International Journal of Internet, Broadcasting and Communication Vol.14 No.1 95-104 (2022)
http://dx.doi.org/10.7236/IJIBC.2022.14.1.95

IJIBC 22-1-12

A New Flash TPR-tree for Indexing Moving Objects with Frequent Updates

Seong-Chae Lim

Professor, Dept. of Computer Science, Dongduk Women's University, South Korea
sclim@dongduk.ac.kr

Abstract

A TPR-tree is a well-known indexing structure that is developed to answer queries about the current or future time locations of moving objects. For the purpose of space efficiency, the TPR-tree employs the notion of VBR (velocity bounding rectangle) so that a regional rectangle presents varying positions of a group of moving objects. Since the rectangle computed from a VBR always encloses the possible maximum range of an indexed object group, a search process only has to follow VBR-based rectangles overlapped with a given query range, while searching toward candidate leaf nodes. Although the TPR-tree index shows up its space efficiency, it easily suffers from the problem of dead space that results from fast and constant expansions of VBR-based rectangles. Against this, the TPR-tree index is enforced to update leaf nodes for reducing dead spaces within them. Such an update-prone feature of the TPR-tree becomes more problematic when the tree is saved in flash storage. This is because flash storage has very expensive update costs. To solve this problem, we propose a new Bloom filter based caching scheme that is useful for reducing updates in a flash TPR-tree. Since the proposed scheme can efficiently control the frequency of updates on a leaf node, it can offer good performance for indexing moving objects in modern flash storage.

Keywords: *Flash memory, TPR-tress, Moving object databases, Indexing scheme.*

1. Introduction

As networks for wireless communications get widely established, various location-based services have permeated our common life at a fast rate [1-7]. Since the location-based services need to efficiently locate moving objects, there has been much interest in the development of spatio-temporal indexing schemes in the database community [3-7]. The spatio-temporal indexing schemes are usually aimed at managing the positional information of moving objects with respect to time. In the aspect of a temporal dimension, user's queries may be issued for querying current or future-time positions of moving objects of interest. In this respect, a Time Parameterized R-tree (TPR-tree) was proposed [6]. Because of its high performance of answering future-time queries with less storage usage, the TPR-tree and its variants were intensively researched [4-7].

The TPR-tree is an indexing structure based on the notion of a bounding rectangle that always encloses the overall positions of a group of moving objects indexed. Rather than saving individual positional information of moving objects, the TPR-tree structure saves rectangular information that represents a maximum range of places for a group of moving objects. Since the data size needed to express such a bounding rectangle is much less than that required for locating individual moving objects, respectively, the TPR-tree scheme has the benefit

Manuscript Received: November. 22, 2021 / Revised: November. 28, 2021 / Accepted: December. 2, 2021
Corresponding Author: sclim@dongduk.ac.kr
Tel: +82-2-940-4589, Fax: +82-2-940-4170
Professor, Dept. of Computer Science, Dongduk Women's University, Korea

of less storage usage. However, the use of bounding rectangles inevitably leads to considerable misleading to false candidate leaf nodes during query processing times. This is because there exist large gaps between real positions of bounded objects and overestimated sizes of computed rectangles. Such a space gap is referred to as *dead space* [4-6]. Because of dead space, obsolete node accesses are unavoidable in the use of TPR-tree-based schemes. To minimize those dead spaces, the TPR-tree index is enforced to frequently update its leaf nodes, thereby keeping bounding rectangles as compact as possible [6, 7].

This update-prone property of the TPR-tree index seems to be more harmful when it is used for flash-based database systems. As widely studied in the literature, the performance of flash's random updates is very poor because of its inability to update in place [8-13]. Moreover, when overheads paid for garbage collection are taken into account, the cost for a random update seems to be two orders of magnitude larger than that for a random read [13-15]. Since actions for compacting dead space incur frequent random updates on leaf nodes, therefore, the TPR-tree index may not be feasible as a flash-resident index.

