Journal of Computer Science and Cybernetics, V.31, N.1 (2015), 43–53
DOI: 10.15625/1813-9663/31/1/4630

DO-TPR*-TREE: A DENSITY OPTIMAL METHOD FOR TPR*-TREE

NGUYEN TIEN PHUONG[1], DANG VAN DUC[2]

Institute of Information Technology, Vietnam Academy of Science and Technology
[1]phuongnt@ioit.ac.vn; [2]dvduc@ioit.ac.vn

Abstract. This paper proposes a density optimal method, named as DO-TPR*-tree, which improves the performance of the original TPR*-tree significantly. In this proposed method, the search algorithm will enforce firing up MBR adjustment on a node, if the condition, based on density optimal for the area of its MBR, is satisfied at a query time. So, all queries occurred after that time will be prevented from being misled as to an empty space of this node. The definition of Node Density Optimal is also introduced to be used in search algorithm. The algorithm of this method is proven to be correct in this paper. Several experiments and performed comparative evaluation are carried out. In the environment with less update rates (due to disconnected) or high query rates, the method can highly enhance query performance and runs the same as the TPR*-tree in other cases.
Keywords. DO-TPR*-tree, MODB, R-tree, TPR-tree, TPR*-

1. INTRODUCTION

The recent advances of technologies in mobile communications, GPS and GIS have increased users' attention to an effective management of location information on the moving objects. Their location data has a spatio-temporal feature, which means spatial data is continuously changing over time [1]. So it needs to be stored in a database for efficient use. Such a database is commonly termed as the moving object database [2]. A moving object database is an extension of a traditional database management system that supports the storage of location data for continuous movement. The number of moving objects in the database can be very large. Therefore, to ensure the system performance while updating or processing queries, it is needed to reduce the cost of object updates and have an efficient indexing method.

Some efforts to reduce the cost of object updates have been proposed, such as the RUM-tree [3], or FD-tree [4]. The RUM-tree processes updates in a memo-based approach that avoids disk accesses for purging old entries during an update process. Therefore, the cost of an update operation in the RUM-tree is reduced to the cost of only an insert operation [3]. Whereas, the FD-tree, a tree index that is aware of the hardware features of the flash disk (or SSD), has been proposed to optimize the update performance by reducing small random writes while preserving the search efficiency [4].

Most of the indexing methods are based on the traditional spatial indexes, especially the R-tree [5], R*-tree [6] and its extension, which is typical TPR-tree [7]. Many efforts have been done to propose efficient index structures for future time queries, such as TPR*-tree [8]. The B^{dual}-tree [9] was presented to enhance the update performance of the previous methods. In particular, the B^{dual}-tree works well in the overall performance aspects. However, the TPR*-tree is reported to provide a best performance in terms of the query processing times [9]. The research aims at improving the query processing times, so this method will be compared with the TPR*-tree in section 4.

In the TPR*-tree, the MBR adjustment is a solution to better performance, but its execution is only fired up while having update operations. That is, the TPR*-tree cannot adjust the MBR of a node N, if no indexed object in N changes its location or velocity. The size of the empty space in node N enlarges continuously for a long time, if N is a node without updates. Therefore, if query processes frequently searches into a sub-tree with rare updates, their response times will get longer.

The goal of this paper is to introduce the density optimal method for TPR*-tree, named as DO-TPR*-tree, which improves the performance of the original TPR*-tree significantly. In our proposed method, the search algorithm will enforce firing up MBR adjustment on N, if the condition, based on density optimal for the area of N's MBR, is satisfied at a query time. So, some of queries can be prevented from being misled as to an empty space of N. The algorithm of the method is also proven to be correct in this paper. Several experiments are carried out for performance evaluation. The results show that in the environment with less update rates (due to disconnected) or high query rates, this method is faster than the original method.

