Heidelberg, 2007, pp.181-192.

Sai Tung On is an MPhil candidate in the Department of Computer Science, Hong Kong Baptist University. His research interests include indexing, query processing and transaction management on new storage media.

Haibo Hu received the Ph.D. degree in computer science from Hong Kong University of Science and Technology in 2005. He is an assistant professor in the Department of Computer Science, Hong Kong Baptist University. Prior to this, he held several research and teaching posts at HKUST and HKBU. His research interests include mobile and wireless data management, location-based services, and privacy-aware computing. He has published over 20 research papers in international conferences, journals and book chapters. He is also the recipient of many awards, including the ACM-HK Best Ph.D. Paper Award and the Microsoft Imagine Cup.



Yu Li is a Ph.D. candidate in the Department of Computer Science, Hong Kong Baptist University. His research interest lies in data management on flash-based storage media.

Jianliang Xu received the B.Eng. degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and the Ph.D. degree in computer science from Hong Kong University of Science and Technology in 2002. He is an associate professor in the Department of Computer Science, Hong Kong Baptist University. He was a visiting scholar in the Department of Computer Science and Engineering, Pennsylvania State University, University Park. His research interests include data management, mobile/ pervasive computing, wireless sensor networks, and distributed systems. He has published over 80 technical papers in these areas. He has served as a vice chairman of the ACM Hong Kong Chapter. He is a senior member of the IEEE.