To solve this update-prone problem of the TPR-tree index, we develop a caching scheme that is used for temporarily saving data of updated bounding rectangles. To cache updates on leaf nodes, cache memory is prepared for each parent node of leaf nodes. Such cache memory is accessible by looking up address information saved in a parent node. Since the cached update information is made to be reflected into leaf nodes at once, our flash TPR-tree can reduce the overall number of updates at the leaf level. To diminish the memory usage of caching memory, we employ a Bloom filter index that was devised for efficient membership tests [16-18]. By saving a Bloom filter index in each parent node of leaf nodes, our flash TPR-tree is able to get compacted bounding rectangles in a space-efficient and time-efficient way. Thanks to the reduced number of updates at the leaf level, our TPR-tree can retain its good performance of query processing, even if it is deployed for use in flash storage.

The rest of this paper is organized as follows. In section 2, we present some technical backgrounds relevant to a Bloom filter scheme and a TPR-tree index. The notion and used algorithms of the proposed flash-based TPR-tree are addressed in Section 3, and then the performance gains of the proposed scheme are addressed in Section 4. Lastly, we conclude this paper in Section 5.

2. Preliminaries

In this section, we introduce an indexing technique called a Bloom filter. This indexing technique is employed as a basic mechanism for providing the enhanced performance of our proposed flash TPR-tree.

2.1. Bloom filter

A Bloom filter (BF) is a probabilistic indexing scheme that was developed for efficient membership testing [16]. The BF index is characteristic of high space efficiency, and thus it has been accepted as a compact indexing scheme useful for checking the existence of items being tested [17, 18]. To index an item with key-value k, the BF scheme works with two parameters of m and n_h . Here, integer m is the fixed length of a bit-vector generated from each key-value k, and integer n_h is the number of hash functions used for generating the bit-vector. Depending on k. some bits among a bit-vector with length m are set to 1. For this, hash functions are used for choosing the set of bits to be set with 1. Specifically, each hash function $h_i(k)$ ($1 \le i \le n_h$) is made to yield a hash value x such that $1 \le x \le m$. If the hash value of $h_i(k)$ is equal to x, then the x-th bit of a computed bit-vector is set for k . By applying n_h number of hashing functions on k, repetitively, we can get a random bit-vector whose 1's bits are up to n_h. With two parameters of n_h and m ($n_h < m$), the BF scheme can manipulate an index of length m. That index is made by doing OR bit-operations among individual bit-vectors of its indexed items.

For instance, suppose that a BF index is made for indexing N items whose key-values are equal to k_i ($1 \le i \le N$). With parameters of n_h and m, we can get N number of bit-vectors $v(k_i)$. To build a BF index for N items, the BF scheme repeatedly apply OR bit-operations over $v(k_i)$ ($1 \le i \le N$). From those OR bit-

operations, N number of bit-vectors are merged into a single BF index of length m. Let the BF index be denoted by V_{BF}. Then, to answer membership on an item with k_i, we just perform an OR bit-operation between $v(k_i)$ and V_{BF}. If its resultant value is not zero, then a membership test is returned as being true.

Although the BF indexing scheme shows off the advantages of a fixed small size of index data and fast lookup times for membership testing, it has a shortcoming of the existence of false-positive searches [17, 18]. Since the probability of false-positive searches usually increases with the frequency of deletions of items, the BF indexing scheme is not acceptable for some application domains where a lot of update workloads should be handled in an efficient way. As our proposed cache scheme can determine a proper time point for updating a BF index, the use of the BF indexing is plausible. More detail will be presented in Section 3.

