A New Flash-based B+-Tree with Very Cheap Update Operations on Leaf Nodes

Sungchae **L**im

***Abstract*—**Recently, as the price per bit is decreasing at a fast rate, flash memory has been considered to be an alternative storage of large-scale data-centric systems. Although flash shows off its high speeds of page reads, it also have performance concerns about pool performance of random writes. Therefore, it is crucial to get a way to efficiently update a B+-tree in flash storage. In this light, we propose a new flash B+-tree that stores updated versions of leaf nodes in sibling-leaf blocks (SLBs) so that garbage collection is efficiently performed in the unit of SLB. Since the versions of leaf nodes can be directly accessed from their virtual parent nodes, the search speeds does not deteriorate, differently from other earlier B+-trees. To verify the performance advantages, we use a cost model fitted to usual operations performed in the B+-tree.

***Keywords*—**B+-tree index, buffer pool, database, flash memory, solid-state drive.

I. Introduction

During the past decades, we have seen drastic advances in the technology of digital storage systems. Especially, the NAND-type flash memory has drawn much attention thanks to its salient advantages over the traditional hard disk drives (HDDs) [1, 5, 8, 12]. Since flash memory provides desirable H/W features such as robust shock resistance as well as low power consumption, it has gradually superseded HDDs in the case of low-end computing devices, namely, lab tops and smart phones [1, 2]. Moreover, as the price per bit is recently decreasing at a very fast rate, there is a growing demand for flash-based DBMSs that are able to successfully process harsh workloads of enterprise-scale transaction processing system [3, 5, 9]. In this light, many researches have been conducted to develop a flash-aware B+-tree that can index efficiently a large volume of records on flash storage such as the solid state drive (SSD) [4, 11-16].

Because the B-tree index was invented for its use in HDDs, there is a concern that the B-tree index may suffer from poor stability or performance degradation because of any different I/O feature of flash memory from HDDs [11-16]. As a matter of fact, the absence of in-place updating in flash's functionalities has been regarded as a major technical disadvantage to be tackled [1, 12, 13].

Sungchae Lim is with Dept. of Computer Science, Dongduk Women's University, Seoul, South Korea

The NAND-type flash storage usually partitions its memory space into equal-size blocks. Each block contains 64 or 128 pages of size 0.5KB within it, and a page is used as the smallest unit of data writes [1, 5]. If bit cells in a page *P* have been electronically charged for a data write on *P*, then they should be cleaned before page *P* can be updated with new data. In the case of flash SSD, such a block erase has its smallest unit of a block. Unfortunately, the different unit sizes of data writing and a block erase operation significantly enlarge the cost for an in-place update of a page. This is because an in-place update of a page can be done with a single block erase and a number of page copies that are required to retain other neighboring pages residing in the erased block [4, 10]. Because of the enormous overhead for the in-place update, the in-place updates are not implemented in flash storage. Rather than, an update request on page *P* is processed through a write to any other clean page *X* and changing of physical address of *P* to that of *X*. To solve those problems, we proposed a new flash B+-tree.

The rest of this paper is organized as follows. In Section 2, we give some background knowledge about the flash memory. Next, we present our flash B+-tree in Section 3 and conclude this paper in Section 4.

II. Backgrounds

The B+-tree has been one of the most popular index schemes in disk-based DBMSs over fast decades. Since the B+-tree has a majority of updates within its leaf level and the leaf nodes seems to be scattered across storage, a flash B+-tree seems to inevitably suffer from frequent full merges in the presence of very frequent key updates. To solve such an inherent problem of the B+-tree stored in flash, some ideas are proposed to decrease full merges during the garbage collection time [4, 12, 13, 15]. The flash B+-trees proposed in [13, 15] take the same approach in that key updates are not reflected in real-time on the data of leaf nodes. Instead, the corresponding log data is recorded in other tree levels above leaf level. When the amount of such log data of update operations is estimated to be large enough, the data are actually flushed into leaf nodes in storage through sequent writes, which are cheaper than random updates.

In [13, 15], those update log is incrementally stored in the region of a sub-tree that is composed of the root of a B+-tree and its internal nodes. Since the higher-level sub-tree is emptied at the initial time, it can keep many of update operations until they are reflected actually on leaf nodes. As a result, the use of such emptied sub-tree region can reduce the number of updates at the

level of leaf nodes. When the nodes of the update-absorbing sub-tree becomes full because of key inserts and delete in the tree, the update log stored in it are flushed to the B+-tree index by restructuring the whole leaf nodes in batch mode. At that time, the sub-tree region is initialized again to record new update operations. Unlike that, [13] does not initially allocate any separated sub-tree used for logging updates in the tree. Instead, [13] they can dynamically adjust the sizes of internal nodes based on a simple cost model, which is utilized to decide when the log update operation should be written to shrink its node size. Note that that the increase size of internal nodes slows key searches and leads to more consumptions of the buffer pool space. Those previous schemes with update log on internal nodes can replace many of out-of-place updates with writes to clean pages, although the key search can be slowed due to extra overhead time taken to look up update log in internal node encountered. Besides somewhat poor search times, the schemes [13, 15] above have a problem regarding the concurrency control of the B+-tree. Under the traditional B+-tree concurrency control, the locking protocol called lock-coupling is applied to guarantee a consistent B+-tree state among concurrent B+-tree processes including searchers and updaters.