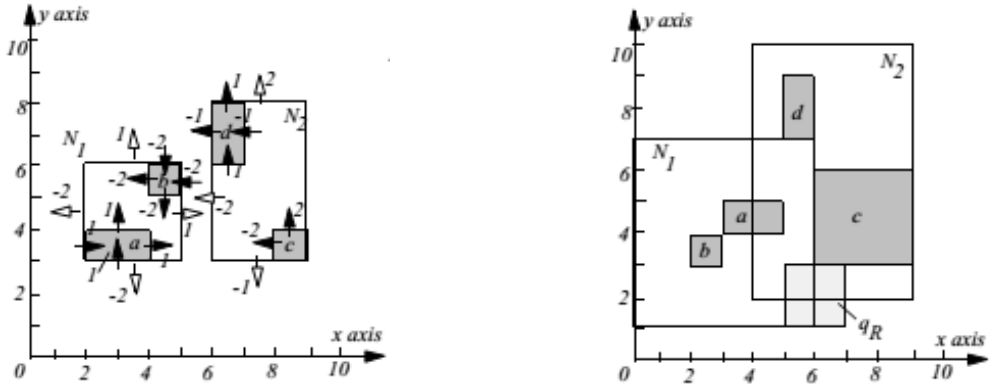
This paper is organized as follows. Section 2 briefly reviews the TPR*-treeas related work, and then introduces the research motivation. Section 3 proposes the method in detail. In the section 4, the results of performance evaluation for justifying the effectiveness of our approach are shown. Finally, section 5 summarizes and concludes this paper.

2. RELATED WORK AND RESEARCH MOTIVATION

2.1. TPR-tree

The TPR-tree [7], which has been devised based on the R*-tree [6], predicts the future-time positions of moving objects by storing the current position and velocity of each object at a specific time point.

According to [8], a moving object Ois represents a minimum bounding rectangle (MBR)O_R that denotes its extent at reference time 0, and a velocity bounding rectangle (VBR) O_V=$\{O_{V1}$-,O_{V1+},O_{V2-},$O_{V2+}\}$ where O_{Vi-}(O_{Vi+}) describes the velocity of the lower (upper) boundary of O_R along the *i-th* dimension $(1 \leq i \leq 2)$.



(a) MBRs & VBRs at time 0 (b) MBRs at time 1

Figure 1: Entry representations in a TPR-tree

Figure 1a shows the MBRs and VBRs of 4 objects a, b, c, d. The arrows (numbers) denote the directions (values) of their velocities, where a negative value implies that the velocity is towards the negative direction of an axis. The VBR of a is $a_V = \{1, 1, 1, 1\}$ (the first two numbers are for

the x-dimension), while those of b, c, d are $b_V = \{-2, -2, -2, -2\}$, $c_V = \{-2, 0, 0, 2\}$, and $d_V = \{-1, -1, 1, 1\}$ respectively. A non-leaf entry is also represented with a MBR and a VBR. Specifically, the MBR (VBR) tightly bounds the MBRs (VBRs) of the entries in its child node. In Figure 1, the objects are clustered into two leaf nodes N_1, N_2, whose VBRs are $N_{1V} = \{-2, 1, -2, 1\}$ and $N_{2V} = \{-2, 0, -1, 2\}$ (their directions are indicated using white arrows).

Figure 1b shows the MBRs at timestamp 1 (notice that each edge moves according to its velocity). The MBR of a non-leaf entry always encloses those of the objects in its sub-tree, but it is not necessarily tight. For example, $N_1(N_2)$ at timestamp 1 is much larger than the tightest bounding rectangle for a, b (c, d). A predictive window query is answered in the same way as in the R*-tree, except that it is compared with the (dynamically computed) MBRs at the query time. For example, the query q_R at timestamp 1 in Fig. 1b visits both N_1 and N_2 (although it does not intersect them at time 0). When an object is inserted or removed, the TPR-tree tightens the MBR of its parent node.

2.2. TPR*-tree

The data structure and the query processing algorithm of the TPR*-tree [8] are similar to those of the TPR-tree. A difference between them is the insertion algorithm. That is, the TPR-tree uses the original insertion algorithm of the R*-tree without modification, while the TPR*-tree makes a modification in order to reflect the mobility of objects.

In the insertion algorithm, the TPR-tree assesses the overall changes of the area and perimeter of MBRs, and the overlapping regions among MBRs that are caused by this object insertion. By choosing the tree-path where such changes remain smallest, the TPR-tree brings the least space possible. This approach can be very efficient for indexing static objects as in the R*-tree, but it cannot be a good solution to moving objects. Because the TPR-tree does not consider the time parameter, it estimates the MBR changes only found at the insertion time without considering the time-dependent sizes of the BRs. To resolve these omissions, the TPR*-tree revised the insertion algorithm to reflect the characteristic of time-varying BRs, which result from the mobility of objects.

