A New Approach for Moving Objects in Spatiotemporal databases

K. Appathurai[1] *Dr. S. Karthikeyan*[2]

Abstract

Spatiotemporal access methods are secret into four categories: (1) Indexing the past data (2) Indexing the current data (3) Indexing the future data and (4) Indexing data at all points of time. All the above categories are having set of indexing structure algorithms [1, 2, 3,14]. In this paper we consider the third group and proposed the new approach for the choose path method of TPR* tree. This new loom gives better performance than existing TPR* tree.

Keywords – Index, Query, Access Methods, Choose path Method and Insertion Method.

I Introduction

Spatio-temporal databases deals with moving objects that change their locations over time. In common, moving objects account their locations obtained via location-aware instrument to a spatio-temporal database server. The server store all updates from the moving objects so that it is capable of answering queries about the past [4, 5, 9, 10,15]. Some applications need to know current locations of moving objects only. This case, the server may only store the current status of the moving objects. To predict future positions of moving objects, the spatio-temporal database server may need to store additional information, e.g., the objects' velocities [8]. Many query types are maintained by a spatio-temporal database server, e.g., range queries "Find all objects that intersect a certain spatial range during a given time interval", k-nearest neighbor queries "Find k restaurants that are closest to a given moving point", or trajectory queries "Find the trajectory of a given object for the past hour". These queries may execute on past, current, or future time data. A large number of spatio-temporal index structures have been proposed to support spatio-temporal queries efficiently [12, 13]. This paper is based on [5].

II Related Work

2.1 R* Tree

The R*-tree as an extension of the B-tree for multi-dimensional static objects. Figure 2.1 shows a 2D example where 10 rectangles ($a,b,\ldots,j$) are clustered according to their spatial proximity into 4 leaf nodes $N1,\ldots,N4$, which are then recursively grouped into nodes $N5$, $N6$ that become the entries of the root. Each entry is represented as a minimum bounding rectangle (MBR). The MBR of a leaf entry Indicates the extent of an object, while the MBR of a non leaf entry (e.g., $N1$) tightly bounds all the MBRs (i.e., a,b,c) in its child node. The R*-tree is optimized for the window query, which retrieves all the objects that intersect a query region. In Figure 2.1, for example, the query visits the root of the R-tree, $N6$, $N4$, and returns object i.

[1]Asst.Prof. and Head, Department of Information Technology, Karpagam University Email : k_appathurai@yahoo.co.uk

[2]Director, School of computer Science, Karpagam University, Coimbatore – 21 Email : skaarthi@gmail.com

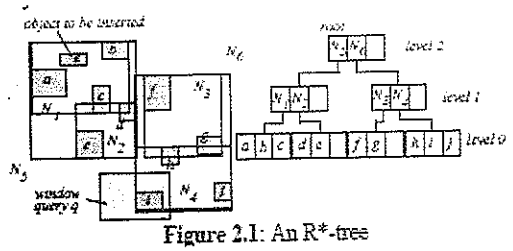
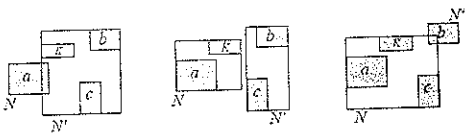


Figure 2.1: An R*-tree

The R*-tree construction algorithm aims at minimizing the following *penalty metrics*: (i) the area, (ii) the perimeter of each MBR, (iii) the overlap between two MBRs (e.g., *N1*,*N2*) in the same node, and (iv) the distance between the centroid of an MBR (e.g., *a* in Figure 2.1) and that of the node (e.g., *N1*) containing it. Minimization of these metrics decreases the probability that an MBR intersects a query region. Given a new entry, the insertion algorithm decides, at each level of the tree, the branch to follow in a greedy manner. Assume that we insert an object *k* into the tree in Figure 2.1. At the root level, the algorithm chooses the entry whose MBR needs the least area improvement to cover *k*; *N5* is selected because its MBR does not need to be enlarged, while that of *N6* must be prolonged considerably. Then, at the next level (i.e., child node of *N5*), the algorithm chooses the entry whose MBR growth leads to the smallest overlap increase among the sibling entries in the node. Note that different metrics are considered at level 1 (leaf nodes are at level 0) and higher levels. An *overflow* occurs if the leaf node is full. In this case the algorithm attempts to remove and re-insert a fraction of the entries in the node, trying to avoid a split if any entry could be assigned to other nodes. The set of entries to be re-inserted are hose whose centric distances are among the largest 30%. In Figure 2.1, *b* is selected since its centroid is the farthest from that of *N1*. Node splitting is performed if the overflow persists after the re-insertion. The R* split algorithm consists of two steps. The first step decides a split axis (from the *x*-, *y*-dimensions) as the one with the smallest *overall perimeter* computed as follows. Figure 2.2 continues the example, which, for simplicity, omits this minimum node utilization constraint (we assume that a node can have a single entry, which corresponds to 33% utilization). The 1-3 division (Figure 2.2a), for instance, allocates the first entry (of the sorted

