Garbage Collection for Multi-version Index on Flash Memory

Kam-Yiu Lam*, Jiantao Wang*, Yuan-Hao Chang‡, Jen-Wei Hsieh§,
Po-Chun Huang‡, Chung Keung Poon* and Chun Jiang Zhu*
*Department of Computer Science, City University of Hong Kong
Email: {jiantwang2,chunjizhu2}@student.cityu.edu.hk, {cskylam,csckpoon}@cityu.edu.hk
‡Institute of Information Science, Academia Sinica, Taiwan R.O.C. Email:{johnson, aufbu}@iis.sinica.edu.tw
§Department of Computer Science and Information Engineering, NTUST Email: jenwei@mail.ntust.edu.tw

***Abstract*—In this paper, we study the important performance issues in using the *purging-range query* to reclaim old data versions to be free blocks in a *flash-based multi-version database*. To reduce the overheads for using the purging-range query in garbage collection, the *physical block labeling (PBL)* scheme is proposed to provide a better estimation on the purging version number to be used for purging old data versions. With the use of the *frequency-based placement (FBP)* scheme to place data versions in a block, the efficiency in garbage collection can be further enhanced by increasing the deadspans of data versions and reducing reallocation cost especially when the spaces of the flash memory for the databases are limited.**

***Index Terms*—Multi-version Index, Multi-version Data, Real-time Data, Flash-based Embedded Database Systems**

I. Introduction

Due to the unique performance characteristics of flash memory as compared with disk storage systems [2], managing an index on flash memory could have serious impacts to the performance of the flash memory system. One of the important issues is garbage collection of indexed data. An update to the index may trigger a sequence of updates to the flash memory, and result in generation of a number of invalid pages [3]. The invalid pages can be reused only after garbage collection to reclaim them to be "free" blocks [2, 5].

Although various efficient methods have been proposed for garbage collection in flash memory, the garbage collection for indexed data has been greatly ignored in previous research works. Most of the proposed garbage collection mechanisms are mainly concentrated on non-structured data with the performance objectives of minimizing the reallocation cost of valid data pages in a block or balancing the wearing levels of the blocks as they have limited number of endurances [4, 5, 7, 8]. However, reallocating an indexed page could have the cascading update problem and generate more invalid pages. In this paper, we study how to perform garbage collection in flash memory which maintains a *multi-version B+-tree index* (MVBT) [1]. Since *delete* operations are not common in embedded multi-version databases, i.e., insert only databases, how to efficiently select data versions for garbage collection and how to place data versions into a physical block could have serious performance impacts to the efficiency of the adopted garbage collection mechanism.

II. Multi-version B+-Tree (MVBT)

In [1], Becker *et al.* proposed the multi-version B+-tree (MVBT) for indexing multi-versions of data items in a database. As shown in Fig. 1, MVBT is a directed acyclic graph of B+-tree nodes maintaining multiple B+-tree root nodes to partition the versions of data items such that each B+-tree root stands for a range of data versions.

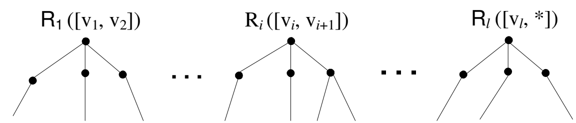

Fig. 1: Root partitions of an MVBT index.

In an MVBT index, each node consists of a number of entries and each entry has the form <*key, in_version, del_version, info*>. The *info* part maintains a pointer and other information for identifying the entry. For a leaf entry, the pointer points to the data version with the corresponding *key* value, while for an inner entry, the pointer points to the next level node. The *in_version* and *del_version* denote the range of version numbers of the data version. The values of the version numbers are monotonically increasing.

If a new data version is inserted, a new entry with *in_version* = v_p and *del_version* = "$*$" will be created, where v_p is the present version number of the index. v_p is also a monotonically increasing integer. When a data version is updated (*i.e.*, a new version is created), the entry that points to the previous data version will be updated by replacing "$*$" of *del_version* with the current value of the present version number, v_p. The entries with "$*$" as their *del_version* are called *live entries*; otherwise they are *dead entries*. Similarly, the corresponding data versions are called dead versions and live versions, respectively. Thus, a live version means that it is the latest version of a data item. A version is said to be of *version* i if its range of versions contains i, *i.e.*, *in_version* $\leq i <$ *del_version* (if *del_version* $\neq$ "$*$") or *in_version* $\leq i$ (if *del_version* $=$ "$*$").