In the insertion algorithm, the TPR*-tree considers how much the BR will sweep the index space from the insertion time to a specific future-time and chooses the insertion paths that the sweeping region remains smallest. The sweeping region of a BR from time t_1 to t_2 ($t_1 < t_2$) is defined to be an index space area that is swept by the BR expanding during the time interval $(t_2 - t_1)$. Fig. 2 shows an example of a sweeping region of an object o in the TPR*-tree. At the initial time 0, we have the BR of O is $O_R(0)$. Until time 1, the BR of O is $O_R(1)$. Sweeping region of o from time 0 to 1 is the gray-filled polygon below.

With this strategy, the TPR*-tree may consume more CPU time for inserts than the TPR-tree. However, it greatly improves the overall query performance (up to five times [8]) for the future-time queries because it compacts the MBRs.

2.3. Research Motivation

In the TPR*-tree, the VBR stores the maximum and minimum velocity of moving objects in the MBR. So the MBR enlarges at a fast and continuous rate. It leads to a large empty space and causes overlaps among nodes' MBRs as time goes on (see Fig. 1b above). It may reduce the performance of query processing because of increasing number of node accesses that require for query processing (q_R in Fig. 1b above has unnecessary node access to N_1 and N_2). To resolve this problem, the TPR*-tree

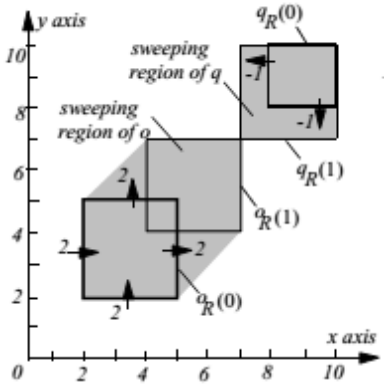
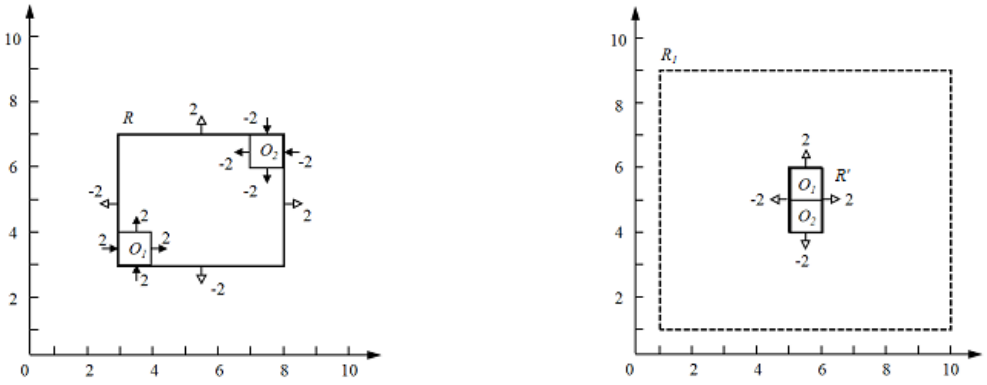


Figure 2: Sweeping region of moving object o (from time 0 to time 1)

executes MBR adjustment on a node whenever object's velocity or position have explicitly changed.



(a) Time 0 (b) Time 1

Figure 3: MBR R at the initial time 0 and its expansion R_1 at time 1

Fig. 3 illustrates the benefit of the MBR adjustment. In Fig. 3a, a MBR is created or updated to capture the objects of O_1 and O_2 at time 0 and its MBR is denoted by rectangle R. Fig. 3b depicts the positions of those objects after the time interval of 1. Objects O_1 and O_2 moved closely to each other. If a MBR adjustment arises at time 1 because of updating a node, a smaller MBR will be available (R' in Fig. 3b). In contrast, the predicted MBR will become a larger rectangle (R_1 of Fig. 3b). Therefore, the empty space of $R_1 - R'$ can be eliminated. In other words, some unnecessary node access to the area of $R_1 - R'$ can be eliminated in the near future.

