Optimization of a Multiversion Index on SSDs to improve System Performance

Won Gi Choi*, Mincheol Shin*, Doogie Lee*, Hyunjun Park‡, Sanghyun Park*,†

* Department of Computer Science, Yonsei University,
50 Yonsei-ro, Seodaemun-gu, Seoul, Korea
{cwk1412, smanioso, edoogie, sanghyun}@yonsei.ac.kr
‡ Department of Computer Science, University of the Pacific,
3601 Pacific Avenue, Stockton, California 95211, USA
j_park32@u.pacific.edu
† Corresponding Author

Abstract— **In this paper, we propose a multiversion index utilizing key features of SSDs (solid state drives). SSDs have many advantages, e.g., fast read/write performance, high energy efficiency, and non-volatility. Thus, SSDs have been considered for several years as a promising alternative to HDDs (hard disk drives). Many studies have made progress in optimizing and modifying HDD-based database management system (DBMS) to suit SSDs. In the case of multiversion databases, which manage not only keys and but also versions, research optimizing SSD query processing has been ignored in comparison with single versioned databases. Generally, the multiversion databases manage an evolving data which is processed in a cyber physical system or an accounting system. Therefore, the data is large and the index structure requires frequent rearrangement of its structure to maximize efficiency, which is called structure modification operation. The multiversion index based on HDDs utilizes random writes to conduct the structure modification operation. This feature can introduce crucial performance problems on SSDs, because the speed of random writes on SSDs is much slower than the speed of sequential writes. We propose a Bulk Split multiversion tree (BSMVBT) index that utilizes sequential pattern I/O and out-of-place updates of SSDs. Experimental results showed that it is 10% – 30% faster than the compared version.**

I. INTRODUCTION

Flash disks or SSDs (solid state drives) have received attention from enterprises and researchers, because they have emerged as promising alternatives to a hard disk drive (HDDs). Because SSDs have the advantages of high read/write performance, low power consumption, high energy efficiency and non-volatility, SSDs are frequently used as cache storage between main memory and the hard disk, as the main data storage.

However, SSDs also have key features that can lower the overall performance of a program. For example, SSDs have asymmetric read and write speeds. Moreover, performance of random writes is lower than the performance of sequential writes. In addition, SSDs have an erase operation that is necessary to manage invalid data resulted from the feature of the flash memory that does not allow data to be updated in place. The erase operation can affect overall performance because overhead of the operation is heavy. Many previous studies have optimized database management systems to overcome these SSD features to improve the overall performance.

FD-tree[10] and psync I/O (PIO) B-tree[12] are representative single version index structure which improve the read/write performance in SSDs. FD-tree reduces the number of random writes utilizing fractional cascading and logarithmic techniques. It converts the random writes of the original B-tree index to sequential writes. PIO B-tree utilizes the internal parallelism of SSDs to improve the speed of random writes. PIO B-tree collects a number of random I/O to send at the same time, because SSDs can handle I/O in parallel, maximizing efficiency.

As sensing technologies have been developed, multiversion databases has been noticed in recent years. The multiversion databases have been utilized in various fields. For example, log record system which monitors the status of objects or location prediction system which tracks the movement of objects can apply the multiversion databases. While the data structure affects the overall performance of system totally, the optimization of the multiversion data structure in SSDs has been comparatively ignored in contrast to single-version. Transactional multiversion B-tree [7] (TMVBT) is a representative multiversion index structure based on a partially persistent database that allows only one update transaction with several read only transactions. The transactional multiversion B-tree generates many random writes to conduct its structure-modification operation, which is called a key split, or a version split. TMVBT has two dimensional spaces, e.g., key and version, so split operations that generate random writes are much more frequent than in the original B-tree index.

Because of the slow speed of random writes in comparison with sequential writes, SSDs lose efficiency. Therefore, the multiversion index also requires adaptation for SSDs to improve read/write performance utilizing sequential I/O. In addition, SSDs have old version data before the garbage collection operation occurs, because flash memory does not allow overwrites. This point can be coordinated with storing multiversioned data.

We propose a multiversion index structure with techniques of converting random writes to sequential writes and utilizing the preserved old version data to perform a structure-modification operation of the index.