2.2 TPR-Tree Index

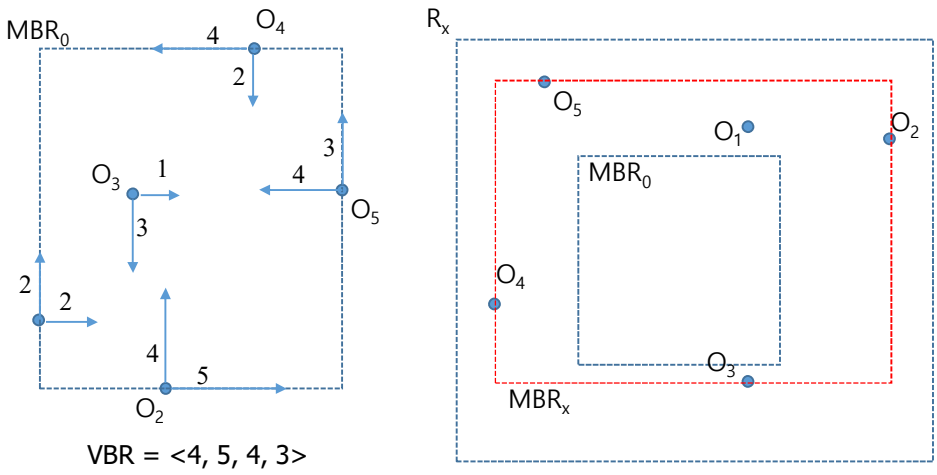
As an indexing structure used for processing positional queries on stationary objects, an R-tree was proposed [3]. In the R-tree, a data page N is stored to contain geographical information of k objects. Then, to represent the set of k objects in N, the R-tree computes the smallest rectangle that encloses all the places of k objects. Such a smallest rectangle is referred to as an *MBR* (Maximum Bounding Rectangle), which is represented with two points being at the ends of its diagonal [3]. To index k objects of N, an MBR representation and the address of N are saved in a leaf node. To answer a user query with target range r, a search process moves toward candidate leaf nodes whose any MBR overlaps with r. If a new insertion of an object overflows data page N, then $(1 + k)$ number of objects are split over N and a newly allocated data page. At the same time, a new MBR is inserted into an associated leaf node in order to index the new data page. Such overflows can be propagated recursively up to the root node of an R-tree.

Although the TPR-tree borrows the concept of the MBR from the R-tree, it was devised for indexing moving objects, rather than indexing stationary ones. To answer range queries on moving objects, the TPR-tree saves velocity information of moving objects as well as positional information. The velocity information is updated whenever any indexed object changes its directions or speeds of movement [4, 5]. To express the velocity information, the TPR-tree employs the notion of *VBR* (Velocity Bounding Rectangle) with respect to each MBR. A VBR is expressed with a velocity vector of $<v_1, v_2, v_3, v_4>$, which expresses the maximum velocities of objects in an MBR with respect to every direction. By expanding an MBR M at the rate of its VBR, the TPR-tree can compute a growing bounding rectangle containing all the objects indexed in M. Since the TPR-tree index can compute such bounding rectangles for future-time points, it can answer future time queries as well as current time queries. We refer to such a constantly expanding bounding rectangle as *CER* (Constantly Expanding Rectangle) for short.

Figure 1 illustrates an example of a CER that is used for indexing five moving objects from time T_0 until time T_x . In Figure 1.(a), notation v_i of $<v_1, ..v_4>$ denotes the maximum speed of the i-th direction of a VBR. In this fugue, MBR_0 is an MBR enclosing five objects being indexed at T_0. In Figure 1.(b), the outer rectangle R_x represents a CER that has been expanded from the MBR_0 until T_x. To compute the CER, the TPR-tree expands MBR_0 at the rate of VBR <4, 5, 4, 3> until T_x.

Since a CER expands at the rate of its VBR, the TPR-tree cannot avoid performance degradations that are caused by the difference between any CER X and the real positions of the objects indexed by X. In the example of Figure 1.(b), the difference between R_x and MBR_x is an example of dead space at the time T_x. Since the size of dead space gets larger over time, the performance of query processing constantly deteriorates because of an increasing number of useless searches coming into dead spaces [4-6].

To ensure better performance, the TPR-tree needs to efficiently keep dead space less. To this end, the traditional TPR-tree algorithms rely on the compaction of CERs in leaf nodes [7]. For instance, we can eliminate the dead space in R_x by updating the CER of Figure 1.(b) with MBR_x . This compaction of CER R_x can be performed when a leaf node containing R_x is accessed by a search process or an update process. However, as we mentioned before, such an update on the leaf node easily impairs the performance of TPR-tree, if the tree is stored in flash storage. To deal with this problem, we propose a caching scheme using a Bloom filter index.



(a) CER that is made at time T_0 (b) Expanded rectangle R_x at time T_x

Figure 1. An example of a CER that is made at time T_0; it has been expanded until time T_x.

3. Proposed TPR*-tree with BF-caches

3.1 Problem Definition

Figure 2 depicts an example of a traditional TPR-tree structure that indexes moving objects whose positional and velocity information are saved in data pages of N_i (i = 1, 2, …). In the figure, CER_1 in Y is made to represent an area where objects O_i ($1 \le i \le 4$) could be placed. When any object in N_1 is accessed for query processing or its velocity update, compaction of CER_1 could be initiated, if needed. Since the nodes located above a leaf level can be managed in buffering memory [4, 5], our research focuses on reducing the amount of updates arising at the leaf level.

