Predicted Range Aggregate Processing in Spatio-temporal Databases

Wei Liao, Guifen Tang, Ning Jing, Zhinong Zhong

School of Electronic Science and Engineering, National University of Defense Technology
Changsha, China
liaoweinudt@yahoo.com.cn

Abstract

Predicted range aggregate (PRA) query is an important researching issue in spatio-temporal databases. Recent studies have developed two major classes of PRA query methods: (1) accurate approaches, which search the common moving objects indexes to obtain an accurate result; and (2) estimate methods, which utilize approximate techniques to estimate the result with an acceptable error.

In this paper, we present a novel accurate prediction index technique, named PRA-tree, for range aggregation of moving objects. PRA-tree takes into account both the velocity and space distribution of moving objects. First, the velocity domain is partitioned into different velocity buckets, and moving objects are classified into different velocity buckets by their velocities, thus objects in one bucket have similar velocities. Then we use aTPR-tree, which is based on the basic TPR-tree structure and added with aggregate information in intermediate nodes, to index objects in each bucket. PRA-tree is supplemented by a hash index on IDs of moving objects, and exploits bottom-up deletion algorithm, thus having a good dynamic performance and concurrency. Also new PRA query methods with a more precise branch-and-bound searching strategy are developed for PRA-tree. Extensive experiments confirm that the proposed methods are efficient and practical.

1. Introduction

Traditional research in spatio-temporal databases often aims at predicted query, which retrieves the moving objects lying inside a multidimensional hyper-rectangle in future. In many scenarios (e.g., statistical analysis, traffic monitoring, etc.), however, users are interested only in summarized information about such objects, instead of their individual properties. Consider, for example, a spatio-temporal database managing the vehicles in a city, results of predicted queries (e.g., finding all vehicles in the center) are meaningless (due to continuous object movements), while the aggregate information (the number of vehicles) is usually stable and measurable (e.g., finding the number of vehicles across the center during the next 5 minutes) [1], [2].

Specifically, given a set S of moving points in the d-dimensional space, a predicted range aggregate (PRA) query returns a single value that summarizes the set $R \subseteq S$ of points in a d-dimensional hyper-rectangle q_R and a future time interval (or a future timestamp) q_T according to some aggregation function (e.g., *count*, *max*, *min*, *sum*, *average*). In this paper, we consider predicted range *distinct count* queries on multidimensional moving points, where the result is the size of R (e.g., the number of vehicles in an area q_R and a future time interval (or a future timestamp) q_T), but the solutions can apply to any other aggregation mentioned above with straightforward adaptation.

1.1 Motivation

A PRA query can be trivially processed as an ordinary query, i.e., by first retrieving the qualifying objects and then aggregating their properties. This approach assumes that the server manages the detailed information of moving objects using a (typically disk-based) spatio-temporal access method [3], [4], [5], however, incurs significant overhead since, the objective is to retrieve only a single value (as opposed to every qualifying object). The most popular aggregate indexes [6], [7] aim at estimating the PRA results. These approaches are motivated by the fact that approximate aggregates with small error are often as useful as the exact ones in practice, but can be obtained much more efficiently. In particular, such estimation methods require only a fraction of the dataset or a small amount of statistics and are significantly faster than exact retrieval. In this paper, we focus on accurate PRA queries processing with a novel aggregate index.

1.2 Contributions

In this paper, we propose a new indexing method, called predicted range aggregate tree (PRA-tree), for efficient

PRA queries processing. The PRA-tree takes into account the distribution of moving objects both in velocity and space domain. First, the velocity domain is partitioned into velocity buckets, and moving objects are classified into different velocity buckets by their velocities, thus objects in one bucket have similar velocities. Then we use aTPR-tree, which is based on the basic TPR-tree structure and added with aggregate information in intermediate nodes, to index the objects in each bucket. We supplement the PRA-tree by a hash index constructed on the IDs of moving objects, and develop a bottom-up deletion algorithm to obtain good dynamic performance and concurrency. Finally, we present novel algorithms that use a more precisely branch-and-bound searching strategy for PRA queries based on PRA-tree.