To avoid problems above, other algorithms [17] adopt an approach such as the conventional redo mechanism. In these algorithms, when a page (i.e., a node) is updated, the redo log record for that update operation is saved within a log area that is reserved in every flash block. The log area for saving redo records is composed of a portion of pages located at the tail of the block. The pages preceding the log area save the original images of nodes stored in the block. To update a node *N* in a block *X*, an associated redo log record is written to the log area of *X*. Corresponding, to get the latest version of node *N* correctly, [17] read both a node saving the original image of *N* and the whole redo log records in block X. If any redo record(s) of *N* is found, the latest version of nod is made through an appropriate redo action on the original image of node. Then, the latest version of *N* is cached in the buffer pool. By caching nodes in the buffer pool, we can prevent the excessive overheads for reading redo records and conducting redo actions. When the log area of block X is filled with redo records, block *X* is initiated to save the latest versions of pages and reclaim the log area.

III. Proposed Method

When it comes to update operations in a B+-tree, most of them seem to arise at the leaf level [17]. Note that nodes at other different levels, i.e., internal nodes, are updated only when tree reconstruction is needed for dealing with overflow or underflow in any leaf node. Since a populated B+-tree has a huge number of leaf nodes that are widely scattered across storage space, it is the case that updates in the B+-tree are usually random page updates. Therefore, a B+-tree with frequent update operations may give birth to undesirable hiccupped I/O throughput of its flash storage because of frequent full merges in FTL. For this reason, the B+-tree needs to be redesigned to efficiently reduce the amount of leaf node updates being performed by FTL, thereby preventing drastic deterioration of flash performance caused by full-merge overheads.

To this end, we enable the B+-tree to conduct out-of-place updates for leaf nodes without the help of FTL. If the B+-tree is responsible for activities of out-of-place updates, then it can easily suffer from excessive I/O overheads paid for keeping the mapping information between the original page ID of any updated node and page IDs of its new versions. To eliminate the need of bookkeeping varying page IDs of an individual node with updates, our B+-tree stores sibling nodes pertaining to the same parent in the boundary of the sibling-leaf block (SLB). Here, a single SLB has a size of one flash block, and part of it is managed as clean pages for saving new versions of leaf nodes in SLB. The clean pages are located at the tail of SLB and a leftmost one among them is allocated to save a new version of an updated leaf node.

Let us assume that the original PID (page ID) of the updated leaf node (say *N*) is *X*, and the PID of the new version of *N* is *X'*, respectively. Since the proposed B+-tree does not rely on FTL, the parent node of *N* needs to be updated to correctly point to the new version with PID *X'*. Note that if node *N* is updated in the help of FTL, then there is no need to update the parent of *N* because the PID for node *N* is not changed. This is because FTL records address mapping information between logical PID and its actual location so that out-of-updates of a page can be hidden. Since we cannot hide the change of physical locations of node *N*, it is need to modify the associated pointer field so that it points to the new version of *N*.

To solve that problem of burden to trace the varying up-to-date PIDs of leaf nodes with updates, we do not save physically the parent of *N* on flash storage. Instead, we use a way to dynamically build the up-to-date index entries of that parent from its child nodes in SLB. That is possible by using the maximum fields of sibling leaf nodes as the key fields of their parent as in [6, 7]. More specifically, we retrieve the whole data of SLB and uses keys of *N*'s parent node using the values of the maximum fields of *N* and its in-SLB sibling nodes. Therefore, if a search process is about to access a certain level-2 node *P* and that node is not currently cached in the buffer pool, our buffer pool manager reads the whole SLB containing *P*'s child nodes and cache node *P* made from its child nodes. From that time, the key search process can successfully find its search path from *P* to the target leaf node. Of course, the index entries of *P* have been already created and cached in the buffer pool, such a building of P is not necessary.

In the meanwhile, if a certain leaf node *L* is updated and written to any clean page, then the changed address of *L* is saved only on the buffer-cached image of *P*. If the cached image is evicted from the buffer pool according to the buffer replacement algorithm, then our buffer pool manager never flushes physically the data of *P* into flash storage. Since the correct data of node P can be regenerated later, if needed, the buffer manage just remove the node P from its buffer pool without any write for

it. We call the level-2 node such as node *P* the *virtual node*, because it does not exist physically on storage.

Since the creation of a virtual node inevitably requires the whole read of sibling leaf nodes of SLB, the I/O cost for accessing a virtual node becomes bigger than the I/O cost for accessing an ordinary internal node stored on storage. If our B+-tree pays too much I/O costs for creating virtual nodes during the time of key searches, therefore, such overhead may outweigh the I/O benefits obtainable from reduced full merges on the side of FTL. However, since the buffer pool of the DBMS can cache internal nodes of a B+-tree and the read speed of a block is so fast in flash, the overhead from the use of virtual nodes is negligibly small. Instead, we can achieve more performance gains from the reduction of full merges during garbage collection in flash storage.

