Asia-pacific Journal of Convergent Research Interchange
Vol.6, No.7, July 31 (2020), pp.21-28
http://dx.doi.org/10.47116/apjcri.2020.07.03

F^2B+-tree: A Flash-aware B+-tree Using the Bloom Filter

Sungchae Lim[1)]

Abstract

As the price per bit of flash storage is rapidly decreasing, diverse research has been done to devise flash-aware B+-tree indexes. Since the original B+-tree structure was devised for indexing data records stored in hard disk drives, a naive transplant of the B+-tree into flash may degrade index performance. This is because flash storage suffers from significant performance disparity between update operations and read/write operations. To solve the problem, we adopt the probabilistic index structure, called the Bloom filter. By using the Bloom filter, we make some free space in each node whose child nodes are leaf nodes. We refer to such a node as the BF node. In the free space of a BF node, our proposed F^2B+-tree stores update logs in order to save histories of key inserts or deletes that have arisen in leaf nodes. Since B+-tree's nodes except for leaf nodes are usually manipulated in a memory buffer pool, the F^2B+-tree can considerably reduce the amount of physical updates on flash storage. Additionally, we cluster a set of sibling leaf nodes in a flash block such that garbage collection can be cheaply performed without full-merges or half-merges. As a result, the F^2B+-tree can prevent unpredictable fluctuations in performance of flash-based databases, which could be caused by background-mode actions needed for garbage collection.

Keywords: Bloom Filter, B+-tree, Database System, Flash Memory, High-performance Index

1. Introduction

As the price per bit of flash storage is rapidly decreasing nowadays, many I/O-related research has been done to use it as storage media of high-performance database systems[1-10]. Among them, the algorithms for flash-aware B+-tree indexes have drawn much attention from computer communities[3][5-9]. The B+-tree index was originally devised for efficient access to data records stored in a hard disk drive (HDD). Since the B+-tree structure is easily able to provide a high degree of fan-outs and space utilization ratio as well as the ability of range searches, it has taken the first place among various other index schemes devised for HDD-based database systems.

When it comes to the B+-tree being used in flash storage such as SSD (solid-state device), it is required to take account of the negative effect of the inefficient update mechanism of flash

Received(March 09, 2020), Review Result(1st: April 04, 2020, 2nd: May 28, 2020), Accepted(June 25, 2020)

1) (Professor) 02748 Dept. of Computer Science, Dongduk Women's Univ., Wolgok-dong, Seongbuk-gu, Seoul, Korea
email: sclim@dongduk.ac.kr

storage[4][5]. This is different from the HDD case, where HDD storage has no significant performance disparity between update operations and read/write operations. Since the B+-tree has most of its updates on leaf nodes that cannot be fully cached in the buffer pool, it cannot avoid many of randomly scattered physical updates. Such an I/O characteristic of the B+-tree is reported to worsen update's inefficiency seriously, if the B+-tree is transplanted into flash storage[2][3][5-9].

Against this problem, earlier researches[1-9] rely on two kinds of schemes, that is, an update caching scheme and an in-storage logging scheme. In the update caching schemes, previous key insert/delete operations are recorded accumulatively in either internal nodes or a separate sub-tree for tracing them[1][6-9]. The areas for saving those update information are managed in main memory. To ensure correct key searches, the cached update information is referred to by a key searcher on the way to its target leaf node. When the memory space for update information becomes full, recorded key inserts/deletes are written at once into leaf nodes. In the in-storage logging scheme, on the other hand, the update information is recorded as log records in log areas in storage[2-5]. The sibling leaf nodes share areas saving update log records. More specifically, a log area is allocated for each flash block. By doing updates in the unit of a flash block, the in-store logging scheme can guarantee efficiency of garbage collection. For correct key searches, real-time redos are executed for restoring data pages correctly. For both of schemes, heavy overhead is not avoidable because of a high cost paid for managing the space used for update information.

In this paper, we adopt the probabilistic index structure of the Bloom filter (BF)[11][12] in order to obtain a compact size of a B+-tree index. Using the free node space saved by the BF, our proposed B+-tree stores update information. We refer to nodes containing BF data as BF nodes, which are parent nodes of leaf nodes. When the free space made in a BF node comes to be exhausted because of key inserts and key deletes, the recorded previous updates are written to associated leaf nodes at once. Because such a batch-style update is executed within a flash block, the proposed scheme can support a low cost of garbage collection.

The organization of this paper is as follows. In Section 2, we describe some technical backgrounds about the Bloom filter and the flash-based B+-tree structure. In Section 3, we present the proposed flash-based B+-tree, and then discuss the performance enhancement of this research in Section 4.