The MBR adjustment is a solution to better performance, but its execution is only fired up while having update operations. That is, the TPR*-tree cannot adjust the MBR of a node N, if no indexed object in N changes its location or velocity (due to disconnected). Otherwise, the size of empty space in node N enlarges continuously for a long time, if N is a node without updates. Therefore, if query processes frequently searches into a sub-tree with rare updates, their response times will get longer. To overcome such a problem, a new method with a new tree based on TPR*-tree is proposed, named as DO-TPR*-tree, enabling the query process to do the MBR adjustment in an efficient manner.

3. THE PROPOSED METHOD

In this section, a new method for improving the query performance in the TPR*-tree is proposed. At first, the technique is presented in section 3.1. Then section 3.2 shows the detailed algorithm.

3.1. Method Description

In this section, the technique for the proposed method is described. Denote P is a query process, N is a leaf node that Preached and T_{u_j} and $T_{u_{j+1}}$ are the jth and the $(j+1)$th update times when the MBR adjustments occur on N, respectively. Assuming that there are $k$user queries accessing N in $[T_{u_j}, T_{u_{j+1}}]$ and P, is the ith query process, happens to arrives at N during the period from T_{u_j} to $T_{u_{j+1}}$. Denote the user query having released P by $Q_{j,i}$, that is, $Q_{j,i}$ is the ith user query accessing N. Let $T_{q_{j,i}}$ be the access time of $Q_{j,i}$ to N resulting in an arrival sequence as below.

$$T_{u_j} < T_{q_{j,i}} < T_{q_{j,2}} < \ldots < T_{q_{j,k}} < T_{u_{j+1}}$$

Because the MBR of Nenlarges continuously during $[T_{u_j}, T_{u_{j+1}}]$, the user queries at $T_{q_{j,i}}$ $(1 \leq i \leq k)$ will view the growing MBR of $N$and thus the possibility of their misleading to Nincreases during that interval. Note that if there is any query having an overlap between its target query region and the empty space of N, then the query will uselessly access Nbecause of misleading.

With the observation above, it is now back to the situation when the query process of $Q_{j,i}$ arrives at N. If this process is able to do its MBR adjustment on Nat time $T_{q_{j,i}}$, then some of the queries $Q_{j,x}(i¡x \leq k)$ can be prevented from being misled as to an empty space of N during $[T_{q_{j,i}}, T_{u_{j+1}}]$.

In proposed method, the search algorithm will enforce firing up MBR adjustment on N, if the condition, based on density optimal for the area of N's MBR, is satisfied at time $T_{q_{j,i}}$. Two definitions are introduced to be used in the following search algorithm.

Definition 1 (Node Density)**.** Given N is a node of TPR*-tree. Node Density of N is the number of entries per unit of the area of N's MBR. Node Density of Nat time T, denoted $D_N(T)$, is the number of entries per unit of the area of N's MBR at time T.

$$D_N(T) = \frac{Num\_E_N(T)}{ABR_N(T)} \tag{1}$$

where, $Num\_E_N(T)$ is the number of the entries inside N at time T. If N is a leaf node, $Num_E_N(T)$is the number of all moving objects inside N. $ABR_N(T)$ is the area of N's MBR at time T.

Definition 2 (Node Density Optimal)**.** Node Density of N at query time T_q is called optimal if its ratio and Node Density of Nat the last update time T_u is smaller than a given number λ.

$$\frac{D_N(T_q)}{D_N(T_u)} < \lambda \tag{2}$$

where, $D_N(T_q)$ is Node Density of N at query time T_q, $D_N(T_u)$ is Node Density of N at the last update time T_u, λ is a given number, depending on the specific application. In the Vehicle Management System, λ should be $h-1$, h is the tree height, according to our evaluation.

In the search algorithm of our proposed method, Node Density Optimal condition (2) of N is checked while user queries are being at time $T_{q_{j,i}}$. If not, MBR adjustments arise on N. So, all queries $Q_{j,x}(i < x \leq k)$, that occur after time will be prevented from being misled as to an empty space of N during $[T_{q_{j,i}}, T_{u_{j+1}}]$. From that, the number of useless node access to N can be reduced. Thus, it improves the query performance of the TPR*-tree. When checking formula (2) the MBR adjustment is called Density Optimal Adjustment (DOA). The tree in density method is based on TPR*-tree, so it is called DO-TPR*-tree. Details about this DOA and the search algorithm for the query process are presented in the next section.