To demonstrate the performance advantages of the proposed flash B+-tree, we develop a cost model based on SSDs in the market. With the cost model, we compared the average search times between our B+-tree and others. From the simulation results, it is testified that the proposed B+-tree can achieve good I/O throughput of flash storage as well as fast search times. As expected prior to the simulations, our performance benefits are mainly originated from the very low update costs in the proposed flash B+-tree. If we can cache the whole internal nodes including virtual nodes, more performance advances can be maximized in our B+-tree. Detail about that will be presented at the conference site

IV. Conclusions

For the disk based B+-tree, a node update can be cheaply done by modifying its associated disk page in place. However, once the B+-tree should be stored on flash memory, the traditional algorithms of the B+-tree seem to be useless due to the prohibitive cost of in-place updates on flash memory. For this reason, the earlier schemes for flash memory B+-trees usually take an approach that saves data of real-time updates into extra temporary storages. Although that approach can easily evade frequent in-place updates in the B+-tree, it can suffer from a waste of storage space and prolonged search times. Particularly, it is not possible to process range searches by using a link connection chaining leaf nodes. To resolve such problems, we devise a new scheme using the sibling-leaf blocks (SLBs). Then, we also design some algorithms that are used to perform the node update in the unit of a SLB and preserve tree consistency. From this, our scheme can improve the storage utilization and the efficiency of key search operations.

References

[1] A. Leventhal, "Flash Storage Memory", Communications of the ACM, 51(7), 2008.
http://dx.doi.org/10.1145/1364782.1364796

[2] J. Gray and B. Fitzgerald, "Flash Disk Opportunity for Server-applications", http://www.research,Microsoft.com/`gray, 2007.

[3] Sungup Moon, Sang-Phil Lim, Dong-Joo Park, and Sang-Won Lee, "Crash Recovery in FAST FTL", LNCS Vol. 6399, pp. 13-22, 20111.

[4] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang, "An Efficient B-tree Layer Implementation for Flash-memory Storage Systems", ACM Transactions on Embedded Computing Systems, Vol. 6(3), 2007
http://dx.doi.org/10.1145/1275986.1275991

[5] S. W. Lee, B. Moon, and C. Park, "Advances in Flash Memory SSD Technology for Enterprise Database Applications", In Proc. of SIGMOD, 2009
http://dx.doi.org/10.1145/1559845.1559937

[6] Yat-Sang Kwong and Derick Wood, "A New Method for Concurrency in B-Trees", IEEE Transactions on Software Engineering 8(3), pp.211-222, 1982.
http://dx.doi.org/10.1109/TSE.1982.235251

[7] Jan Jannink, "Implementing Deletion in B+-Trees", In Proc. of SIGMOD, 1995.
http://dx.doi.org/10.1145/202660.202666

[8] Stepahan Baumann, et al., "Flashing Databases: Expectations and Limitations", In Proc. of DaMon '10, 2010.
http://dx.doi.org/10.1145/1869389.1869391

[9] Yongkun Wang, Kazuo Goda, and Masaru Kitsuregawa, "Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems", In Proc. of DEXA, pp. 777-791, 2009.
http://dx.doi.org/10.1007/978-3-642-03573-9_66

[10] Gap-Joo Na, Sang-Won Lee, and Bongki Moon, "Dynamic In-Page Logging for B+-tree Index", IEEE Trnas. on Knowledge and Data Engineering, Vol. 24(7), pp. 1231-1243, 2012
http://dx.doi.org/10.1109/TKDE.2011.32

[11] Sangwon Park, Ha-Joo Song, and Dong-Ho Lee, "An Efficient Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory", In Proc. of ICESS '07, 2007.

[12] D. Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi, "Tree Indexing on Solid State Drives", In Proc. of the VLDB, 2010
http://dx.doi.org/10.14778/1920841.1920990

[13] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, Shashi Singh, "Lazy-Adaptive Tree: an Optimized Index Structure for Flash Devices", In Proc. of VLDB, pp. 361-372, August 2009.
http://dx.doi.org/10.14778/1687627.1687669

[14] Chang Xu, Lidan Show, Gang Chen, Cheng Yan, and Tianlei Hu, "Update Migration: An Efficient B+ Tree for Flash Storage", In Proc. of DASFAA, pp. 276-290, 2010.
http://dx.doi.org/10.1007/978-3-642-12098-5_22

[15] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu, "Flash-Optimized B+-Tree", Journal of Computer Science and Technology, Vol. 25(3), 2010.

[16] Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, Peng Wei, "A Reliable B-Tree Implementation over Flash Memory", SAC 08, 2008.

[17] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)", Acta Informatica, Vol. 33(1), pp. 351-385, 1996.
http://dx.doi.org/10.1007/s002360050048