The rest of the paper is organized as follows: Section 2 reviews previous work related to ours. Section 3 provides the problem definition and an overview of the proposed methods. Section 4 presents the PRA-tree index structure and its construction, insertion, deletion and update techniques. Section 5 elaborates algorithms for PRA queries. Section 6 evaluates the proposed methods through extensive experimental evaluation. Finally, Section 7 concludes the paper.

2. Related Work

Section 2.1 discusses the most popular TPR-tree index, while Section 2.2 overviews the aggregate R-tree (aR-tree) which motivates our solution.

2.1 The TPR-tree
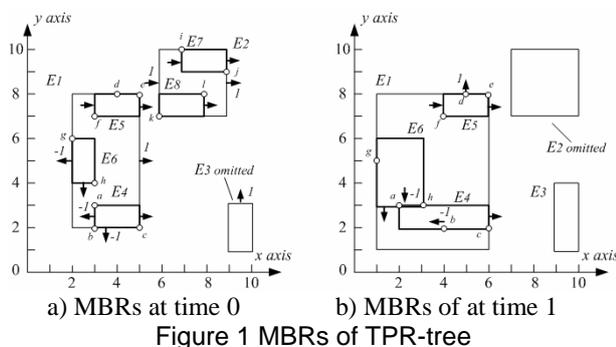

a) MBRs at time 0   b) MBRs of at time 1
Figure 1 MBRs of TPR-tree

The TPR-tree is an extension of the R-tree that can answer predicted queries on dynamic objects. The index structure is very similar to R-tree, and the difference is that the index stores velocities of elements along with their MBRs in nodes. The leaf node entry contains not only the position of moving objects but also their velocities. Similarly, an intermediate node entry also stores MBRs and velocity vector VBRs of its child nodes. As in traditional R-tree, the extents of MBR are such that tightly encloses all entries in the node at construction time. The velocity vector of the intermediate node MBR is determined as follows: (i) the velocity of the upper edge is the maximum of all velocities on this dimension in the

sub-tree; (ii) the velocity of the lower edge is the minimum of all velocities on this dimension. This ensures that the MBR always encloses the underlying objects, but it is not necessarily tight all the time. The TPR-tree inherits the problems related to the R-tree, such as overlap and dead space. Since the index structure is dynamic, its query performance degrades quickly with time. The algorithms for insertion and deletion are also similar to R-tree, while the TPR-tree uses time-parameterized metrics for parameters such as the area, perimeter, and distance from the centroid [4]. For example, Figure 1a) shows the MBRs and their velocities of TPR-tree at construction time 0, and Figure 1b) shows MBRs of the same TPR-tree at future time 1.

2.2 The Aggregate R-tree (aR-tree)

The aR-tree [8] improves the conventional R-tree by adding aggregate information into intermediate nodes. Figure 2a) shows an example, where for each intermediate entry, in addition to the minimum bounding rectangle (MBR), the tree stores the number of objects in its subtree (i.e., count aggregate function). To answer a RA query q (the shaded rectangle), the root R is first retrieved and its entries are compared with q. For every entry, there are three cases: 1) The MBR of the entry (e.g., e1) does not intersect q, and thus its subtree is not explored further. 2) The entry partially intersects q (e.g., e2) and its child nodes are fetched to continue the search. 3) The entry is contained in q (e.g., e3), then we simply add the aggregate number of the entry (i.e., 3 for e3) without accessing its subtree. As a result, only two node accesses (R and R2) are necessary, while a conventional R-tree (i.e., without the aggregate numbers) would also visit R3. The cost savings increase with the window of the query, which is an important fact because in practice RA queries often involve large rectangles.
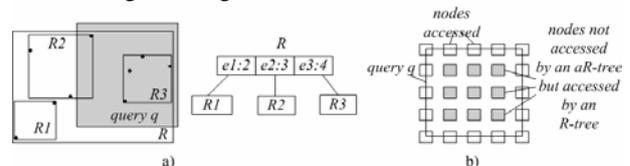

Figure 2 The aR-tree

3 Preliminary and Overview

We start with a concrete definition of aggregate query in spatio-temporal databases before presenting the PRA-tree index. By definition, a spatiotemporal object is a unified object with spatial and temporal extent [2]. A pure spatial object can be a point, a line, or a region in two or three-dimensional space. The position and/or shape either changes continuously (e.g., the motion of vehicles) or discretely (e.g., the shrink of forest) with time. In this paper we concentrate on the points (moving objects), which have continuously moving positions, in two-dimensional space. In spatio-temporal databases, a

moving object is represented as *<Loc, Vec, A_1, A_2...A_n>*, where *Loc*, *Vec* denote the position and velocity vector respectively, and A_1, A_2...A_n denote the non-spatial properties of this object.