2. Technical Backgrounds

2.1 Bloom Filter

The Bloom filter (BF) is a probabilistic index of which data is represented by bit-vectors[11]. The BF index is created using the OR bit operation among bit-vectors that are computed with a fixed number of hash functions. The BF index has three integer parameters of n, k, m, where n is the maximum number of items to be indexed, k is the number of hash functions used for creating bit-vectors, and m is the bit size of the BF index. When an item x is indexed with the BF scheme using the parameters above, its bit-vector of v(x) is expressed with m bits. Here, if the hash value of hash function $h_i(x)$ ($1 \le i \le k$) is integer j ($1 \le j \le m$), then the j-th bit of v(x) is set to 1; otherwise, the bit is set to zero. To index a item x using the BF scheme, we first compute an index bit-vector of v(x), which has the bit length of m and k bits of 1. Then, we apply the OR bit operation between v(x) and its current BF index, where the initial BF index is set with m bits of zero.

With the BF index built from BF bit-vectors of indexed items, we can answer a set membership query for a given item. The membership test using the BF index is very simple. For membership test of item x, we just check wether or not the result of an AND bit operation between the BF index and v(x) equals to v(x). If equivalent, then membership query on x will be returned with answer 'YES'.

Since the size parameter m can be appropriately controlled, the BF scheme is very useful for many applications with memory limitation[11][12]. Because of the feature of the OR bit operation, however, the BF scheme cannot avoid false positives. Furthermore, the BF index cannot support the deletion operation without false negatives. The BF scheme is not good for an application having many items that are removed over time. In other words, the BF scheme is suitable for indexing either a fixed set of items or a constantly growing set of items[11].

2.2 Flash-based B+-trees

Due to the high popularity of NAND-type flash for its fast random reads, there have been lots of researches for devising flash-aware B+-trees in database communities. A majority of researches are focused on the problem of the expensive update operation in flash. Since leaf nodes of a large B+-tree cannot be cached in the buffer memory of a database system, updates at a leaf level result in frequent physical updates on flash storage. Therefore, frequent updates on leaf nodes seems to be a big threat to B+-tree performance in flash storage[11-13].

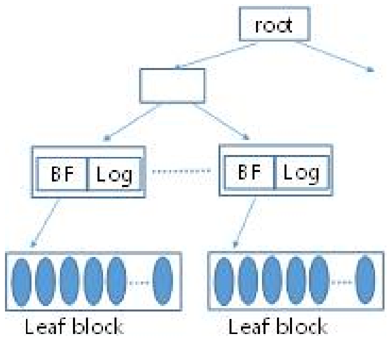
To solve this problem, diverse flash-based B+-trees have been proposed in literature[2-9]. Some studies proposed a B+-tree structure using a flash translation layer[4]. [3][5] employed a logging technique to reduce physical updates on storage. For example, when attempting to insert a key k into a leaf node L, an update record of (k, L, insert) is written in a separate log area instead of updating L itself. The accumulated update records are reflected into associated leaf nodes in a batch mode. For fast key search speeds, such update records could be saved in buffer memory. For this, the update records are stored in internal nodes. Note that the internal nodes of B+-tree are usually cached in a buffer pool. As another approach, [8] used a separated subtree where update records are recorded and dumped down to a leaf level for getting free space in the sub-tree. In the schemes above, there is a significant overhead for key searching, because extra time for reading update records is consumed for every key search.

3. Proposed B+-tree Using the Bloom filter

3.1 Idea Sketch

In order to prevent the performance degradation caused by key deletion/insertion, our proposed B+-tree structure is based on two ideas for performance enhancement. The first idea is to save BF data in parent nodes of leaf nodes. Thanks to the compact size of BF data, we can save some free space in each BF node and use it for storing insertion/deletion information relevant with leaf nodes. From this, our tree can reduce physical updates on leaf nodes without extra consumption of storage or memory space needed for saving update information. The second idea is to minimize I/O cost paid for garbage collection by clustering sibling leaf nodes in the same storage block. Since leaf nodes having same parent are updated in the unit of a flash block, a cost for garbage collection for those nodes are very cheap in our scheme.

To describe the proposed flash-based BF-embedded B+-tree, or F^2B+-tree for short, we use [Fig. 1]. In the figure, BF data is stored at level 2 of F^2B+-tree, where level 1 is that of leaf nodes. The BF area saves the BF index that is made from key values saved in leaf nodes belonging to a leaf block. Together with the BF data, update records for key deletion/insertion are saved in the log area so that they can be dumped down to sibling leaf nodes at once. In the figure, the boxes label with ‘Log’ are the areas for update records.