list) into *N*, the other 3 entries into *N*2 . The algorithm computes the perimeters of *N* and *N'*, and performs the same computation for the other (2-2, 3-1) divisions. A second pass repeats this process with respect to the MBRs' right boundaries. Finally, the overall perimeter on the *x*-axis equals the sum of all the perimeters obtained from the two passes.



(a) 1-3 division (b) 2-2 division (c) 3-1 division

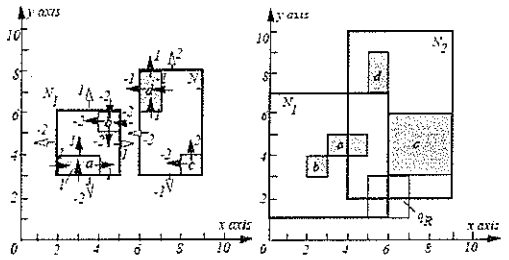
Figure 2.2: Possible divisions in splitting *N1* on the *x*-axis

After deciding the split axis, the split algorithm sorts the on the selected dimension, and gain, examines all possible divisions. The final division is the one that has the minimum overlap between the MBRs of the resulting nodes. Continuing the previous example, assume that the split axis is *x*; then, among the possible divisions in Figure 2.2, the 2-2 incurs zero overlap (between *N* and *N*2) and thus becomes the final splitting.

2.2 TPR tree

A moving object *o* is represented with (i) an MBR *oR* thatdenotes its extent at reference time 0, and (ii) a velocity bounding rectangle (VBR) $oV=\{oV1-,oV1+,oV2-,oV2+\}$ where $oVi-$ ($oVi+$) describes the velocity of the lower (upper) boundary of *oR* along the *i*-

th dimension ($1 \le i \le 2$). Figure 2.3a 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 aV={1,1,1,1} (the first two numbers are for the xdimension), while those of b,c,d are bV={-2,-2,-2,-2}, cV={-2,0,0,2}, and dV={-1,-1,1,1} respectively. A non-leaf entry is also represented with an MBR and a VBR. Specifically, the MBR (VBR) tightly bounds the MBRs (VBRs) of the entries in its child node. In Figure 2.3a, the objects are clustered into two leaf nodes N1, N2, whose VBRs are N1V={-2,1,-2,1} and N2V={-2,0,-1,2} (their directions are indicated using white arrows). arrows).



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

Figure 2.3: Entry representations in a TPR-tree

Figure 2.3b shows the MBRs at timestamp 1. 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, N1(N2) at timestamp 1 is much larger than the tightest bounding rectangle for a,b (c,d). A predictive [16] indow 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 qR at timestamp 1 in Figure 2.3b visits both N1 and N2 (although it does not intersect them at time 0). The TPR-tree is optimized for timestamp queries in interval [TC, TC+H], where TC is the current update time, and H is a tree parameter called the horizon. The update algorithms are exactly the same as those of the R*-tree, by simply replacing the four penalty metrics of the previous section with their integral counterparts. returns the overlapping area (centroid distance) between N1 and N2 at time t. These integrals are solved into closed formulae. When an object is inserted or removed, the TPR-tree tightens the MBR of its parent node. Figure 2.4 shows the MBRs after inserting a new object e (into N1) at time 1. N1is adjusted to the tightest MBR bounding a,b,e, by computing their respective extents at time 1. Note that this does not compromise the update cost because N1 must be loaded (written back) from (to) the disk anyway to complete the insertion. On the other hand, the MBR of N2 is not tightened because it is not affected by the insertion.

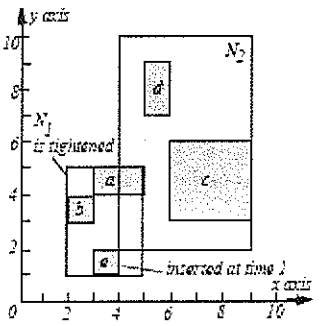


Figure 2.4: N1 is tightened during an insertion at time 1

2.3 TPR* Tree

The TPR*-tree improves the TPR-tree by employing a new set of insertion and deletion algorithms that aim at minimizing cost. This raises the question about the choice of appropriate parameter values used for optimization. We optimize the TPR*-tree for the static point interval query q, whose (i) MBR has length |qRi|=0 on each axis, (ii) VBR={0,0,0,0}, and (iii) query interval qT ={0,H},

where H is the horizon parameter (also used in the original TPR-tree). As shown in the experiments, this choice leads to nearly-optimal performance independently of the query parameters.