The paper is organized as follows. In Section 2, we review previous research about SSD optimization techniques. In Section 3, we propose a multiversion index structure and explain its basic operations and structure

978-1-5090-1897-0/16/$31.00 ©2016 IEEE

modification operations. In Section 4, we evaluate the performance of our structure. Finally, we conclude in Section 5.

II. Related Work

A. FD-tree

FD-tree is an index structure designed to improve the random write performance of SSDs. FD-tree takes advantage of hardware features of the SSDs by utilizing sequential accesses, eliminating the slow random writes. As shown in Figure 1, FD-tree is composed of a B+tree called a head tree and several levels. Each level is limited to a constant size and the level is ordered by key. As the level decreases, the size of the level increases at a constant rate. FD-tree applies a fractional cascading algorithm to improve the read performance.

FD-tree uses a fence to implement the fractional cascading. A fence is a data structure that has a pointer to indicate the first entry of a specific page. During search operation of FD-tree, the fence accesses the pointer to avoid searching all of the pages of the next level. The key of the first entry of each page in the next level is same as the key of the fence. Because the entries of each level are sorted in ascending order, including the fence, the pages in the sorted level efficiently utilize a sequential write during FD-tree operations.

The insert operation is accomplished by inserting a key into the head tree which is composed of the original B+tree. When the number of keys exceeds the size of the leaf level of the head tree, a merge operation is performed to flush the keys to a lower level. The merge operation is based on the merge-sort algorithm. After the merge operation, an algorithm that reflects fences to higher levels is executed to support the search operation.

The search operation is accomplished by looking up a key in the head tree. If the key is not in the head tree, a binary search is performed to find the greatest fence key equal to or less than the desired key. The search operation is supported by the fence which points to the pages in the next level. The delete operation inserts a fake key equal to the key that is to be deleted. When the fake key meets the specific key through the merge operation, the key will be removed physically.

It has been shown that FD-tree dominates the other B-tree variants on SSDs by utilizing the above efficient techniques. However, FD-tree is only utilized in a single-version database and is regarded as an unsuitable index structure for multiversion data.

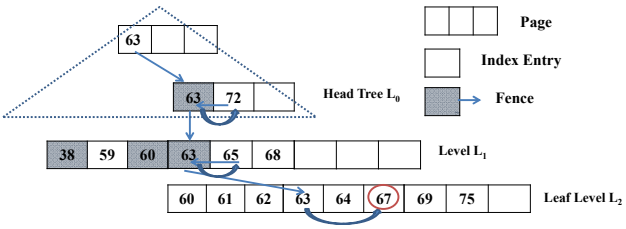

Figure 1: Overview of FD-tree

B. Transactional Multiversion B-tree

The transactional multiversion B-tree (TMVBT) (see Figure 2) is an extension of the original B-tree adapted for SSDs to index multiversioned data. In [1], Becker et al., proposed the multiversion B-tree (MVBT) for indexing multiversions of data. MVBT is a directed acyclic graph of B-tree nodes. MVBT maintains multiple B-tree root nodes in its own data structure to partition the version space of the data items. The transactional multiversion B-tree is a variant of MVBT that allows multiple data items in one transaction and supports ARIES-based recovery. TMVBT allows partial persistence, the same as MVBT, which has only one update transaction in recent version. However, the number of read only transactions is unlimited and can access any version less than the update version.

TMVBT is mainly used to support temporal databases for indexing evolving data, such as engineering designs, land registers, scheduling applications, inventory control systems, moving object databases, etc. Furthermore, because TMVBT preserves the previous state of the data, TMVBT is applied to recovery systems that do not store log files. Even though TMVBT has high availability in a specific database, research on optimizing TMVBT for SSDs has been ignored in comparison with single version index structure.

As shown Figure 3, the transactional multiversion B-tree (TMVBT) has two basic operations; key split and version split. The key split is the same as the key-split operation of the conventional B-tree. When the number of keys in one node exceeds the capacity of the node, it requires a distribution of keys. The key split leads to distributing keys according to the key range. The key split only occurs on active pages that are modified by a current update transaction.