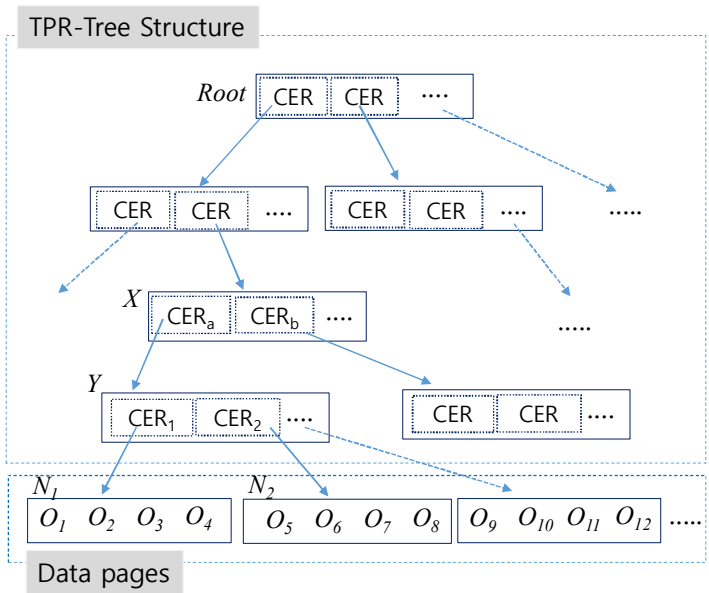


Figure 2. A snapshot of a traditional TPR-tree and some data pages indexed.

A user's query $O_{r,t}$ is composed of a target range r and a target time point t. To answer $O_{r,t}$, a search algorithm for a TPR-tree first reads its root node, and then searches down to candidate leaf nodes, by following

CER paths. During this downwards search, the shapes of CERs are computed with respect to the target time point, *t*. Through the top-down search process, the search process can collect candidate leaf's CERs to be inspected. By retrieving the data pages addressed by the collected CERs, the search process can find a resultant set of objects satisfying query $O_{r,t}$ [4, 5].

In many cases, the number of candidate CERs could be larger than the optimal one because of the existence of dead space. Therefore, it is crucial to keep CERs compact to improve the average query time. Although such CER compaction affects favorably the average query time of a TPR-tree, it should pay additional I/O costs needed for updating leaf nodes. Therefore, it is necessary to consider a trade-off between fast query times and update costs for CER compaction. In particular, the update cost comes to be a more negative factor on performance, if the TPR-tree index is used in flash storage. Recall that a random update of flash has very poor performance, compared to that of a random read [8, 10-13]. Because of this weakness of flash storage, a frequent rate of CER compaction may degrade the performance of flash-resident TPR-tree. To solve this problem, we propose a novel technique that can reduce the number of updates on leaf nodes, while keeping CERs compact.

3.2 BF-Cache for the Proposed TPR*-tree

To retain compact CERs without frequent updates of leaf nodes, our flash TPR-tree relies on a caching scheme that temporally caches compacted CERs in memory, rather than immediately updating them within a leaf node. By delaying update times of leaf nodes, our flash TPR-tree can reduce the actual number of updates arising at the leaf level.

Figure 3 illustrates the proposed flash TPR-tree, which is drawn by considering the TPR-tree of Figure 2. In the figure, a cache area is pointed to by node *X*. The paper assumes that almost all nodes except for leaf nodes reside in buffering memory. Such a buffering mechanism is very common in modern database systems [10, 13-15]. For this reason, we focus on a way to reduce updates at the leaf level, because updates of buffered nodes are not expensive.

