TPR-Tree를 위한 이동 점의 묶음 갱신

황두동, 이응재, 이양구, 류근호
충북대학교 데이터베이스 연구실

Bulk Updating Moving Points for the TPR-tree

Hoang Do Thanh Tung, Eung-Jae Lee, Yang-Koo Lee, Keun-Ho Ryu
Database Laboratory, Chungbuk National University, Korea

Abstract

Assisted by high technologies of information and communication in storing and collecting moving object information, many applications have been developing technical methods to exploit databases of moving objects effectively and variously. Among them, today, Current and Anticipated Future Position Indexing methods manage current positions of moving objects in order to anticipate future positions of them or more complex future queries. They, however, strongly demand update performance as fast enough to guarantee certainty of queries as possible. In this paper, we propose a new indexing method derived from the TPR-tree that should has update performance considerably improved, we named it BUR-tree. In our method, index structure can be inserted, deleted, and updated with a number (or bulk) of objects simultaneously rather than one object at a time as in conventional methods. This method is intended to be applied to a traffic network in which vast number of objects, such as cars, pedestrians, moves continuously.

1. Introduction

Current technology allows systems to collect information of objects moving in traffic networks instantaneously and retain it in databases including object's positions and velocities, called moving object databases. As a result, there have been many algorithms or methods proposed and deployed, such as modeling, indexing, querying, etc, to meet the requirements for exploiting the vast data in moving object databases. One of the most concerned methods is an indexing method to organize data in order to speed up database access and facilitate query process. So far, indexing methods can be classified into two areas of research, the first area focuses on indexing methods managing past information of objects, the second focuses on indexing methods managing current and future information of objects. Our paper concentrates on dealing with the update problem of an indexing method for current and future information, the TPR-tree method [1].

The TPR-tree is an R*-tree [2] based indexing method. It uses current positions and velocities of moving objects as functions of time to build up its index structures. This method allows users to collect anticipated future queries about objects in space such as a traffic network. Therefore, speed of current-information updating performance is dispensable to ascertain queries reasonably and certainly because vast number of objects generates updates in a definite current-time period. In fact, while most methods update one object at a time into their structure, a real network system can be invoked for updates by a number of objects at a time unit. For example, in a big city like Seoul or others, the number of running cars at a time can reach at least 1 million, if we assume each car must generate an update during average every 10 minutes, the number of updates are possibly at least 1666.6 at any second. In case each car generates an update during average 1 hour (this appears impossible), the number of update are possibly at least about 300 at any second. On the other hand, if each car generates an update in every shorter time, the number of updates will be much greater at any second. Intuitively, if a method can update a bulk (a number) of updates into its structure once at a time, it must speed up update performance significantly whereby queries become more reasonably and more confidently. From this idea, we now propose a method based on TPR-tree that manages to not only update a bulk of objects into its structure but also optimize the index structure supporting query performance; we named it BUR-tree. This method exploit a compact, easy-to-maintain main memory internal node table that help clustering input data into suitable packets (bulk), and that help inserting the packets, including a number of objects, directly into leaf nodes simultaneously.

The remainder of the paper is organized as follows. Section 2 briefly describes some methods proposed to improve update performance for moving object indexing

methods, mostly for the TPR-tree. Section 3 presents the BUR-tree technique, discussing its structure, algorithms. Finally section 4 concludes the paper with our summary and future work.

2. Related Works

In this session, we discuss some improvement of TPR-tree based methods, some of which improved update performance. And we also discuss a recent update technique for R-tree based indexing methods, a bottom-up update approach.

In 2000, Saltenis et al. [1] proposed the TPR-tree, which adapts the R*-tree construction algorithms to moving objects. The main idea is to make the bounding rectangles as functions of time by current information of objects, namely positions and velocities so that the enclosed moving objects will always be kept in their rectangles over time. However, the author has not given any solution to update current information of objects in time except using bulk loading algorithm that is only appropriate to tree-building process.