III. The Update Problem in Garbage Collection

Although MVBT is an efficient index structure for disk-resident databases, managing an MVBT on flash memory could incur heavy update cost and generate a lot of invalid pages in processing update operations. Here we use the example shown in Fig. 2 to illustrate the update problem in MVBT and show how it affects the cost in garbage collection. In the example, to simplify the discussion, it is assumed that each page contains an index node or a data version.

Fig. 2(a) shows the initial structure of an example MVBT index. It consists of a root node R and two child nodes of

978-3-9815370-2-4/DATE14/©2014 EDAA

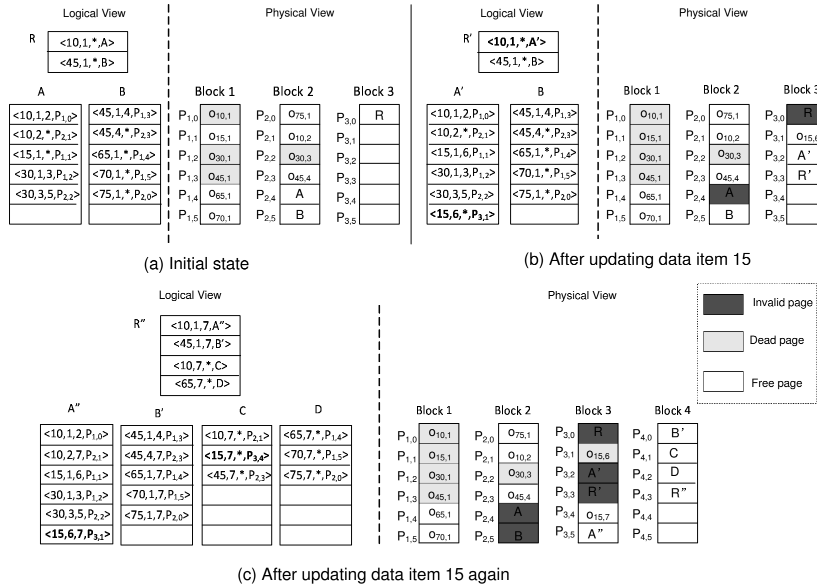

Fig. 2: The motivational example of the problem if index updates in flash memory

R, A and B. If an update operation arrives to update data item 15, the entry $<15, 1, *, P_{1,1}>$ for the latest version of 15 is marked as dead, and then a new entry $<15, 6, *, P_{3,1}>$ is created and inserted into node A, as shown in Fig. 2(b). Because of the use of out-place update, the modified node $A^{'}$ ($A \rightarrow A^{'}$) is written into a new free page. Similarly, node R is written to a new free page as the entry $<10, 1, *, A>$ has to be updated to be $<10, 1, *, A^{'}>$.

If another update operation arrives to update data item 15, the entry $<15, 6, *, P_{3,1}>$ that points to the latest version of data item 15 is marked as dead, and a new entry $<15, 7, *, P_{3,4}>$ is allocated to point to the newly created data version. However, node $A^{'}$ is full and the new entry cannot be inserted. Therefore, the live entry in node $A^{'}$ and the new entry $<15, 7, *, P_{3,4}>$ are merged with the live entries in the sibling node B. However, there are totally 6 live entries after merging, and a *strong version overflow* occurs, leading to a key split [1]. (It is assumed that the maximum number of entries is 5.) Thus, nodes C and D are created (Fig. 2(c)). In total, four new nodes $A^{''}$ ($A^{'} \rightarrow A^{''}$), $B^{'}$ ($B \rightarrow B^{'}$), C, and D have to be written into four different free pages in the flash memory to process the second update operation. Note that if the tree consists of more levels, the number of node updates will be more. As can be observed in Fig. 2(c), blocks 2 and 3 contain some valid and invalid pages. If they are reclaimed in garbage collection, the data contained in the valid pages need to be reallocated. However, reallocating the data versions and index nodes triggers the updates of higher level nodes (e.g. node A'') and the total reallocation cost could be very expensive if the tree has more levels.

IV. Garbage Collection of Indexed Data

In a flash memory chip, a free block is a physical block that has been erased and is maintained in a *free block pool*. If the number of blocks (T_{free}) in the free block pool is lower than a pre-defined threshold (T_{GC}), the garbage collection process will be invoked to reclaim the used blocks until the number of free blocks (T_{free}) is more than another threshold (T_{normal}) with $T_{normal} > T_{GC}$. The main issue in garbage collection is how to select the physical blocks (called *victim blocks*) for reclaiming to be free blocks with the objectives of minimizing the garbage collection cost and balancing the wear levels of the blocks.