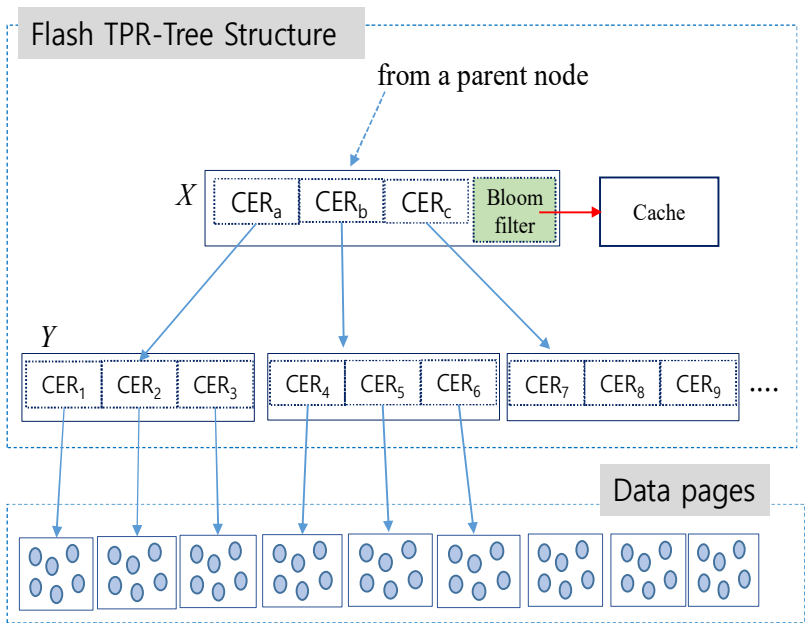


Figure 3. An example of the proposed flash TPR-tree with a Bloom filter index.

To save a Bloom filter (BF) index and a pointer to the associated cache area, we allocate a portion of space in a leaf's parent node, that is, node *X* in the example of Figure 3. In this paper, we refer to a node containing a BF index as a *BF node*, which is located only at the parent level of leaf nodes.

To see more detail about the use of BF nodes, let us suppose that CER_1 of Figure 3 is about to be updated

for compaction. At this point of time, the proposed flash TPR-tree algorithm saves a newly computed CER_l in the cache area pointed to by node *X*. To index the new CER_l in the BF cache, the flash TPR-tree computes a bit-vector of $v(x)$, where *x* is the ID of CER_l. Note that the ID of CER_l is composed of the page ID of *Y* and the offset of CER_l in Y. After computing bit-vector $v(x)$, the flash TPR-tree adds it to the BF index of *X*. By using BF node *X*, our TPR-tree can cache compacted CERs that are saved in child nodes of *X*. The search algorithms for using the BF node are presented in the next section.

3.3 Algorithms for the Proposed Flash TPR-tree

To utilize techniques already devised for the traditional TPR-tree, our search algorithm is also designed based on the previous one. The difference from the previous one is the use of BF-based caches. By using the BF cache linked from a BF node, our TPR-tree can efficiently reduce useless access to dead space.

The search algorithm for our flash TPR-tree is given in Figure 4. In lines 2-3, the algorithm collects all the BF nodes that contain any CER overlapped with a given range query, (Q_r, Q_t). This CER collecting step can be done via a traditional search algorithm. From this, in line 7 the algorithm can select all the candidate CERs that will be used for searching down to candidate leaf nodes.

If any of the candidate CERs were updated for compaction and saved in a BF cache, then the search algorithm has to access the cached CERs. For this, the search algorithm performs a membership test in line 10. This membership test is done via an OR bit-operation between a BF index and the BF bit-vector of the tested CER in line 10. If the test result is true, then a compacted CER saved in the BF cache is used for query processing as in line 12. This collecting of candidate CERs in leaf nodes is repeated through the lines of 9 – 15. With respect to each candidate CER, an associated data page is retrieved and a search algorithm is executed for gathering resultant objects. Those steps are done in lines 16-17. Finally, the set of objects satisfying the given query is returned in line 18.