In contrast to the key split, the version split occurs on inactive pages that were modified by the previous update transaction. The version split has the role of splitting the version range of a data item, resulting in the creation of a new root that represents the specific version range. The creation of the new root supports high concurrency control because several read-only transactions can access the old root structure and read data from the old version tree.

Because of the SSDs' poor random-write performance, TMVBT's key split and the version split cause a heavy cost while processing queries. Due to the feature of TMVBT that the version of an update transaction continuously increases, version splits occur frequently. Furthermore, since TMVBT allows multiple data to be updated in one update transaction, many key splits are also performed frequently. Therefore, TMVBT requires an adaptation for SSDs with a method of eliminating random writes.

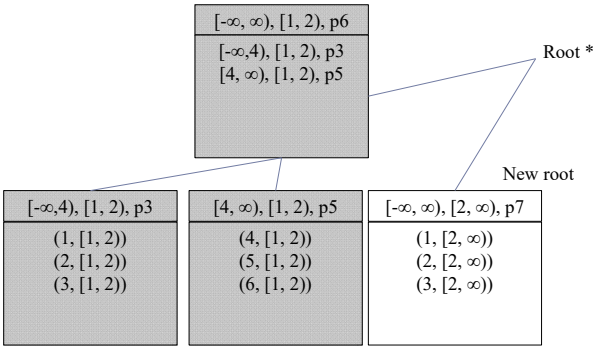

Figure 2: Overview of a Transactional Multiversion B-tree

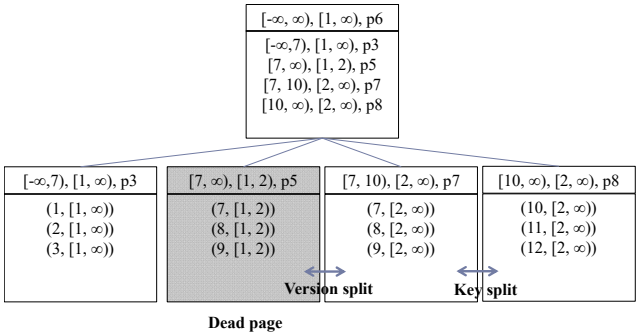

Figure 3: Key Split and Version Split of a Transactional Multiversion B-tree

III. BULK SPLIT MULTIVERSION TREE

A. Overview

In this paper, we propose a multiversion index structure based on FD-tree. FD-tree doesn't support multiversion database, so we modify it by adding timestamp data to the entry as shown in Figure 4. Each entry has information about keys and a timestamp equivalent to the version. Entries also have a dead flag that indicates when the entry is dead.

The Bulk Split multiversion index tree is composed of a head tree and several levels. The head tree is a variant of TMVBT that supports a new type of entry structure. Entries in each level are sorted by key and timestamp value. Firstly, entries are sorted by key value, and in case of same key, entries are sorted by timestamp value.

In addition, we regard out-of-place SSD updates as a solution that reduces random writes. Because the multiversion database is intended to be partially persistent, a read-only transaction can access invalid pages in the old version without an additional split operation. Thus we allow each root structure that represents a specific version range to manage its own file pointer to decide whether to utilize file append when structure modification operation occur. The read-only transaction can access old version data without making a new level for them. Furthermore, the file append leads to better performance when all live data in the old version are inserted to a new root.

B. Insert Operation

The insert operation starts by inserting an entry into head tree that is composed of TMVBT. When the number of leaf nodes of the head tree exceeds a constant value, the data must be flushed to a lower level. The RunMergeOperation that merges two adjacent levels based on merge-sort algorithm occur recursively from a level which includes leaf nodes. After the RunMergeOperation function completes, the InsertFenceOperation that inserts fence into higher level to support the search operation. Like the FD-tree, we utilize fractional cascading and insert a fence which is a data structure that points to the ID of a page in the next level.

Once all of the merge operation processes are complete, we check whether the total sum of dead entries in each level exceeds a threshold. If so, we execute a structure modification operation called a sequential version split operation.