[Fig. 1] An Example of the Proposed F2B+-tree Structure

Using a BF index at level 2, any searcher S makes a decision about whether or not it needs to search down to a linked leaf block. If the downward search is decided via a set membership test, then searcher S reads a leaf block and tries to find a leaf node containing its search key. Then, the procedure for finding a target data record is executed by following the same search algorithm as that designed for the original B+-tree.

The rest of space in a level-2 node is used for saving update records with the data structure of (*op*, *k*, record ID). Here, the values for *op* are 'plus' or 'minus' representing insertion or deletion, respectively. Two values of *k* and record ID are data for a key entry expressing ID of a data record with key *k* Whenever an update arises at a leaf node, a corresponding update record is appended into the log area of its parent node. If the log area becomes full, then the accumulated update records are reflected to a leaf block. Since nodes of level 2 are usually manipulated within memory buffers, we can reduce a number of physical updates in leaf nodes.

3.2 Idea Algorithm for the Key Search

Naturally, a greater size of a BF index in a node supports a lower false-positive rate. However, since the larger BF index takes away space for saving update records from a BF node, we need to consider a trade-off between the false-positive rate and the number of reduced update costs. A feasible size of the BF index can be estimated using the well-known formula given in [3][4]. The formula is as follows:

$$FP(m,k,n) \cong (1 - e^{(-k \times n/m)})^k \qquad \text{——} \qquad (1)$$

, where FP(m, k, n) is the false-positive rate of the BF(m, k, n).

For a B+-tree structure with nodes of size 4 KB, we allocate 20% of node space as the space used for update records. In this case, for keys with sizes of 20 bytes to 40 bytes, we can save updates records of 40 to 20 in number. Theoretically, we can reduce the number of physical updates on leaf nodes in the proportion of the log area size.

In [Fig. 2], we show how to search down for a target leaf node with a given search key. The procedure SearchRecor(d) of [Fig. 2] receives search key k and returns the address of a leaf node containing that key; if there is no key in the B+-tree, the algorithm returns the result of NULL. In the algorithm, we assume that a buffer pool is used to cache nodes, as with usual database systems.

```
Procedure SearchRecord (in k, in root)
1.  tree_level ← the height of the tree with the root pointer root;
2.  Current ← Buf.read_node(root); // reading of a node using a buffer pool
3.  while (tree_level ≧ 2) // searching down until a level-2 node is reached
4.      Get the address to the next child node by using the index entries in Current ;
5.      Let child be the node address of that child node to be accessed;
6.      Current ← Buf.read_node(child); // reading of the next child node
7.      tree_level ← tree_level - 1;
8.  end_while
9.  for update records r in Current
10.     if r.key == k && r.op == 'minus' then return NULL; // deleted key
11.     if r.key == k && r.op == 'plus' then return r.rec_ID; // inserted key
12. end_for
13. bit_vector ← BF_hash(key); // compute a bit-vector using the given hash functions
14. if bit_vector == (Current.BF & bit_vector) then // membership testing using BF data
15.     Block ← Buf.read_block(Current.child_ID); // reading of a leaf block
16.     Check if there is a leaf node containing a index entry having its key value of k;
17.     if found then
18.         Let key_entry be the found key entry; // index entry (key, rec_ID)
19.         return key_entry.rec_ID; // return ID of the data record with key k
20.     else
21.         return NULL; // case of a false-positive
22.     end_if
23. else
24.     return NULL;
25. end_if
```

[Fig. 2] Algorithm for a Key Search in the Proposed F2B+-Tree

For space limitations, we do not present the detailed algorithm for an update process that performs key insertion or deletion. In the case of update processes in our scheme, they just insert update records into nodes of level 2. While an update record is being written to a BF node P, we do not modify the BF index of node P. A chance to the BF index is performed when the accumulated update records are flushed to leaf nodes.

4. Conclusion

In this paper, we proposed an F^{2}B+-tree index that incorporates the Bloom filter into the B+-tree structure. Since B+-tree's nodes except for leaf nodes are usually updated within a buffer pool, saving of update records does not impair the performance of the F^{2}B+-tree index. By delaying physical updates in leaf nodes, the proposed F^{2}B+-tree can reduce considerably physical updates. Additionally, we cluster a set of sibling leaf nodes in a flash block so that garbage collection can be cheaply performed without full-merges or half-merge. As a result, the proposed F^{2}B+-tree can prevent undesirable fluctuations of storage throughput[13-15].