An aggregate function takes a set of tuples and returns a single value that summarizes the information contained in the set of tuples. Spatio-temporal aggregate query first retrieves the qualifying objects that satisfy a spatio-temporal query predicate and returns aggregation information (e.g., count, average) on the non-spatial properties of objects.

Given a range query $q(q_R,q_T)$, which obtains the objects lying in the query window q_T and space area q_R, and a moving objects dataset P, a spatio-temporal query predicate $Sel(P)$ can be expressed as $Sel(P)=\{p_i|\exists t \in q_T \ such \ that \ p_i(t) \in q_R \}$. Especially, if $q_T=t_f$, where t_f denotes a future timestamp, then the query q is a predicted timestamp range query. Generally, the query space area is a static rectangle. Therefore, we can define predicted range aggregate query as the following.

Definition 1 (predicted range aggregate query). Given a dataset *P*, in which each tuple is represented as *<Loc, Vec, A_1, A_2,...,A_n>*, where the domain of A_i is D_i, and a aggregate function $f : D_1 \times D_2 \times ... \times D_n \rightarrow D_{agg}$, where D_{agg} denote the range of *f*. then an predicted range aggregate query can be defined as follows:

$$f(q,P)=f(Sel(P))=f(\{p_i| \exists t \in q_T \ such \ that \ p_i(t) \in q_R \}).$$

Similar to the aR-tree, we enhance the conventional TPR-tree by keeping aggregate information in intermediate nodes. To answer a PRA query $q(q_R,q_T)$, the root node is first retrieved and its entries are compared with q. For every entry, there are three cases: 1) the MBR of this entry does not intersect query area q_R all through the query time q_T, then this subtree is not visited further; 2) the entry is totally contained in the query area q_R during q_T, and we simply add the aggregate information without accessing its subtree; 3) the entry partially intersects the area q_R during q_T, then its child nodes need to be fetched and continue searching until the leaf. Using this approach, the cost of processing PRA queries can be significantly reduced.

However, the TPR-tee is constructed merely in the space domain, and slightly considering the velocity distribution of moving objects. At construction time, TPR-tree index clusters objects mainly according to their spatial proximity into different nodes; however, the velocities of objects in the same page are always discrepant greatly. The minority of objects with higher velocity make the velocity-bounding rectangle (VBR) relatively larger. In addition, the MBR will increase extremely with time, causing deterioration of the query performance and dynamic maintenance. Actually, in most applications PRA queries often visit unnecessary intermediate TPR-tree nodes due to the extremely large MBRs and massive dead space.

Figure 3 shows the moving objects in two dimensional space with velocity vector (10, 0) and (-10, 0) for example. Figure 3 a) illustrates the MBRs of TPR-tree constructed merely in space domain. Obviously, the MBRs of intermediate TPR-tree nodes extend extremely along the horizontal direction, thus incurring the degradation of query performance. Therefore, an effective aggregate query indexing technique needs to consider the distribution of moving objects in both velocity and space domain to avoid the deterioration of query and dynamic performance.

Motivated by this, we propose a predictive range aggregate tree (PRA-tree) index structure. First, the velocity domain is partitioned into buckets with about the same objects number. Then moving objects are classified into different velocity buckets by their velocities, and objects with similar velocities are clustered into one bucket. Finally, an aTPR-tree structure is presented for indexing moving objects in each bucket.