In 2003, Yufei Tao [3] proposed TPR*-tree, which integrates novel insertion/deletion algorithms to enhance performance of the TPR-tree. TPR*-tree uses *swept regions*, formula (1)

$$Cost(q) = \sum\nolimits_{every\ node\ o} A_{SR}(o', q_T) \tag{1}$$

for insertion/deletion algorithm in place of the *integral metric*, formula (2)

$$\int_{now}^{now+H} A(t)dt, \tag{2}$$

(A is the area of a MBR). The update performance in his experiment outperformed that of the TPR-tree. The improvement for update performance is thanks to using *Choosepath* algorithm to find the best insertion path in insertion algorithm. Although Choosepath algorithm deducts a cost of some extra node accesses, it leads to a better tree structure so that delete algorithm can be better to find the node to delete. Consequently, total update performance seems better. Intuitively, this idea appears not reasonable.

In 2004, Bin Lin and Jianwen Su [4] proposed a new Bulk loading algorithm for TPR-tree in dealing with non-uniform datasets. Their paper presented a histogram-based bottom up algorithm (HBU) using histogram to refine tree structures for different distributions. Empirical studies showed that HBU outperforms the bulk loading algorithm of the TPR-tree for all kind of non-uniform datasets. Despite that, bulk loading algorithm is still only appropriate to the tree-building process.

Besides ideas improving the TPR-tree, in 2003 Mong Li Lee [5] proposed a Bottom update approach for R-tree [6] based indexing methods for moving objects. The strategy of this approach is that by using a hashing table storing object IDs and their page addresses; it gives direct accesses to leaf nodes instead of starting point at the root for updates. Moreover, to adjust a tree from bottom up to the root, it uses a compact main memory summary structure storing all internal nodes in order to find parent nodes. Although its experiments shows that the update performance is much better than that of R-tree, this approach only considers changes of object's positions instead of their velocities. This method assumed that position distance between two adjacent updates is not too far. This seems not suitable to methods like the TPR-tree, because the TPR-tree considers change of velocities so that object's positions between two updates can be in long distance.

Inspired from the update problem for moving object methods and the interesting idea of the bottom update approach, we had our proposal, BUR-tree.

3. BUR-tree

In our method, we are supported that the number of objects generating update at a time t is greater than one and this number are much less than total number of objects in a tree. Moreover, the tree structure must be built up by a bulk loading algorithm at once, so this structure is first considered nearly optimal.

To facilitate bulk insertion, the index structure platform for our method as following

- The index structure is kept intact the same as that of the TPR-tree.
- An *internal node table*, seeing Figure [1], in main memory is used for storing all internal nodes of the tree. Different from the summary structure table of bottom up update approach [5], with our table, we can travel from root down to any internal node without a disk access.
- A *hashing table*, seeing Figure [1], is to store all object IDs and their page addresses. Practically, the hashing table storing antecedents of objects is dispensable to any index structure to locate object in a structure for deletion process. However, instead of storing object's MBR as to spatial methods, we store page addresses of objects.

As seeing the *internal node table* in Figure [1], besides *level*, *MBR* of each internal node, and *Nptr* which is a pointer to the physical node, we use *childpos* pointers, memory addresses for a node's childs , and *par* pointers, memory addresses for a node's parent, in order to easily trace Childs and a Parent of a node in table and yet we use *num* to remember the number of child pointers of a node and help utilizing deleted entry space (we do not discuss this parameter in this paper). In fact, the size of each entry in the table is a small fraction of the size of the corresponding TPR-tree node. This savings is achieved because the individual MBRs of the child nodes in the TPR-tree are excluded in the table as discussion in [5].

The maintenance cost for the main memory internal node table is not relatively inexpensive. The reasons are that first we avoid leaving a node to underflow by forcing deletion operations to run after insertion operations (discussed in the below paragraph), and secondly if an internal node is split, a new entry will simply be added into the end of the internal node table. Furthermore, in our insertion algorithm, we always try to avoid enlarging the directory bounding rectangle of a node except MBRs of its Childs, so cost of propagating tree adjustment is minimized.