The downside of the proposed F^{2}B+-tree comes from an inherent drawback of the FB index, that is, existence of false positives. When a false positive occurs unfortunately during a key search, our search algorithm reads a whole flash block and scans all the leaf nodes saved in that. If frequency of such false-positive searches is high, the efficiency of key searching becomes poor. However, such performance degradation is not realistic for two reasons. First, the I/O overhead for reading leaf nodes is trivial in the case of flash storage. Because time for reading a block is very short in flash, the extra reads caused by false-positive searching may not degrade the B+-tree performance severely. Second, the false-positive rate itself can be kept within a low degree. Since the BF index is made for a relatively small number of leaf nodes, the rate of false-positive is very low.

Acknowledgement

This work was supported by the Dongduk Women's University grant in 2019.

References

[1] Sangwon Park, Ha-Joo Song, Dong-Ho Lee, An Efficient Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory, International Conference on Embedded Software and Systems, **(2011)** May 14-16; Daegu, Korea.

[2] Chin-Hsien Wu, Tei-Wei Kuo, Li Ping Chang, An Efficient B-tree Layer Implementation for Flash-memory

Storage Systems, ACM Transactions on Embedded Computing Systems, **(2012)** Vol.6, No.3, pp.1-23, DOI: https://doi.org/10.1145/1275986.1275991

[3] Gap-Joo Na, Sang-Won Lee, Bongki Moon, Dynamic In-Page Logging for B+-tree Index, The 18th ACM conference on Information and knowledge management, **(2012)** November 4-6; Hong Kong, China.

[4] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler, Flashing Databases: Expectations and Limitations, International Workshop on Data Management on New Hardware, **(2010)** Jun 23; Indianapolis, Indiana, USA.

[5] Sungchae Lim, A New Flash-based B+-tree with Very Cheap Update Operations on Leaf Nodes, International Conference on Engineering Technologies and Big Data Analytics, **(2016)** Jan 21-22; Bangkok, Thailand.

[6] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, Ke Yi, Tree Indexing on Solid State Drives, Proceedings of the VLDB, **(2010)** Sep 13-17; Singapore, DOI: https://doi.org/10.14778/1920841.1920990

[7] Hua-Wei Fang, Mi-Yen Yeh, Pei-Lun Suei, Tei-Wei Kuo, An Adaptive Endurance-aware B+-tree for Flash Memory Storage Systems, IEEE Transactions on Computers, **(2013)**, Vol.63, No.4, pp.2661-2673.

[8] Sai Tung On, Haibo Hu, Yu Li, Jianliang Xu, Flash-Optimized B+-Tree, Journal of Computer Science and Technology, **(2010)**, Vol.25, No.4, pp.509-522.

[9] Chang Xu, Lidan Show, Gang Chen, Cheng Yan, Tianlei Hu, Update Migration: An Efficient B+ Tree for Flash Storage, International Conference on Database Systems for Advanced Applications, **(2010)**, April 1-4; Tsukuba, Japan.

[10] Jirapong Tongpang, Prasong Praneetpolgrang and Nivet Chirawichitchai, Hybrid Recommendation Technique Selection Center Donated Bags with Data Envelopment Analysis, International Journal of Disaster Recovery and Business Continuity, **(2013)**, Vol.4, pp.45-56.

[11] Biplob Debnath, Sudipta Sengupta, Jin Li, David J. Lilja, David H.C. Du, BloomFlash: Bloom Filter on Flash-based Storage, The 31st International Conference on Distributed Computing Systems, **(2011)**, July 25; Washinton DC, USA.

[12] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo, The Dynamic Bloom Filters, IEEE Transactions on Knowledge and Data Engineering, **(2010)**, Vol.22, No.3, pp.120-133.

[13] Siu Kang, Kazusa Ando, Tetsuya Yuasa, The Planning and Implementation of Smartphone Application Designed to Efficient Donation for Direct Support to the 2011 Tohoku Earthquake-Affected Area, International Journal of Disaster Recovery and Business Continuity, **(2015)**, Vol.6, pp.1-8.

[14] Ashima Pansotra, Simar Preet Singh, Additive Hough Transform and Fuzzy C-Means Based Lane Detection System, International Journal of Disaster Recovery and Business Continuity, **(2017)**, Vol.8, pp.1-28.

[15] Sohrab Khan, Nor Zairah Ab. Rahim, Nurazean Maarop, A Review on Antecedents of Citizen's Trust in Government Social Media Services, International Journal of Disaster Recovery and Business Continuity, **(2018)**, Vol.8, pp.21-30.