Figure 3 b) and c) shows the MBRs of PRA-tree that considers the distribution of velocity domain. The moving objects are classified into two velocity buckets. The bucket1 (as shown in figure 3 b)) contains the moving objects with velocity vector (10, 0), while bucket2 (as shown in figure 3 c)) contains the moving objects with velocity vector (-10, 0). As seen, the MBR of each bucket moves with time, but the shapes of MBR keep unchanged. So PRA-tree can hold a good query performance during all the future time.

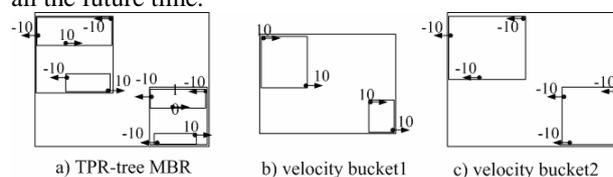a) TPR-tree MBR   b) velocity bucket1   c) velocity bucket2

Figure 3 MBRs of PRA-tree

In addition, existing TPR-tree update algorithms work in a top-down manner. For each update, one index traversal to locate the item for deletion and another traversal to insert a new item are needed. The deletion operation affects the disk I/O cost greatly, for the MBRs of TPR-tree become looser with time, thus incurring lots of area overlaps between MBRs, and the searching operation for deletion needs to visit all the TPR-tree nodes at worst case. The top-down approach for visiting the hierarchical structure is very simple, and easy for dynamic maintenance. Nevertheless, for TPR-tree like index structures, the area between intermediate node MBRs inevitably overlaps each other (which is not occurred in other traditional indexes such B-tree), so that the top-down searching strategy is inefficient in nature. This manner results in the deterioration of TPR-tree, and is not suitable in frequent update applications. Motivated by this, we exploit the bottom-up delete strategy to improve the dynamic performance of PRA-tree.
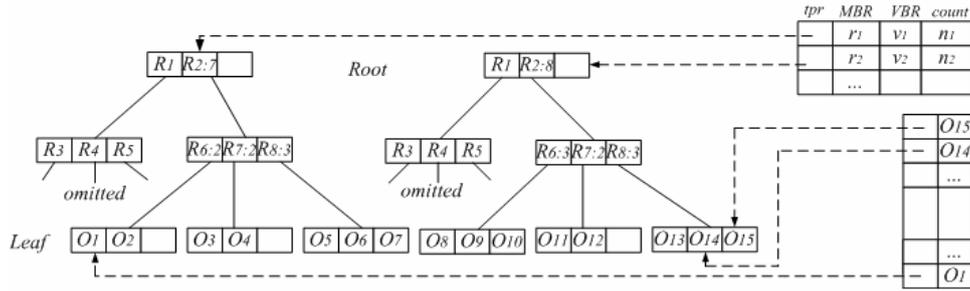
Figure 4 The structure of PRA-tree

4. The PRA-tree

Section 4.1 discusses the structure of PRA-tree, while Section 4.2 provides the construction method, Section 4.3 gives the insertion, deletion and update techniques, Section 4.4 evaluates the effect factors of PRA-tree performance.

4.1 The PRA-tree Structure

Specifically, in PRA-tree structure we keep the basic TPR-tree index structure and add aggregate summary into its intermediate nodes to make a new index, the aggregate TPR-tree (aTPR-tree). The entries in aTPR-tree are organized as vector *<MBR, VBR, Agg, ptr>*, where *MBR*, *VBR*, *Agg*, *ptr* denote the space area, velocity range, aggregate information of this node and pointer to its subtree respectively. In addition, we introduce a main memory linear queue for management of velocity buckets, the items of which are formed as *<MBR, VBR, Agg, tpr>*, where *MBR, VBR, Agg, tpr* denote the space area, velocity range, aggregate information of this bucket and pointer to its corresponding aTPR-tree respectively. To improve the dynamic performance of PRA-tree, we supplement a disk-based hash index structure to access the leaf node of PRA-tree directly. Further, we modified the original TPR-tree node item as vector *<entry, ···, entry, parentptr>*, where *entry* denotes the child nodes contained in this node, and *parentptr* denotes the physical address of its parent node. Compared with node page size, the space consumption by pointer *parentptr* is trivial, and its effect on the fanout of PRA-tree is ignorable.