Normally, update process is separated into two periods, the first period executing operations to delete antecedents of

inserted objects, and the second executing operations to insert new objects. This process usually requires a cost of adjusting tree because some node can underflow or overflow, especially when a large number of objects need to be updated at a time. Therefore, to keep tree structure as optimized as it first built up, we expect to avoid influencing on this structure as much as possible. Our strategy is that to update the number of objects once at a time and reduce adjustment cost, we cluster updated (inserted) data into bulks of data, each of which will be inserted directly into leaf nodes. After that, we either delete antecedents of objects while inserting objects into leaf nodes or do not delete until after insertion process finishes. In short, insertion algorithm is the main plank of our improvement.

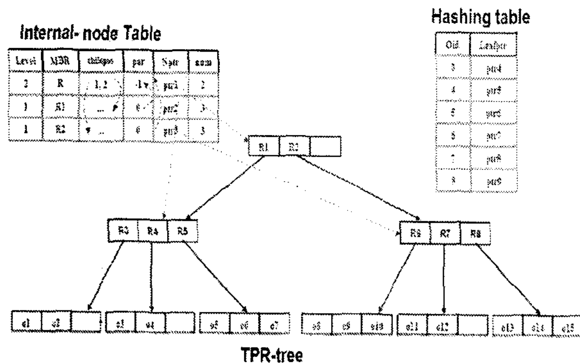

Figure 1.Summary Structure for BUR-tree

3.1 I nsertion Algorithm

In this session, we describe steps of the bulk insertion algorithm. We assume a number of objects need to be updated and they are interspersed over space. Steps of the insertion algorithm as following

Step 1: Clustering inserted data is by using internal node table. We travel the tree in top-down manner from the root node in main memory by using the internal node table as seeing the Figure 2, there are 5 black pointers interspersed in the space. At each node, we count on *clustering rules* to cluster data into finer clusters (in fact, they are MBRs of tree's nodes) and decide which node's child is following. Moreover, we use a queue to store objects lying outside cluster areas as *unclustered data.* As seeing Figure 2, two points belong to the cluster R1, only one belongs to the cluster R2, and the other black points will be kept in the queue. This step will stop at node's level adjacently above leaf node's level. At the end of step 1, if a node at this level contains some inserted objects, it will be a final cluster. In short, all total inserted objects are separated into two kinds of data, clustered data and unclustered data.

Step 2: Before clustered data continue to be inserted into the leaf nodes that are now Childs of node clusters, we look for the antecedents of the inserted objects in any those leaf nodes by using the hashing table for direct access to leaf nodes. If a leaf node contains any antecedents, we delete them and adjust their MBR. Finally, we continue to insert each clustered data (a bulk) into leaf nodes under guidance of clustering rules, seeing Figure 2, three points found the suitable leaf node to insert.

Step 3: We propagate adjustment from bottom of the tree.

Step 4: We delete all antecedents of objects of unclustered data (queue) from the tree. Finally, we insert all objects of unclustered data into the tree by original insertion algorithm of the TPR tree.

Using this insertion algorithm, we expect that update performance is much improved and the structure after updated will be in a few overlaps.

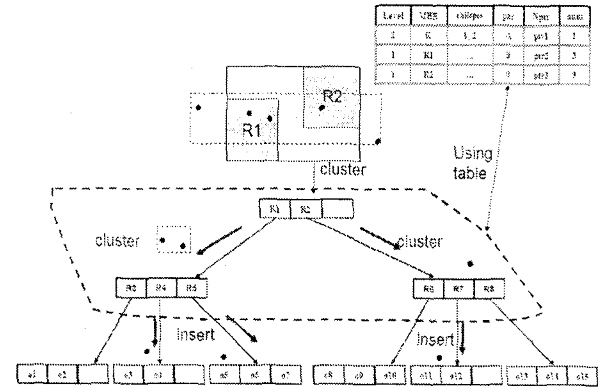

Figure 2. Inserting 5 black points into the tree by clustering input data into bulks (fine clusters)

3.2 Clu stering rules

Clustering rules supports three possibilities to cluster data when input data (or a raw cluster) reaches a node in the tree. As seeing the Figure 3, A, B, and C are MBRs of child's nodes on a current node. Our work here is to check any node clipping input data (clipping means ability to enclose some objects of a MBR), and cluster input data into finer clusters.