Algorithm 1: *SearchUsingBloomFilter*

Input : *Root* = address to the root node of a TPR-tree
Q_r = target query range to be answered
Q_t = target query time

```
1  begin
2    N ← Tree.readNode(Root); // read of the root node into N
3    Λ_BF ← Tree.SearchBFNode(N, Q_r, Q_t); // get all the BF nodes containing any CER
       overlapped with Q_r at the time of Q_t
4    S_obj ← ∅; // initialization of a set of the query result
5    foreach τ ∈ Λ_BF do // processing for each BF node
6      N ← Tree.readNode(τ); // read of a BF node into N
7      Γ_CER ← Tree.SearchCER(N, Q_r, Q_t); // get all the candidate CERs
8      Γ'_CER ← ∅;
9      foreach ζ ∈ Γ_CER do // for each CER in node N
10       if ζ exists in the BF index of N then // membership test
11         ζ' ← get the updated CER saved in BF cache pointed to by N;
12         if IsOverlapped(ζ', Q_r, Q_t) = true then // query testing
13           Γ'_CER ← ζ' ∪ Γ'_CER; // collecting of updated CER saved in BF cache
14       else if IsOverlapped(ζ, Q_r, Q_t) = true then
15         Γ'_CER ← ζ ∪ Γ'_CER;
16     Perform a search algorithm on Γ'_CER w.r.t. query (Q_r,Q_t); // read of candidate leaf nodes
17     Add IDs of the moving objects searched in line 16 into S_obj;
18   Return the object set of S_obj as a search result;
```

Figure 4. Algorithm for returning moving objects satisfying a given range query.

The algorithm for updating a BF node is presented in Figure 5. A plausible timing for updating a BF node could be one of two cases. First, when a leaf node of a TPR-tree is about to be updated for reflecting object's velocity change, a BF node can be updated. Second, after a range query is completed by reading any leaf node *N*, the proposed TPR-tree can determine the compaction of any CER in *N*. Involved with this aggressive approach, some schemes were proposed for assessing a benefit that is obtainable by the aggressive compaction of a CER [2, 4, 6]. The update of a BF node would be possible in either of those two cases.

The update algorithm begins with node ID of a BF node being updated as well as a threshold value of dead space. In lines 2 to 4, the algorithm first reads the BF node and computes the compacted form of each CER in that node. Then, to check if this CER was already stored in a BF cache, the algorithm uses the BF index of *N*. That is, the algorithm performs an OR bit-operation for a membership test in line 6. If the result of that membership test is true, then the BF cache is updated with the newly computed CER in line 7.

Otherwise, if the testing result is not true, then the algorithm computes the size of dead space between a previous CER and that of the newly computed one. If the difference is greater than a given threshold value, then the new CER will be inserted into the BF cache through lines 9-14. During those steps, if the BF cache overflows, then the BF area should be initialized for handling the current overflow. For this initialization of the BF cache, the associated leaf nodes are updated to save the cached CERs in line 13. Finally, the BF index and BF cache are cleaned in line 14.

Algorithm 2: *UpdateBloomFilterNode*

Input : $Node$ = address to a BF node considered to be updated
$Threshold$ = threshold size of dead space tolerable

```
1  begin
2      N ← Tree.readNode(Node); // read of the BF node to N
3      foreach ζ ∈ N do // processing of each CER contained in N
4          ζ' ← Tree.compactCER(ζ); // re-computing of a CER for compaction
5          vector ← create a BF vector with a unique value of (ID of N, offset of ζ in N );
6          if (vector OR N.BF_Index) ≠ 0 then // membership test of ζ on BF index
7              Update ζ in the BF cache of N with a new compact CER of ζ';
8          else if SizeOfDeadSpace(ζ, ζ') ≥ Threshold then // check need of updating
9              if there is free space for saving ζ' then // check free space in BF cache
10                 Insert ζ' into the BF cache pointed to by N;
11                 N.BF_Index ← vector OR N.BF_Index; // update of BF index
12             else
13                 Write every CER in the BF cache of N into its correspondign leaf node;
14                 Clean the BF cache of N and set N.BF_Index to zero, correspondingly;
```

Figure 5. Algorithm for updating a BF index and its BF cache.

4. Performance Evaluation

Since a poor false-positive rate of the BF index degrades the efficiency of query processing of the proposed flash TPR-tree, we need to appropriately control it in a low range. Note that a higher false-positive rate leads to a greater number of useless accesses to BF cache areas. The false-positive rate of a BF index depends on three parameters, that is, the bit size of the BF index, the number of used hash functions, and the number of indexed items [16]. Let us denote them by m, n_k, and N, respectively. According to an earlier study [16-18], the probability of false-positive testing is computed as follows:

$$P_f = (1 - (1 - \frac{1}{m})^{n_k * N})^{n_k} \approx (1 - e^{-\frac{n_k * N}{m}})^{n_k} \quad (1)$$

Among those parameters, m and N both vary with respect to the node size of a flash TPR-tree and the size of a BF index. Since those two parameters are decisive by choosing the node size and the index's portion of space in each BF node, we just need to pick an optimal integer value for n_k with respect to given m and N. As an optimal value of n_k yielding the least P_f, we can use an integer of $\lceil m * \ln 2/N \rceil$ [8, 17].