Figure 4 illustrates the PRA-tree structure. The top right corner is a velocity bucket queue, in which each item describes the MBR, VBR and aggregate information (i.e., count) of its corresponding aTPR-tree. The bottom right corner is a hash index constructed on IDs of moving objects, the item is defined as vector *<oid, ptr>*, where *oid* denotes the identifier of moving objects, and *ptr* denotes physical offset of the object entry in leaf node; the left of Figure 4 is the aTPR-tree structures pointed to by the velocity bucket queue. In this figure, the pointer to parent node is not clearly depicted for concision.

4.2 Construction

We use spatio-temporal histograms as mentioned in [9] on two velocity dimensions to compute the velocity buckets number and their velocity range. The main idea is to divide the moving objects set into different partitions with about the same objects number. Then the algorithm scans the set of moving objects in sequence, and inserts objects into the aTPR-tree pointed to by corresponding bucket according to their velocities. To avoid redundant disk I/Os caused by frequent insertion one by one, construction algorithm exploits the bulk loading technique [9] to construct the aTPR-tree. However, unlike the traditional bulk loading method, the construction algorithm must compute and store the aggregate information in the intermediate nodes.

The foremost problem for PRA-tree is to choose the number of velocity buckets. With too few velocity buckets the PRA-tree can not gain the optimal query and dynamic maintenance performance, while too many velocity buckets may cause shifts between velocity buckets when update, thus resulting in the deterioration of index structure. So choosing a proper number of velocity buckets can make the query and dynamic performance of PRA-tree optimal. In the experimental section, we evaluate the effect of velocity buckets number on the query and dynamic maintenance performance.

4.3 Insertion , Deletion and Update

The insertion algorithm is straightforward. When inserting a new moving object into the PRA-tree index, the algorithm first scans the main-memory velocity bucket queue to find the bucket that contains this object, and the aTPR-tree related to this bucket can be obtained by the pointer in the queue item. Then a new object entry is produced and inserted into a suitable leaf node using the standard TPR*-tree insertion algorithm. Finally, a new item is produced and inserted into the hash index. If overflow occurs while insertion, the aTPR-tree node must be split with the method mentioned in [4]. In addition, the algorithm must modify the aggregate information of this bucket and aTPR-tree nodes along the searching path.

To delete an object entry from the PRA-tree, a simple straight approach is to first find the velocity bucket that

contains this object and delete the object entry from the aTPR-tree with standard deletion algorithm. However, usually the searching path for deletion may be several, so the algorithm must keep the searching path to perform another operation to complete the modification of aggregate information along the path, thus causing redundant disk I/Os. Motivated by this, we exploit the bottom-up deletion strategy like [10] to improve the deletion performance. The deletion algorithm first locates the leaf node that holds the object entry with hash index, and then deletes the entry from this leaf node directly. Then the algorithm ascends the branches of PRA-tree by the pointer *parentptr* until the root node, and modifies the MBR, VBR and aggregate information of intermediate nodes along the path meanwhile. Finally, the corresponding object item in the hash index must be deleted to reflect the change and also the bucket item related to this aTPR-tree must be modified.

The update algorithm uses the standard deletion-insertion mechanism in TPR-tree. The algorithm first deletes the old entry from the PRA-tree and then inserts a new entry into the PRA-tree.

4.4 Effects on Index Performance

PRA-tree partitions moving objects into velocity buckets that do not overlap each other in velocity domain. So the index can perform a good concurrency when a large amount of concurrent dynamic operations occur. Obviously, the concurrency improves with the number of velocity buckets. However, with too large a bucket number the index structure is prone to instability. That is to say, because the velocity range of each bucket is too small, when an object is updated, the likely object's shift from one bucket to another bucket may cause excessive disk I/Os. In addition, limited by the system memory, and considering the space and velocity distribution uncertainty of moving objects, too many velocity buckets do not reduce node accesses for processing PRA query. Therefore, PRA-tree with a proper number of velocity buckets can obtain the optimal query and update performance.

The cost of maintain the velocity bucket queue is inexpensive. The queue is constructed along with PRA-tree, when dynamic operations such as insertion and deletion occurred, the summary in which must also be updated. The queue is pinned in main memory, thus decrease the visiting cost greatly. And the space consumed by velocity queue is rather small and ignorable.