- In case of (a) **fully clipping**, if MBRs of A and B is clipping input data fully, we can immediately gain two new clusters (left and right) in A and B, after that we simply continue to check A's Childs with the left cluster as a new input data. Similarly so do we for B's Childs.
- In case of (b) **partially clipping**, A and B are only clipping the input data partially. Visually, however, we can enlarge B to fully clip the input data such that the expanded area will not overlap any other MBRs. Therefore, we will enlarge B and gain two new clusters. After that we continue to check A and B's Childs. Remember that B has new logical size regardless of its Childs.
- Finally, in case of (c) **dead lock**, A and B are clipping the input data only partially, furthermore none of them can be enlarged to fully clip the input data. Hence we temporally ignore two remaining objects and input them into unclustered data (the queue). Then we continue to check A and B's Childs with the two new clusters.

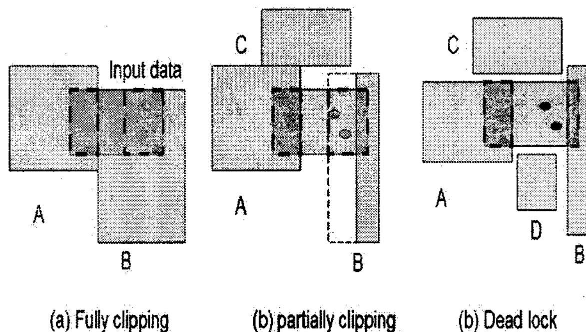

Figure 3.Clustering rules for clustering data: clipping input data fully (a), clipping partially (b), dead lock (c)

3.3 Delet ion Algorithm

Although we hope to delete almost all antecedents of the inserted objects during insertion process, we describe a deletion algorithm to delete objects remaining in unclustered data. A simple algorithm is that using hash table is to obtain leaf node's addresses of all unclustered objects. We collect objects of the same node, delete them together, and adjust their MBR. After that, we repeat this process with parent nodes in bottom up manner. In short, this algorithm operates recursively until reaching root node. In this algorithm, instead of using a Findparent algorithm, we simply use pointers *par* of the table to node's parents in internal node table.

4. Conclusion

Motivated by traffic networks, which has large volume of updates from moving cars, and urgent matters of speeding up update performance for current and future position indexing methods like the TPR-tree, this paper proposed the method BUR-tree giving bulk update strategy for the TPR-tree. Nevertheless, this update strategy can easily be applied to the members of the family of TPR-tree (or R-tree) based indexing methods. Our strategy hopefully improves not only much update performance but also the index structure. In the future, we are going to accomplish our implementation of our method and complement it.

Acknowledgment

This work was supported by MOST and KOSEF RRC (ICRC of Cheongju University) in Korea.

Reference

[1] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," In Proc. of the 2000 ACM SIGMOD Int'l Conf. on Management of Data, Dallas, Texas, USA, May 2000.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Se eger, "The R*-tree: An Efficient and Robust Access Method For Points and Rectangles," In Proc. of the 1990 ACM SIGMOD Int'l Conf. on Management of Data, Atlantic City, New Jersey, USA, May 1990.

[3] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," In Proc. of the 29th Int'l Conf. on VLDB, Berlin, Germany, September 2003.

[4] Bin Lin, and Jianwen Su, "On Bulk Loading TPR-tree," In Proc. of the 2004 IEEE Intl Conf. on Mobile Data Management (MDM'04), Berkeley, California, January 2004.

[5] Lee, M. L., W. Hsu, C. S. Jensen, B. Cui, "Supporting Frequent Updates in R-Trees: A Bottom-Up Approac", DB Technical Report TR-6, April 2004.

[6] Guttman. "R-Trees: A Dynamic Index Structure For Sp atial Searching". In Proc. ACMSIGMOD International Conference on Management of Data, pages 47–57, Bos ton, June 1984.