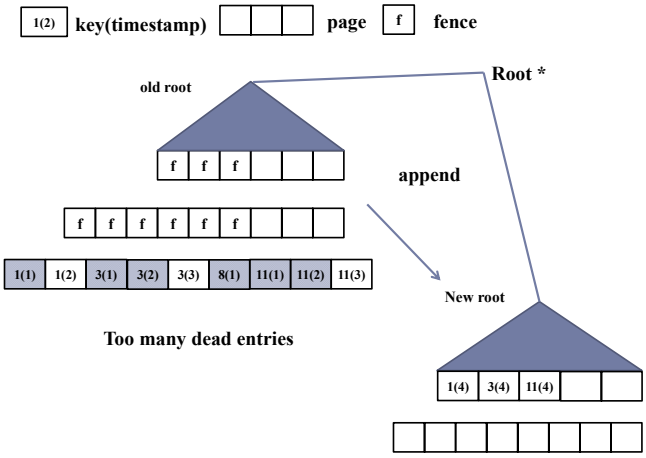

Figure 4: Overview of Bulk-Split Multiversion Tree. The structure modification operation leads to inserting all live data in the old version to the new root, which is appended to the old root in the file.

Algorithm 1 InsertOperation(Entry, Root, Run)

Description

Insert operation of the Bulk Split multiversion tree. Firstly, Insert entires to TMVBT. When the number of leaf nodes of the TMVBT exceeds a constant value, the data must be flushed to a lower level. When the merge operation processes are complete, we check whether the total sum of dead entries in each level exceeds a threshold. If so, we execute a structure modification operation called a sequential version split operation.

Input

Entry : Input entry which has key, timestamp, Dead/Live flag and pointer to pages in next level

Root : TMVBT root which has a specific version range, file descriptor and pointer information of the current version

Run : Data structure that manages levels under tree

Output none

```
1  InsertOperation to TMVBT
2  If [ the number of TMVBT leaves exceeds a
   sepecific constant value]
   Then
3     RunMergeOperation(level 0 , level 1, Root);
      /* level 0 is composed of TMVBT leaf nodes*/
4     InsertFenceOperation(Run)
5     If [ CheckDeadEntryNumExceed() == true ]
      then
      /* CheckDeadEntryNum() checks the sum of the
   levels' dead entries*/
6       SequentialVersionSplitOperation(Root, Run);
      end

   end
```

C. Search Operation

Each level is sorted by key-timestamp order. Basically, entries in a level are sorted in key order. When entries have the same keys, they are sorted in timestamp order. A search operation is conducted sequentially from the root of the head tree. The search operation tries to find the key or the greatest fence value less than the desired key that points to pages in the next level. When the fence is found, the search operation continues in next level. When the key is found, a binary search algorithm is used to find the desired timestamp value. This process continues until the desired key-version entry is found.

D. Merge Operation

The merge operation is the main operation of the Bulk Split multiversion tree index. The process is performed on two adjacent levels when the smaller one exceeds its size limit. The operation sequentially scans two levels, and combines them into one level to utilize the conventional merge-sort algorithm. The operation compares the key value and timestamp value of the two entries and arranges them in ascending order.

We developed the existing FD-tree merge operation to support the multiversion. Using a dead flag, we implemented not only a physical deletion that actually removes the entry but also a deletion by version update that splits the version range. During the merge operation, when two entries from different level have the same key, we compare their timestamps and dead flags. The dead flag stores the timestamp when an entry becomes dead. If an entry in the smaller level which sets its dead flag meets another entry in the bigger level with the same key value, the dead flag of the entry in the bigger level is changed with the one in the smaller level, and the smaller level entry is not written to the new level. Increasing the number of dead entries results in splitting the version range. This variant of the merge operation enables us to split the version range utilizing sequential I/O because handling the dead entries is occurred in unit of level. When live entries are separated from old and dead entries, random writes that occur by inserting live entries are transformed to sequential writes due to the structural features. Under the maximum bandwidth of SSDs, the merge operation utilizes the sequential reads and writes with better performance than the conventional multiversion B-tree.