- **Insertion**

Figure 2.3.1 shows the high level description of the TPR* insertion. Specifically, given a new entry e at insertion time TI, the TPR*-tree first identifies the leaf N that will accommodate e with the choose path algorithm. If N is full, a set of entries, selected by pick worst, are removed from N and re-inserted. Any leaf node that overflows during the re-insertion will be split using node split, after which a new entry will be added to the parent node. This may cause the parent to overflow, and is handled in a similar way. Next we elaborate choose path, pick worst, and node split, and explain why the corresponding algorithms in the TPR-tree are not efficient.

Algorithm Insert (e)

/* Input: e is the entry to be inserted. */

1. re-insertedi=false for all levels

   1ddiddhd1 (h is the tree height)

2. initialize an empty re-insertion list

   Lreinsert

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) End Insert

Algorithm Node Insert (N, e)

/* Input: N is the node where entry e is inserted */

1. if N is a leaf node

2. enter the information of e

3. if N overflows

4. if re-inserted0=false //no re-insertion at leaf level yet

5. invoke Pick Worst to select a set

   Sworst of entries

6. remove entries in Sworst from N; add

   them to Lreinsert

7. re-inserted0=true

8. else

9. invoke Node Split to split N into itself and N'

10. let P be the parent of N

11. Node Insert(P, δ) or Node Insert(P,N') if N has been split

12. else //N is a non-leaf node

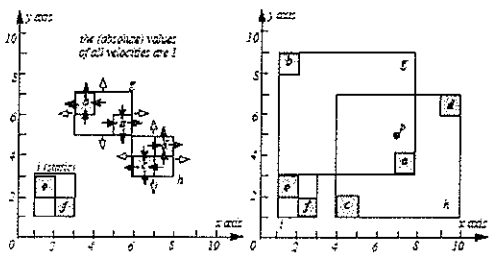
13. similar to lines 2-9 except that (i) the MBR/VBR of the affected child node is adjusted, and (ii) in lines 4, 7 replace re-inserted0 with re-insertedi where i is the level of N End Node Insert

Figure 2.3.1 : Overview of the TPR* insertion algorithm

• Choose Path

Given a new object, the traditional TPR-tree selects, at each non-leaf level, the branch with the smallest deterioration (in terms of certain penalty metrics) to continue the insertion. The efficiency of this "greedy" approach drops considerably, if multiple branches have

the same (zero) deterioration. To illustrate this, we use Figure 2.5a that shows 6 leaf nodes a,b,...,f with their parent nodes g,h,i that are the entries of the root (the absolute values of all velocities are 1). Note that although the MBRs of g,h are disjoint at time 0, they overlap significantly at timestamp 2 (Figure 2.5b). Consider the insertion of (static) point p at time 2. At the root level, g and h have no deterioration because inserting p into either one does not expand the corresponding MBR / VBR. In this case the algorithm must rely on the "tie-breaking" conditions which, however, are much less effective. In the example, h is preferred because it has smaller MBR, inside which the best leaf node to include p is d. The best choice, however, is to insert p to node a, as it requires significantly smaller MBR expansion than d. Note that this problem becomes even more serious as time progresses and the overlaps between MBRs become increasingly larger. Eventually, the greedy algorithm becomes almost random, i.e., it just picks one of the numerous candidate branches with zero penalty. The problem is less serious in R-trees (i.e., static data) where the MBRs do not grow with time.