3.2. Algorithm

Algorithms for node insertion and deletion are identical with the previous ones in [8]. According to [8], the insertion algorithm first identifies the leaf N that will accommodate $e$with the *choose path* algorithm in line 3. If N is full, a set of entries, selected by *pick worst*, is removed from N and re-inserted in line 7.

Algorithm Insert (*e*)
Input: *e* is the entry to be inserted
Output: Inserted *e* into Node *N*
1. *re-inserted*$_i$= false for all levels $1 \leq i \leq h-1$ (*h* is the tree height)
2. Initialize an empty re-insertion list $L_{reinsert}$
3. Invoke **Choose Path** to find the leaf node *N* to insert *e*
4. Invoke **Node Insert** (*N*, *e*)
5. For each entry *e'* in the *Lreinsert*
6. Invoke **Choose Path** to find the leaf node *N* to insert *e'*
7. Invoke **Node Insert** (*N*, *e*)
8. End for
Algorithm End.

Figure 4: Insert algorithm

A future-time query from a user is expressed in the form of (Q_{BR}, Q_T). Here, Q_{BR} denotes the query target region (includes MBR and VBR) in the 2-dimensional query space, and Q_T is the target query time interval of that query (includes start time and end time). The future-time query is answered by predicting the moving objects that will pass Q_{BR} during the interval of Q_T. The proposed search algorithm is given in Fig. 4, where it returns a set of object ids satisfying a query of (Q_{BR}, Q_T).

The algorithm searches down for the leaf level from the root node. During the downward search, the search process finds its search paths by computing the BRs from each entry e in the internal nodes. In the meantime, the process will cache the encountered BR data as in line 11. When arriving at a leaf node in line 5, this algorithm selects a group of object id's satisfying the given query condition, and return them in line 21.

Theorem 1 (Algorithm DOA_Search is correct)**.** *Algorithm DOA_Search returns the complete set of intersecting valid objects.*

Proof. Every node of the tree is traversed. This can be seen in line 3 (every leaf node) and in line 9 (every non-leaf node) (i).

```
Algorithm DOA_Search(Q_BR, Q_T, rootNode)
  Input: (Q_BR, Q_T) = future-time query answered.
         rootNode= address to the root of a sub-tree being searched.
  Output:S_BR= objects' ids satisfying (Q_BR, Q_T).
1. IF rootNode is a leaf node THEN
2.  S_BR←∅;     // initialization
3.  FOR EACH moving object O in rootNode
4.    IF OIN (Q_BR, Q_T) THEN  //O is expected computed to be positioned d in Q_BR within interval
    Q_T
5.          S_BR= S_BR∪ O.id;
6.    END IF
7.  END FOR
8.  ELSE  // rootNode is a non-leaf node
9.  FOR EACH index entry e in rootNode
10.    IF OVERLAP(e.MBR,Q_BR,Q_T) THEN // if there is an overlap between the MBR of e and Q_BR
       within Q_T
11.        Cache(e); // cache the content of e into the memory buffer
12.        nodeDensity = Cal_NodeDensity(e.childNode);  // calculate the Node Density of the
           child node pointed to by e
13.        λ = h-1; // h is the tree height
14.        IF (nodeDensity>= λ) THEN // an DOA execution is needed on childNode
15.            Adjust (e.MBR, e.VBR);  // adjust the MBR and VBR data of e
16.        END IF
17.        childS_BR←DOA_Search(Q_BR, Q_T, childNode);
18.        S_BR ← S_BR∪childS_BR;
19.    END IF
20.  END FOR
21. RETURN S_BR
22. END IF
```

Figure 5: Proposed Search algorithm for the query process

For every leaf node traversed, all of its objects which locate in query region at query time interval are inserted into the result set of the corresponding query. This can be seen from line 4 to 5 (ii).

For every non-leaf node traversed, all of its entries which intersect at least one of the range queries arerecursive searched to find the satisfy leaf nodeand inserted into the result set of the corresponding query (iii).

The algorithm terminates when all the nodes in the tree are traversed (iv).

From (i), (ii), (iii) and (iv) we have proved that DOA_Search algorithm is correct.

We next estimate the performance in our method according to the range query size and compare it with the original TPR*-tree. □

4. EXPERIMENT

In this section, the query processing performance between this proposed method and the original TPR*-tree is compared. From the experiment comparisons, the performance advantages of the proposed method over the TPR*-tree are shown.