In this paper, instead of selecting the blocks with the smallest number of valid pages for garbage collection to minimize the reallocation cost, we propose to reclaim "old" data versions to be free blocks to minimize the reallocation cost. Thus, the problem is how to efficiently select "old" data versions to be converted to invalid versions so that the blocks storing them can be reclaimed to be free blocks.

A. Reclaiming Data Versions and Index Nodes

In a flash memory chip maintaining a multi-version index, the pages of a physical block may be used to store data versions (called *data pages*) or index nodes (called *index pages*). An index page becomes invalid after an update due to out-of-place update. Since delete operations are not common to many embedded multi-version databases [6], a data version becomes invalid only after a *purging operation* to convert "dead" data versions to be invalid. Then, the pages storing the invalid data, called *invalid data page* could be reclaimed. Therefore, the garbage collection process for reclaiming data pages can be divided into two sub-processes: *purging* and *erasure*. The erasure process is to reinitialize the pages in a block such that they can store new data.

In selecting data versions for purging, the purging process starts from the data version with the smallest $del_version$ as it has the lowest probability to be accessed by application queries. A smaller $del_version$ means that the data version was *dead* earlier. In performing the purging process, a purging version number $v_{th}(t)$ is specified in each invocation at time t. The data versions with $del_version$ smaller than $v_{th}(t)$ are purged (converted to be invalid). Similar to the present version number v_p, $v_{th}(t)$ is also a monotonically increasing integer.

After a data version is purged, the corresponding index entry will become invalid. If all the entries of an index node are invalid, the index node will become invalid, too.

An important concern in setting the value for $v_{th}(t)$ in each invocation of the purging process is to maximize the *deadspans* of data versions. The deadspan of a data version $o_{k,i}$ is defined as the difference between the present version number v_p when it is purged and $del_version$ of $o_{k,i}$.

Maximizing the deadspans of data versions can increase the probability that application queries can find their required data versions. Even a data version is dead, it is still maintained. It is converted to be invalid and be reclaimed only when the free space in the flash memory is insufficient. Therefore, the purging version number $v_{th}(t)$ should be increased only if the number of reclaimed blocks cannot meet the threshold requirement (T_{normal}).

Minimizing the reallocation cost in garbage collection and maximizing the deadspans of data versions are conflicting goals in garbage collection. If a flash memory block contains both valid and invalid data pages after purging, we may increase the purging version number to convert the remaining data versions to be invalid. This will reduce the reallocation cost but the deadspans of the data versions will be reduced. On the other hand, we may use a smaller purging version number to reclaim the block which contains both valid and invalid pages. This will increase the deadspans of data versions but the reallocation cost will be higher since the valid data versions need to be reallocated to other free pages.

Algorithm 1 Garbage Collection ($v_{th}(t)$)

```
1:  if T_free ≤ T_GC then
2:      Set the GC flag;
3:      while the GC flag is set do
4:          if there exists a block with all pages are dead then
5:              Select block with the smallest del_version as victim
                block.
6:          else
7:              Select block with the smallest in_version as victim
                block.
8:          end if
9:          if all pages in the victim block are invalid then
10:             Erase the victim block.
11:             Add the victim block to the free block pool.
12:             T_free++.
13:         else
14:             Increase v_th(t) by f.
15:             Invoke purging-range query Q(−∞, v_th(t), ∗).
16:         end if
17:         if T_free ≥ T_normal then
18:             Clear the GC flag;
19:         end if
20:     end while
21: end if
```

B. Purging-Range Query

For efficient execution of the purging process, we may implement it as a *purging-range query*. A purging-range query Q can be specified as $Q(v_l, v_h, key)$, where key = "$*$" such that it accesses to all data items maintained in the database. v_h is set to be the purging version number $v_{th}(t)$ at the current time t, and $v_l = -\infty$. In processing a purging-range query Q, if the $del_version$ of a version is smaller than $v_{th}(t)$, it will be converted to be invalid.

After the purging process, if all the pages in a block are invalid, the block will be reclaimed by the erasure process without any reallocation cost. If the number of free blocks (T_{free}) is still less than T_{normal}, the purging-range query may be re-executed after $v_{th}(t)$ is advanced by a value f, *i.e.*, $v_{th}(t) = v_{th}(t-1) + f$. Algorithm 1 summarizes the steps for the garbage collection.