(a) MBRs & VBRs at time 0 (b) MBRs at time 2

Figure 2.5 : Inserting p at time 2

Motivated by this, we propose a *choose path* algorithm which, given a new object, returns the insertion path with the minimal increase in equation 3-1 (called cost degradation in the sequel) among all the paths. Towards this, choose path maintains a priority queue QP that records the candidate paths inspected so far. In Figure 2.5b, at the root QP is initiated with {[(g),0], [(h),0],[(i),20]}, where each number indicates the cost degradation (for the static point interval query with qT=[0,1]), if p is inserted into the corresponding path. The degradation is 0 for g and h because, as mentioned earlier, their MBRs/VBRs do not need to be expanded. Note that at this point we have not accessed any of nodes g,h,i, i.e., the cost degradation is computed from their extents stored in the root. At each step, *choose path* explores the path with the smallest cost degradation. In this example, it visits node g and inserts two paths (a,g) and (b,g) in QP, after which QP = {[(h),0], [(a,g),3], [(i),20], [(b,g),32]}. Notice that (a,g) and (b,g) are *complete*, meaning that they include the leaf level (although leaves a and g are not visited). Similarly, the next path expanded is (h), and QP becomes QP={[(a,g),3], [(d,h),9], [(c,h),17], [(i),20], [(b,g),32]}. Now the algorithm terminates with (a,g) as the overall best path, because its (accumulated) cost degradation is smaller than that of all the other entries in QP. Note that [(i),20] is not explored at all, as it already incurs larger degradation at the highest level. *Choose path* finds the best insertion path at the cost of some extra node accesses. This, however, pays off due to the following reasons. First, it leads to a better tree structure, which improves the query performance. Second, our experiments show that in most cases it only needs to explore on average 2-3 complete paths because most paths will terminate at very high levels (e.g., (i) in Figure 2.5b). Third, *choose path* only visits non-leaf nodes that usually reside in the buffer. Fourth, each update in spatiotemporal databases usually involves one deletion (followed by an insertion), which as explained in the next section, is usually the dominating factor in the total update cost. The deletion requires a query to locate the object to be removed. Due to its

improved query performance with respect to the TPR-tree, the update overhead of the TPR*-tree is much lower. A similar situation exists for the relative performance of updates in R*- and R-trees; although the R*-tree involves more complex insertion operations, it results in faster updates due to its better structure.

- **Pick Worst**

Insertion to a full node generates an overflow, in which case both TPR- and TPR*-trees re-insert a fraction of the entries from the node. The TPR-tree, following the strategy of R*-trees. So the reinsertion becomes useless and a node split must occur. In general, entries selected in this manner are usually those that move away most quickly from the centroid of the bounding MBR, instead of those that decide the extents. Again, this problem is not important for conventional R trees. In *pick worst* returns a set of entries whose removal reduces the MBR or VBR of the parent node.

- **Node Split**

Similar to TPR-trees, the split algorithm of the TPR*-tree computes the overall perimeter for each dimension *i*, by considering all possible divisions of the entry list sorted according to the starting/ending points of their extents on this dimension. Then, the split axis is selected as the one with the smallest overall perimeter.

- **Deletion**

To remove an object e whose (i) MBR at the deletion time TD is eR(TD), and (ii) VBR is eV, the deletion algorithm first identifies the leaf node that contains e, by searching the tree using eR(TD) as the query window. Specifically, a node o is visited if and only if (i) its MBR o(TD) at time TD contains eR(TD), and (ii) its VBR oV contains eV. A difference from normal window queries is that the search terminates as soon as e is found.

III Statement of Problem

In TPR* Tree, the node insertion algorithm takes more time for searching the free leaf node. In particular the choose path method the greedy approach is followed. This becomes not optimized. So the performance is decreases.

IV Proposed Work

The proposed algorithm updates all the objects data better than TPR* Tree -index structure. In TPR* Tree index algorithm, the time taken for selecting the new node for insertion is too long. So the performance is very low in case of moving Objects. Using existing algorithm with some modifications in Choose Path Function, It is possible to reduce the searching time for node selection. So that all the objects data are updated better than TPR* Tree –index.

The following is the proposed algorithm i.e TPRC* Tree -index. In TPRC* Tree -index Choose path function maintain a heap method, instead of searching all the entire tree for finding the free node, we search few more nodes for finding the free node, so automatically the searching time is reduced and quickly stored the data from moving objects.

Algorithm Insert (z)

/* Input: z is the entry to be inserted. */

1. re-insertedi = false for all levels 1d"id"h"1 (h is the tree height)

2. initialize an empty re-insertion list Lreinsert

3. invoke Choose Path to find the leaf node N to insert z

4. Invoke Node Insert(N, z)

5. for each entry z' in the Lreinsert

6. invoke Choose Path to find the leaf node N to insert z'

7. Invoke Node Insert(N, z) End Insert

Algorithm Node Insert (N, z)

/* Input: N is the node where entry e is inserted */

1. if N is a leaf node

2. enter the information of z

3. if N overflows

4. if re-inserted0=false //no re-insertion at leaf level yet

5. invoke Pick Worst to select a set Sworst of entries

6. remove entries in Sworst from N; add them to Lreinsert

7. re-inserted0=true

8. else

9. invoke Node Split to split N into itself and N'

10. let P be the parent of N

11. Node Insert(P,") or Node Insert(P,N') if N has been split

12. else //N is a non-leaf node

13. similar to lines 2-9 except that (i) the MBR/VBR of the affected child node is adjusted, and (ii) in lines 4, 7 replace re-inserted0 with re-insertedi where i is the level of N End Node Insert

In Choose path function, Cheap method is followed for finding the path i.e [(g,0], [(h),0], [(i), 15] here g is the path expanded 0 for accumulated penalty shown in the figure 4.1. using simulation method the betterment is conformed.