4.1. Experiment data

The experimental datasets is generated by using an algorithm same as GSTD (Generating Spatio-Temporal Datasets) [10], a well-known data generator used in many previous researches on perfor-

mance evaluations of the index schemes for moving objects [11, 12]. With this algorithm, four datasets include [1000, 10000, 50000, 100,000] moving objects are generated with a random velocity each in the range of [-50, 50] and maximum velocity changes in each update to 5. Those objects are simulated to move around within the normalized 2-dimensional query space of the size 0 by 10,000. In that query space, an object is represented as a point and its initial position is uniform distribution.

Data file was created as a text file containing the records of the object. The objects were randomly generated at time points t_0 (including identification, minimum bounding rectangle, velocity, reference time of object). At time t_1, t_2, t_3,... random number of objects are generated with velocity changes. Detail of the record is described as a structure below:

```
struct MovingObject
{
int oi;  // unique identification of object
float mbr[4];  // MRB of object
float vbr[4];  // VRB of object
float re;  // reference time at which the object
is inserted or updated
}
```

In the structure, *oid* is the unique identification of object, *mbr* an array with 4 elements describe the location and size of the object, *vbr* an array with 4 elements describe the edges' velocities of the *mbr* and *ref* a value for the time at which the object was inserted or updated. Sample data are shown in Table 1 below.

oid	X1	X2	Y1	Y2	Vx1	Vx2	Vy1	Vy2	ref time
0	3383.691	3383.713	6253.745	6253.767	-13.148	-13.148	-46.418	-46.418	0
1	1463.102	1463.125	1174.597	1174.619	29.806	29.806	18.544	18.544	0
2	699.142	699.164	1529.711	1529.734	-24.100	-24.100	4.767	4.767	0
3	8125.313	8125.335	747.351	747.373	-24.063	-24.063	38.967	38.967	0
4	1899.701	1899.724	7856.583	7856.605	11.642	11.642	5.390	5.390	0
...									
520	1721.404	1721.427	7964.688	7964.710	9.970	9.970	-5.254	-5.254	1
521	9796.890	9796.912	1979.648	1979.671	27.163	27.163	-26.776	-26.776	1
522	6352.922	6352.944	9722.270	9722.292	49.548	49.548	-25.330	-25.330	1
...									
850	1960.613	1960.635	4836.069	4836.092	-22.374	-22.374	-36.063	-36.063	3
851	6955.288	6955.311	4882.243	4882.265	-1.135	-1.135	-10.259	-10.259	3
...									

Table 1: Sample data of moving objects

The experiments are performed on a Windows 7 equipped with a Pentium 4 processor of 2

GHz and 1 GB of main memory. By comparing this method with the original TPR*-tree, its good performance features will be analyzed.

4.2. Experiment Results

4.2.1. Effect of range query size

In the experiment, the average number of retrieved data, the average number of accessed node and the query execution time are measured. In experiment, the size of Q_{BR} is [100, 100]. The last update timestamp of the DO-TPR*-trees is 10. The tables below describe results of the future time queries with the query time interval $Q_T$from 10 to [20,30,40,50] in each dataset.

Q_T	Avg. data retrieves	Avg.node access	Total Time (seconds)
10-20	3,62	10,04	0,31
10-30	5,66	10,98	0,37
10-40	6,42	13,02	0,42
10-50	5,48	14,18	0,42

Table 2: The future time queries on DO-TPR*-tree of 1K Dataset

Q_T	Avg. data retrieves	Avg.node access	Total Time (seconds)
10-20	47,12	34,74	1,3419
10-30	41,74	42,1	1,5522
10-40	52,4	47,62	1,9027
10-50	41,6	53,98	1,9928

Table 3: The future time queries on DO-TPR*-tree of 10 K Dataset

Q_T	Avg. data retrieves	Avg.node access	Total Time (seconds)
10-20	225,12	82,82	5,06
10-30	233,70	102,08	6,40
10-40	254,14	119,48	8,92
10-50	288,00	133,26	11,33

Table 4: The future time queries on DO-TPR*-tree of 50 K Dataset

Its experimental results are depicted in Figures 6. The average number of retrieved data (Fig. 6a), the average number of accessed node (Fig. 6b) and the query execution time (Fig. 6c) are given on the vertical axes, while the future query time interval Q_T is shown on the horizontal axis. The graphs show how the average number of retrieved data (Fig. 6a), the average number of accessed node (Fig. 6b) and the query execution time (Fig. 6c) change with the expansion of the future query time interval Q_T.