Suppose that the node size of our flash TPR-tree is equal to 4 KB, which is a very common size in database applications [5-7]. We also assume that the data size of a CER is equal to 40 Bytes. In this case, the maximum fan-out size of a BF node becomes 0.1 K. If the space utilization of a TPR-tree lies at around 75 %, we can say that N for a BF node is about 5.6 K; its maximum value is about 10 K. Note that the maximum size of N can be computed by multiplying the fan-out size of a BF node and the number of CERs in a leaf node, that is, it becomes 0.1 K x 0.1 K.

Based on the observations above, we can develop an analytic model that assesses update costs as well as the false-positive rate of the proposed flash TPR-tree. Table 1 shows the model parameters and their values. In the table, we can adjust the portion of a BF index within a BF node from 5% to 10%. According to the storage usage of a BF index, the values of m and n_k vary in the ranges of Table 1. Based on these values, we can compute other parameters as well.

Table 1. Considered performance parameters and their values.

Parameters	Meanings & Parameter's Values
N	No. of items in a BF cache, 100
m	size of a BF index, $200 \sim 400$ Bytes
n_k	No. of hash functions, $22 \sim 44$
P_f	false-positive rate, $2.1 \times 10^{-7} \sim 4.5 \times 10^{-4}$
U_{node}	average space utilization of a node, 75%
$Fout$	average fan-out, $90 \times 0.75 \sim 95 \times 0.75$
C_{update}	update cost of a leaf node, 0.1 msec
C_{cache}	cost for accessing a BF cache, 0.1 usec
N_{reduce}	average No. of reduced updates, $N - Fout$

In Table 1, we assume that 5%-10% of a node space is allocated for the BF index for each BF node. As shown in Table 1, the false-positive rate is very low using a small size of BF indexes. With the parameters above, the performance gain of the proposed scheme is computed as follows:

$$G_{BF} = N_{reduce} * C_{update} * P_{clean} - P_f * C_{cache} + LookupCost - CampationGain \quad (2)$$

In equation (2), the value of P_{clean}, $LookupCost, CampationGain$ depend on the type of user queries issued. The parameter P_{clean} denotes the probability of the executions of the cleaning of a BF cache. In its values seem to be below $1/(Fout \times N)$. Let us supposes that the BF lookup cost and the gain from compacted CERs are almost the same. From this assumption, we can say that we gain a performance gain that lies between around 40 msec to 48 msec concerning 1K accesses to the BF node. Due to the efficiency of BF cache, the proposed flash TPR-tree can provide the reliable performance of query processing for flash-based database systems.

5. Conclusion

In this paper, we proposed a caching scheme devised for enhancing the query performance of a flash TPR-tree. Since I/O costs for random updates are very expensive in the case of flash storage, it is necessary to lay more considerations on a way to reduce the number of updates on tree's leaf nodes. However, since such updates are enforced to reduce the size of undesirable dead space, frequent updates on leaf nodes are inevitable

from the aspect of query processing times. For this reason, we elaborated a way to cache newly updated bounding rectangles of leaf nodes, thereby reducing the number of updates at the leaf level. From this, we can achieve a good balance between query processing times and low update overheads.

To index a set of cached bounding rectangles using only a tiny size of data, we employ the Bloom filter index. Although the Bloom filter index has a shortcoming of false-positive searches, it offers space efficiency and fast membership testing times. Since the rate of false-positive searches can be managed to be a low one, and it does not cause any inconsistency of data, we can employ the Bloom filter index for our research. Owing to the cached bounding rectangles accessed via the Bloom filter indexes, our flash TPR-tree can diminish updates of leaf nodes, while keeping them compact properly. The performance gains of the proposed TPR-tree are evaluated through an analytical cost model. From this, we showed that the proposed scheme offers a significant advantage from the aspect of I/O efficiency.

Acknowledgement

This work was supported by the Dongduk Women's University grant in 2020.