Algorithm 2 MergeOperation(level 1, level 2, Root)

Description
When the number of entries in specific level exceeds its allowed size, it requires an operation to merge its data with the data of the next level

Input
level : Target levels of merge operation
Root : TMVBT root which has the specific version range, file descriptor and pointer information of the current version

Output none

```
1  Entry p,q;
2  While (p != NULL && q!=NULL)
   then
3    p = ReadEntryFrom(level 1);
4    q = ReadEntryFrom(level 2);
5    If [p.key > q.key]
     then
6      write q to L' // L' is new level to insert
7      point to next entry of q in level 2
     end
8    If [p.key < q.key]
     then
9      write p to L'
10     point to next entry of p in Level 1
     end
11   If [ p.key == q.key ]
     then
12     If [p.timestamp > q.timestamp]
       then
       /*dead flag expresses when entry is dead*/
13       If [p.dead != 0] /* delete by delete operation*/
         then
14         q.dead = p.dead;
15         write q to L'
16         point to next entry of p,q in level 1, level 2
         end
17       If [p.dead == 0] /* delete by version split */
         then
18         q.dead = p.timestamp
19         write q to L'
20         point to next entry of p,q in level 1, level 2
         end
       end
21     If [ p.timestamp == q.timestamp ]
       then
22       /* physically remove */
23       point to next entry of p,q in level 1, level2
       end
     end
24   If [ L' is full ]
     then
25    MergeOperation with next level
     end
   end
26 Replace L' with level 2
```

E. Delete Operation

In contrast to the other operations, the delete operation is simple. It starts by inserting an entry with a dead flag. The dead flag is set to represent when the entry is dead. A saved timestamp indicates when the delete operation occurred. The delete operation is actually performed during the merge operation. During the merge operation, when two dead entries that have the same key meet each other, the entry in the higher level is removed and the other entry saves the higher-level entry's dead flag in its own flag. Thus, when we search the entry, we can tell that the entry is dead and when it occurred.

F. *Structure Modification Operation*

The Structure-modification operation is necessary to manage the version ranges and provide high concurrency control in the multiversion tree. The main idea of the operation is to make a data structure to manage information, such as file descriptors and pointers to eliminate additional writes that is necessary to preserve the old version data. Flash memory has the feature of out-of-place updates; a page cannot be overwritten and it produces additional writes. For example, if one page is updated by a transaction, it becomes two pages: an old version page and a new version page. Even though the multiversion database has many read-only transactions and can access these old pages without additional write operation, the conventional multiversion tree index does not use the feature.

Thus, we propose to provide information about the file descriptor and file pointer of a specific version range to the root of the tree that represents a particular version. A root structure uses the information to locate where the levels and head tree are written in the file. If version splitting is unnecessary, the levels and head tree are overwritten at a previous location because the cost of the version splitting is more crucial than the performance benefit resulting from utilizing old pages. However, when too many dead entries remain in the total structure, version splitting is necessary. In this case, a new head tree and new levels are appended to the back of the old head tree and levels, and a new file pointer provided for a new root structure. Because the size of each level is invariant, appending and accessing levels or a head tree with particular file pointer information is possible. After a new root structure receives the file information, all live entries in old version tree are inserted into a new tree transforming random writes to sequential writes. Read transactions can access the old pages by referencing the file information in the old root without additional writes to make a new level for the old data.

Algorithm 3 SequentialVersionSplitOperation (Root, Run)

Description
Increasing the number of dead entries in Sequential Split multiversion tree requires a split oepration by version range.
Input
Root : TMVBT root which has specific version range, file descriptor and pointer information of the current version
Run : Data structure that manages the levels under tree

Output none
1 InsertNewRoot(newRoot);
/* create a new data structure which has a new file descriptor, and pointer information of the current
2 version.*/
Insert all live entries in Root & Run to new Root
/* old version is preserved at the location which is pointed to by the old Root structure*/
End

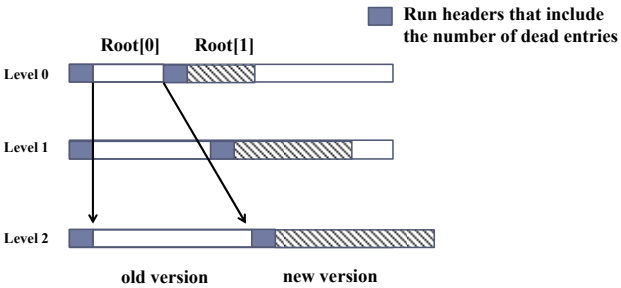

Figure 5: Sequential Version Split operation. When dead entries do not exceed a threshold, old version levels are overwritten by the new version levels. When the dead entries are enough to split, the old version levels are preserved and the new version levels are appended to old version levels.