Q_T	Avg. data retrieves	Avg.node access	Total Time (seconds)
10-20	473,32	128,04	12,83
10-30	493,52	153,60	19,64
10-40	485,58	174,10	24,18
10-50	475,28	197,40	28,23

Table 5: The future time queries on DO-TPR*-tree of 100 K Dataset

These experiment results show that the average number of accessed node and the query execution time increase fast as the future prediction time goes farther (the size of Q_T increases).

4.2.2. Performance comparisons

In experiment, the performance of this method, DO-TPR*-tree is compared with the original TPR*-tree. Its experimental results are depicted in figure 7, where the graphs show how the average number of retrieved data (Fig. 7a), the average number of accessed node (Fig. 7b) and the query execution time (Fig. 7c) change with the expansion of the future query time interval Q_T.

These experiment results show that the average numbers of retrieved data are the same but the average numbers of accessed node and the query execution time of our method reduce by up to 30%. Thus, this method is faster than the original method.

From the experiments, it has verified that this method is more effective than the original method of TPR*-tree. In general, in the environment with much chance of DOA executions, with less update rates or high query rates, the method can enhance query performance. Conversely, if there is no chance to execute the DOA, this method will run the same as the TPR*-tree.

5. CONCLUSION

In this paper, the density optimal method for TPR*-tree, named as DO-TPR*-tree is introduced, which improves the performance of the original TPR*-tree significantly by capability of executing the MBR adjustment during query processing. The algorithm of the method is also proven to be correct in this paper. Several experiments and performed comparative evaluation are carried out. In the environment with less update rates (due to disconnected, typically in Vietnam by telecommunication network infrastructure) or high query rates, this method can highly enhance query performance and runs the same as the TPR*-tree in other cases. Consequently, compared to the TPR*-tree, the proposed method is more suitable to process the future-time queries in all situations.

ACKNOWLEDGMENT

This research has been funded by the Research Project, VAST01.04/14-15, Vietnam Academy of Science and Technology.

REFERENCES

[1] D. L. Lee, W. C. Lee, J. Xu, and B. Zheng, "Data management in location-dependent information services: Challenges and issues," *IEEE Pervasive computing*, vol. 3, no. 3, pp. 65–72, 2002.

[2] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving objects databases: Issues and solutions," in *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*. IEEE, 1998, pp. 111–122.

[3] Y. N. Silva, X. Xiong, and W. G. Aref, "The RUM-tree: supporting frequent updates in R-trees using memos," *The VLDB Journal*, vol. 18, no. 3, pp. 719–738, 2009.

[4] Y. Li, B. He, Q. Luo, and K. Yi, "Tree indexing on flash disks," in *Proceedings of ICDE'09, IEEE Computer Society*. IEEE, 2009, pp. 1303–1306.

[5] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of ACM SIGMOD84*, vol. 14, no. 2. ACM, 1984, pp. 47–57.

[6] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of ACM SIGMOD Record*, vol. 19, no. 2. ACM, 1990, pp. 322–331.

[7] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," in *Proceedings of ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 331–342.

[8] Y. Tao, D. Papadias, and J. Sun, "The TPR*-tree: an optimized spatio-temporal access method for predictive queries," in *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB Endowment, 2003, pp. 790–801.

[9] M. L. Yiu, Y. Tao, and N. Mamoulis, "The B^{dual}-Tree: indexing moving objects by space filling curves in the dual space," *The VLDB Journal*, vol. 17, no. 3, pp. 379–400, 2008.

[10] Y. Theodoridis, J. R. Silva, and M. A. Nascimento, "On the generation of spatiotemporal datasets," in *Proceedings of the 6th International Symposium on Advances in Spatial Databases*. Springer, 1999, pp. 147–164.

[11] B. Lin and J. Su, "On bulk loading TPR-tree," in *Proceedings IEEE International Conference on Mobile Data Management*. IEEE, 2004, pp. 114–124.

[12] D. Pfoser, C. S. Jensen, Y. Theodoridis *et al.*, "Novel approaches to the indexing of moving object trajectories," in *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000, pp. 395–406.

Received on August 27 - 2014
Revised on March 14 - 2015