References

[1] B. Donnet, P. Raoult, T. Friedman, and M. Crovella, "Efficient Algorithms for Large-scale Topology Discovery," in *Proc*. of ACM SIGMETRICS, pp. 327–338, June. 2005.
DOI: doi.org/10.1145/1071690.1064256

[2] Thi Nguyen, Zhen He, Rui Zhang, and Phillip Ward, "Boosting Moving Object Indexing through Velocity Partitioning," in *Proc*. of the VLDB Endowment, pp. 860–871, May 2012.
DOI : doi.org/10.14778/2311906.2311913

[3] Rslan E., Hameed H.A., and Ezzat E., "Spatial R-tree Index based on Grid Division for Query Processing," *International Journal of Database Management Systems*, Vol. 9, No. 6, pp. 25-36, 2017.
DOI : 10.5121/ijdms.2017.9602

[4] Hendawi A.M., Bao J., Mokbel M.F., and Ali M., "Predictive Tree: An Efficient Index for Predictive Queries on Road Networks," in *Proc*. of ACM ICDE, pp. 1215–1226, 2015.
DOI: 10.1109/ICDE.2015.7113369

[5] Koide S., Tadokoro Y., Yoshimura T., and Xiao C., "Enhanced Indexing and Querying of Trajectories in Road Networks via String Algorithms," *ACM Trans. Spatial Algorithms Systems,* Vol. 4, No. 1, pp. 1-41, 2018.
DOI : doi.org/10.1145/3200200

[6] Seong-Chae Lim, "An Indexing Scheme for Predicting Future-time Positions of Moving Objects with Frequently Varying Velocities," *Journal of the Korea Society of Computer and Information*, Vol.15 No. 5, pp. 23 – 31, 2012.
DOI : doi.org/10.9708/jksci.2010.15.5.023

[7] Jianzhong Qi, Rui Zhang, Christian S. Jensen, Kotagiri Ramamohanarao, and Jiayuan HE, "Continuous Spatial Query Processing: A survey of Safe Region based Techniques," *ACM Computer Surveys*, Vol. 51, No. 3, pp. 1–39, May 2019.
DOI : doi.org/10.1145/3193835

[8] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler, "Flashing Databases: Expectations and Limitations," in *Proc. of ACM DaMon*, pp. 9-18, June, 2010.

[9] Tseng-Yi Chen, Yuan-Hao Chang, Yuan-Hung Kuan, Ming-Chang Yang, Yu-Ming Chang, and Pi-Cheng Hsiu, "Enhancing Flash Memory Reliability by Jointly Considering Write-back Pattern and Block Endurance," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 23, No. 5, pp 1–24, September 2018.
DOI : doi.org/10.1145/3229192

[10] 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.

[11] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon, "Flash-based Extended Cache for Higher Throughput and Faster Recovery," *Journal of the VLDB Endowment*, Vol. 5, No. 11, pp. 1615-1626, 2012.

[12] John Colgrove, et at., "Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components," in *Proc*. of SIGMOD, pp. 1683-1694, May 2015.
DOI : doi.org/10.1145/2723372.2742798

[13] Laura M. Grupp, John D. Davis, and Steven Swanson, "The Bleak Future of NAND Flash Memory," in *Proc. of the USENIX Conference on File and Storage*, Feb. 2012.

[14] Sungchae Lim, "A New Flash-based B+-tree with Very Cheap Update Operations on Leaf Nodes," in *Proc*. of International Conference on Engineering Technologies and Big Data Analytics, pp. 45-49, January 2016.
DOI : dx.doi.org/10.15242/IIE.E0116022

[15] Sungchae Lim , "F^2B+-tree: A Flash-aware B+-tree Using the Bloom Filter," *Asia-pacific Journal of Convergent Research Interchange*, Vol.6, No.7, pp. 21-28, July 2020.
DOI : 10.47116/apjcri.2020.07.03

[16] B. H. Bloom, "Space/time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, Vol. 13, No. 7, pp. 422–426, 1970.

[17] Benoit Donnet, Bruno Baynat, and Timur Friedman , "Retouched Bloom filters: Allowing Networked Applications to Trade off Selected False Positives against False Negatives," in *Proc*. of ACM CoNEXT, pp. 1-12, December 2006. DOI : doi.org/10.1145/1368436.1368454

[18] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo, "The Dynamic Bloom Filters," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 22, No. 3, pp. 120-133. January 2010.
DOI : 10.1109/TKDE.2009.57