IV. EXPERIMENT AND RESULT

In this section, we describe the evaluation of our Bulk Split multiversion tree in comparison with the transactional multiversion B-tree. We implemented TMVBT and Bulk Split multiversion tree using Direct I/O, and we experiment the performance of the tree with input queries that is made virtually.

We ran our experiments on a computer with an Intel i7-3770 quad core CPU 3.4GHz on CentOS 6.6 with 4GB main memory and an OCZ Vector 120GB SSD.

We experiment the operation of the tree index using Direct I/O. The size of one page was 4KB, and the size of one entry was 24 bytes. We simulated the case of 1000 insertions in five transactions, 1000 insertions in eight transactions and 1000 insertions in ten transactions. In the case of TMVBT, raw flash memory cannot allow overwrites so the split operations can affect more than three pages (target page, created page, parent page). Even when we only modify the key-value range in the header, it requires additional page writes. Thus, when many data inputs are inserted into TMVBT, a critical performance issue results. Moreover, if the version was frequently changed, too many version splits would occur, and require many random writes. Figure 6 shows that the Bulk Split multiversion tree is 1.1 - 1.3x faster than TMVBT in case of insert operations. Figure 7 shows the experimental result of execution time when the operations are composed of inserts and deletes. Because it utilizes sequential I/O, Bulk Split multiversion tree receives the benefit of the performance. The number of data items in one transaction and the number of version changes affect the gap of performance between TMVBT and the Bulk Split multiversion tree.

Moreover, Figure 8 shows that the operation time is varied according to the number of leaves of head tree. The more the number of leaves of tree is, the faster the operation is. As the number of leaves increases, the tree utilizes sequential I/O better. We experiment it with 1000 insertions in ten transactions.

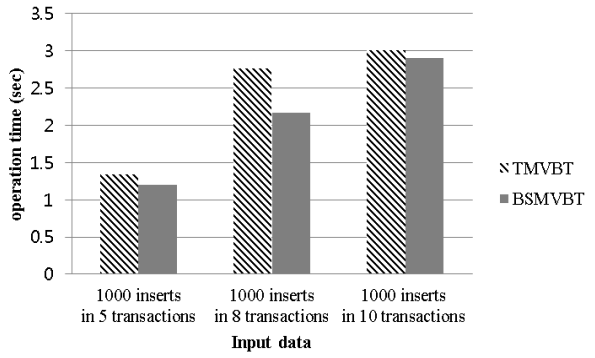

Figure 6: Experimental result. The Y-axis shows the total operation time, and the X-axis shows the type of queries about insertions.

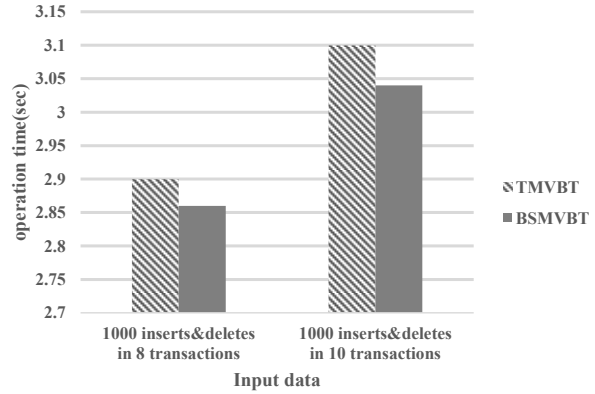

Figure 7: Experimental result. The Y-axis shows the total operation time, and the X-axis shows the types of queries about insertions and deletions.