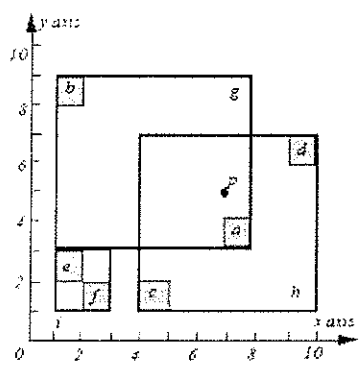


Figure 4.1 inserting p at time 2

V Conclusion

This paper presents a novel indexing technique, the TPRC* Tree –index. which can answer queries about the past, the present. The TPRC* Tree index is based on TPR* Tree -index. it avoids duplicating objects while indexing of chronological information and thus achieves major space reduction and proficient query processing. we improve choose path function of node insertion, so that greatly reduce the searching time for finding of free nodes for node insertion. So that the performance has improved.

References

[1] Long-Van Nguyen-Dinh, Walid G. Aref, Mohamed F. Mokbel 2010. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). Bulletin of the IEEE Computer SocietyTechnical Committee on Data Engineering

[2] M. Pelanis, S. ¡ Saltenis, and C. Jensen. Indexing the past, present, and anticipated future positions of moving objects.TODS, 31(1):255–298, 2006.

[3] Z.-H. Liu, X.-L. Liu, J.-W. Ge, and H.-Y. Bae. Indexing large moving objects from past to future with PCFI+-index. In COMAD, pages 131–137, 2005.

[4] V. Chakka, A. Everspaugh, and J. Patel. Indexing large trajectory data sets with SETI. In CIDR, 2003

[5] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In VLDB, 2003.

[6] D. Lin, C. Jensen, B. Ooi, and S. ¡ Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In MDM, pages 59–66, 2005.

[7] C. Jensen, D. Lin, and B. Ooi. Query and update efficient B+-tree based indexing of moving objects. In VLDB, 2004.

[8] M. Mokbel, T. Ghanem, andW. G. Aref. Spatio-temporal access methods. IEEE Data Eng. Bull., 26(2):40–49, 2003.

[9] J. Ni and C. V. Ravishankar. PA-tree: A parametric indexing scheme for spatio-temporal trajectories. In SSTD, 2005.

[10] P. Zhou, D. Zhang, B. Salzberg, G. Cooperman, and G. Kollios. Close pair queries in moving object databases. In GIS, pages 2–11, 2005.

[11] Dan Lin, Christian S. Jensen, Beng Chin Ooi, Simonas S¡ altenis, BBx index :Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects, MDM 2005 Ayia Napa Cyprus

[12] P. K. Agarwal and C. M. Procopiuc. Advances in Indexing for Mobile Objects. IEEE Data Eng. Bull., 25(2): 25–34, 2002.

[13] G. Kollios, D. Gunopulos, V. J. Tsotras. On Indexing Mobile Objects. In Proc. PODS, pp. 261–272, 1999.

[14] K.Appathurai, Dr. S. Karthikeyan. A Survey on Spatiotemporal Access Methods.International Journal of Computer Appliations. Volume 18, No 4, 2011.

[15] Mohamed F. Mokbel, Xiaopeng Xiong, Moustafa A. Hammad, and Walid G. Aref, Continuous Query Processing of Spatio-temporal Data Streams in PLACE, 2004 Kluwer Academic Publishers. Printed in the Netherlands

[16] Su Chen · Beng Chin Ooi · Zhenjie Zhang, An Adaptive Updating Protocol for Reducing Moving Object Database Workload.

Author's Biography

K. Appathurai was born on 12th May 1974. He received his Master degree in Computer Applications from University of Bharathidasn in 1998. He completed his M.Phil from Manonmaniam Sundaranar University in 2003. He is working as an Asst. Professor and Head of the Department of Information Technology at Karpagam University, Coimbatore. Currently He is pursuing Ph.D. His fields of interest are Spatial Database.

Karthikeyan S. received the Ph.D. Degree in Computer Science and Engineering from Alagappa University, Karaikudi in 2008. He is working as a Professor and Director in School of Computer Science and Applications, Karpagam University, Coimbatore. At present he is in deputation and working as Assistant Professor in Information Technology, College of Applied Sciences, Sohar, Sulatanate of Oman. He has published more than 14 papers in Natrional/International Journals. His research interests include Cryptography and Network Security