The benefit of using a purging-range query for selecting data versions to be purged is that it can reuse all the database facilities supported for range queries from applications to identify the to be purged data versions efficiently. Another important benefit of using this scheme for reclaiming old versions is that the wear levels of the blocks can be better balanced as older data versions are in general purged and reclaimed first. Older versions mean that they have been stayed in a page longer. Reclaiming them can increase their wear levels.

C. Determination of Purging Version Number

As shown in Algorithm 1, in each invocation, the purging version number $v_{th}(t)$ is increased by f to the previous purging version number. The number of invocations (as well as the processing cost) of the purging-range query in each garbage collection depends on the chosen value for f. A small value of f increases the number of invocations and the processing cost of the purging-range query while a large value of f may purge some young versions that are unnecessary to be purged, leading to decrease in *deadspans* of data versions.

To provide a better way to determine the purging version number, a *physical block labeling (PBL)* scheme is proposed. As shown in Fig. 3, an additional attribute, called *block label*, is added in each node, *e.g.*, G, to indicate the physical blocks that store the data versions indexed by the node G (*i.e.*, data blocks) and the index nodes of the sub-tree rooted at G (*i.e.*, index blocks). For example, the *block label* of leaf node A is $\{B_0, B_2\}$, which means that the data versions indexed by the leaf node A and leaf node A itself are stored in blocks B_0 and B_2. The *block label* is updated along with updates of the index tree until the resided node becomes dead.

With *PBL*, the garbage collector searches from the tree root with the smallest $del_version$, *e.g.*, R_1, until a node G is found such that the total number of physical blocks, including the blocks indicated by the root nodes $R_1, \cdots, R_{i-1}$ and G, is larger than or equal to $T_{normal} - T_{free}$. Then, $v_{th}(t)$ is set as *del_version* of the entry that points to node G for execution of the purging-range query. In this way, the purging-range query may be executed just once to obtain the required number of reclaimed blocks.

D. Data Placement in Flash Memory

1) The Sequential Placement (SQ) Scheme: One of the critical issues that could seriously affect the efficiency of the proposed garbage collection mechanism including *PBL* is the placement of data versions and index nodes in the blocks. If the data versions under a sub-tree are distributed in different blocks, the estimation using *PBL* could be inaccurate.

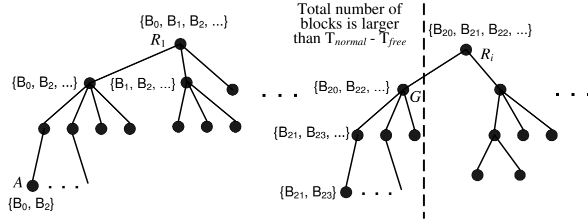

Fig. 3: PBL to the index tree

Block 1	
<1,3>	$O_{1,1}$
<2,2908>	$O_{2,2}$
<3,5>	$O_{1,3}$
<4,6>	$O_{3,4}$
<5,7>	$O_{1,5}$
<6,8>	$O_{3,6}$
<7,9>	$O_{1,7}$
<8,10>	$O_{3,8}$

Block 2	
<9,11>	$O_{1,9}$
<10,12>	$O_{3,12}$
⋮	⋮

. . .

Block n	
<2908,*>	$O_{2,2908}$
⋮	⋮

(a) R_{vr} = 2905

Block n+1	
<1,3>	$O_{1,1}$
<2,4>	$O_{3,2}$
<3,5>	$O_{1,3}$
<4,6>	$O_{3,4}$
<5,7>	$O_{1,5}$
<6,8>	$O_{3,6}$
<7,9>	$O_{1,7}$
<8,10>	$O_{3,8}$

Block n+2	
<9,*>	$O_{1,9}$
<10,*>	$O_{3,10}$
⋮	⋮

(b) R_{vr} = 7

Note that two elements of <*f*, *s*> in each page denote the *in_version* and *del_version* of the resided data version in that page, respectively.

Fig. 4: An example of a version range R_{vr}

A simple method for placing data versions and index nodes in flash memory is to use separate physical blocks for each of them such that once a block, called *data block*, has been assigned to store a data version, it will not be used for storing index nodes, and vice versa. The blocks which store the index nodes are called *index blocks*. Once a new data (either data version or index node) needs to be stored into a new free block, a free block with the *smallest* wear level number in the free block pool will be selected. The wear level number of a block increases with the number of times that block is allocated to store data. Selecting the free block with the smallest wear level can balance the wear levels of the blocks in the flash memory. The new data will be put into the first page of the selected free block and the following pages in the block (both index block and data block) will be allocated sequentially until all its pages have been allocated. We call this scheme as the *sequential placement (SQ)*.