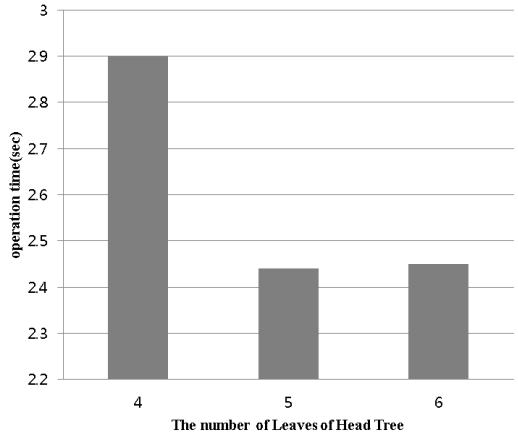

Figure 8: Varying the number of Leaves of Head Tree in case of 1000 inserts in ten transactions.

V. CONCLUSION AND FUTURE WORK

Previous multiversion index structures originally based on hard disk drives generate too many random writes due to their rearrangement of structure that is necessary to balance the structure. Because SSDs have better performance on sequential writes than random writes, the multiversion index structure requires a new design to improve overall performance. In this paper, we proposed a multiversion index structure that utilizes key features of SSDs (solid state drives). We designed our tree index with methods of reducing slow random writes. We converted the conventional index structure to utilize sequential writes. Furthermore, we mitigated additional random writes resulting from out-of-place update of SSD. Because a multiversion database is assumed to have only one update transaction and a few read-only transactions, we can read access version data without additional structure modification operations. As the SSD overwrite cost is heavy, eliminating the cost of the version splits affects the overall performance of the index structure.

We believe that it is possible to apply the proposed data structure to practical systems. Moreover, it is possible to extend our algorithm to exploit characteristics of the data in specific system, such as location prediction system. Furthermore, we will apply the index structure to flash aware distributed database systems.

ACKNOWLEDGMENT

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning (NRF-2015M3C4A7065522).

REFERENCES

[1] Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. An asymptotically optimal multiversion B-tree. In VLDB Journal, Dec.1996, vol. 5, no.4, pp. 264–275.

[2] Chazelle, B. and Guibas, L. J. Fractional cascading: I. a data structuring technique. Algorithmica, 1(2), 1986

[3] Do, J., Zhang, D., Patel, J. M., DeWitt, D. J., Naughton, J. F. and Halverson, A. Turbocharging DBMS Buffer Pool Using SSDs. In SIGMOD, 2011

[4] Do, J., Zhang, D., Patel, J. M., DeWitt, D. J. Fast Peak-to-Peak Behavior with SSD Buffer Pool. In ICDE, 2013

[5] Gal, E. and Toledo, S. Algorithms and data structures for flash memories. In ACM Comput. Surv., 2005, vol. 37, no. 2, pp.138-163

[6] Graefe, G. Write-optimized b-trees. In VLDB, 2004, pp. 672–683

[7] Haapasalo, T., Jaluta, I., Seeger, B., Sippu, S. and Soisalon-Soininen, E., Transactions on the multiversion B+tree. In The 12th International Conference on Extending Database Technology (EDBT ’09), 2009.

[8] Kimura, K. and Kobayashi,T. Trends in high-density flash memory technologies, In IEEE Conference on Electron Devices and Solid-Sate Circuits, 2003.

[9] Li, X. et al. A new dynamic hash index for flash-based storage. In WAIM, 2008

[10] Li, Y., He, B., Luo, Q., and Ke, Y. Tree indexing on solid state drives. In Proceedings of VLDB Endowment 2010.

[11] O’Neil, P. E., Cheng, E., Gawlick, D. and ’Neil, E. J. The log-structed merge-tree(lsm-tree). In Acta Inf., 1996, vol.33, no.4, pp. 351–385.

[12] Roh, H., Park, S., Lim, S., Shin, and Lee, S.-W. B+tree Index Optimizations by Exploiting Internal Parallelism of Flash-based Solid State Drives. In PVLDB 2012, vol. 5, no. 4, pp. 286–297.

[13] Stoica, R. and Ailamaki, A. Improving flash write performance by using update frequency. In Proceedings of VLDB Endowment, July 2013, 6(9):733–744.

[14] Thonangi, R., Babu, S., and Yang J. A practical concurrent index for solid-state drive. In The International Conference on Information and Knowledge Management (CIKM’12, 2012, pp. 1332–1341