Placing data versions sequentially in a data block may reduce the version range for a data block and improve the accuracy in estimation using *PBL* as the creation times of the data versions are similar. However, the efficiency of *SQ* in reclaiming dead versions depends highly on the update frequencies of the data versions in the block.

For a given purging version number at time t, $v_{th}(t)$, the number of data blocks that could be reclaimed depends on the ranges of version numbers of the data versions in the blocks. For a data block containing all dead data versions, we define a *version range* (R_{vr}) as the difference between the maximum $del_version$ and the minimum $del_version$ of the data versions in the block. If R_{vr} is smaller, the data block can be reclaimed by a smaller purging version number. For example, as shown in Fig. 4 (a), the version range R_{vr} of block 1 is 2905. In block 1, the update frequency of data item o_2 is much lower than the other data items in the block, leading to data version $o_{2,2}$ having a much larger $del_version$ of 2908. However, as shown in Fig. 4(b), the version range R_{vr} of block $n+1$ is only 7 as all the data items in the block having similar update frequencies. Therefore, a larger value of $v_{th}(t)$ need to purge the data block 1 in Fig. 4(b) in order to reclaim it with zero reallocation cost.

2) The Frequency-Based Placement (FBP) Scheme: To improve the efficiency in purging, we propose the *frequency-based placement (FBP)* scheme for placing data versions into a data block. Similar to *SQ*, separated blocks are used for placing data versions and index nodes. The performance objective of *FBP* is to maximize the deadspans of data versions and reduce the reallocation cost. It is assumed that the update frequency for each data item is given. Following their update frequencies, they are classified into L categories, $S_1, S_2, \cdots, S_l, \cdots, S_L$, with data items in the same category having update frequencies within a range of pre-defined update frequencies for the category. For the sake of simplicity, the K data items are evenly distributed into L categories. For $1 \leq l < L$, $S_l = \{o_{\lceil \frac{K}{L} \rceil \cdot (l-1)+1}, \cdots, o_{\lceil \frac{K}{L} \rceil \cdot l}\}$; while $S_L = \{o_{\lceil \frac{K}{L} \rceil \cdot (L-1)+1}, \cdots, o_K\}$. A data block is used for storing the data versions of data items belonging to the same category sequentially.

Since data versions with similar update frequencies are placed sequentially in a data block, the version range (R_{vr}) of the block can be limited by the value of the range of update frequencies of the category. This can increase the probability to purge the data block using a smaller purging version number. The data placement policy of *FBP* is the same as *SQ*, i.e., the free block with the smallest wear level number will be selected for placing new data.

V. Conclusions

In this paper, we propose the *frequency-based placement (FBP)* scheme to improve the efficiency in garbage collection. To reduce the overhead for processing the purging-range query, we propose the *physical block labeling (PBL)* scheme to provide a better estimation on the purging version number to be used for purging old data versions. The efficiency of the propose schemes is illustrated by simulation experiments. An important future work is to design efficient techniques to reduce the number of updates in managing the multi-version index and integrate it with the proposed garbage collection mechanism.

References

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. an Asymptotically Optimal Multiversion B-tree. *VLDB Journal*, 5:264–275, 1996.

[2] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[3] D. Kang, D. Jung, J. Kang, and J. Kim. μ-tree: an Ordered Index Structure for NAND Flash Memory. In *Proceedings of the International Conference on Embedded Software*, pages 144–153, 2007.

[4] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash Memory Based File System. In *Proceedings of the USENIX Technical Conference*, pages 155–164, 1995.

[5] O. Kwon, K. Koh, J. Lee, and H. Bahn. FeGC: an Efficient Garbage Collection Scheme for Flash Memory based Storage Systems. *Journal of Systems and Software*, 84(9):1507–1523, 2011.

[6] E. Lee and S. Seshia. *Introduction to Embedded Systems-A Cyber-Physical Systems Approach*. 2011.

[7] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[8] G. Xu, Y. Liu, X. Zhang, and L. M. Garbage Collection Policy to Improve Durability for Flash Memory. *IEEE Transactions on Comsumer Electronics*, 58(4):1232–1236, 2012.