5 The Predicted Range Aggregate Query Algorithm

We now illustrate the algorithm for answering a predicted range aggregate query with PRA-tree. There are two types of queries: predicted time-slice range aggregate query and predicted window range aggregate query, which have

been thoroughly surveyed in [1]. For example, we consider the following queries: i) "how many cars are expected to appear at the center of city 10 minutes from now?" and ii) "how many cars are expected to cross the center of city during the next 5 minutes?" Section 5.1 presents the algorithm for queries as case 1, and section 5.2 details the algorithm for queries as case2.

5.1 Predicted Time-slice Aggregate Range Query (PTRA Query)

PTRA query returns the aggregate information of objects that will fall in a rectangle at a future timestamp based on the current motion of moving objects. Obviously, according to the current MBR and VBR of each node in PRA-tree, we can get the MBR of this node at future timestamp, and then an aggregate range search is implemented under the PRA-tree at this timestamp.

Specifically, given a PTRA query $q(q_R, q_T)$, the algorithm first scans the velocity bucket queue, and for each bucket we can get whether any object in this bucket lies in the query area q_R at timestamp q_T according to the bucket's MBR and VBR. If q_R does not intersect the space area covered by a velocity bucket at future timestamp q_T, the aTPR-tree pointed by this bucket need not to be visited. Or else if q_R contains the space area covered by a velocity bucket at future timestamp q_T, the aggregate information is returned from the bucket item directly and added to the query result. Otherwise, if q_R intersects the space area covered by a velocity bucket at q_T, the corresponding aTPR-tree is obtained and then from the root point, the algorithm recursively computes the topological relation between the MBR of this node at future timestamp q_T and query area q_R. If the MBR is contained in q_R, then the algorithm adds the aggregate information in this node to the query result, or else if the MBR does not intersect q_R each other, then the node is skipped; otherwise this subtree is explored further until the leaf level.

5.2 Predicted Window Aggregate Range Query (PWRA Query)

PWRA query returns the aggregate information of objects that will fall in a query rectangle at a future time window based on current motion of moving objects. To answer a PWRA query, a straightforward method is to judge whether the MBR of current node is totally contained in the query area at some future time. If true, then the algorithm only adds the aggregate information without visiting this subtree; or else if the MBR of this node does not intersect the query area at any future time, then this node is skipped. Otherwise, this subtree is explored further. This approach will access nodes whose MBR partially intersect the query area while the moving objects enclosed in totally lie in the query area at different timestamp (as shown in figure 5),thus causing excessive disk I/Os.

Motivated by this, we present an enhanced predictive range aggregate query (EPRA) algorithm with a more precise branch and bound criterion. Before introducing the EPRA algorithm, we give the following lemmas.

Lemma 1 Given a moving rectangle R with fixed edge velocities and a static query rectangle q, let $q \cap R \neq \Phi$. If the four vertices of R lie in the query area q at different future timestamps, then the moving points enclosed in R will lie in the query area q during the future time.

Proof: As an illustration of Lemma 1, consider Figure 5 where the relationship between $R(a, b, c, d)$ and query q is shown as case a) and case b), since $q \cap R \neq \Phi$. Supposing the VBR of R is $< v_x^-, v_x^+, v_y^-, v_y^+ >$, then the direction of VBR can only be depicted as in Figure 5 a) and b), for vertices a, b, c and d will lie in q at future time.

In case a), c has the velocity $< v_x^-, v_y^+ >$, for $\forall p \in R$ point p has the velocity $< v_x, v_y >$, where $v_x > v_x^-$ and $v_y < v_y^+$. If c lies in q at future timestamp t_c, then c must cross the line l_2 and not cross the line l_3 at t_c. So that for point p, it must have crossed line l_2 before t_c and still not cross line l_3, the p lies in q at some timestamp before t_c.

In case b): for a, b and c will lie in q at future times, obviously at least either b or c must lies in q at some future timestamp t_s with d, then at t_s the topological relationship between R and q can be illustrated as Figure 5 a). According to the discussion above, we can conclude that every point enclosed in R will lie in q at some future timestamp.
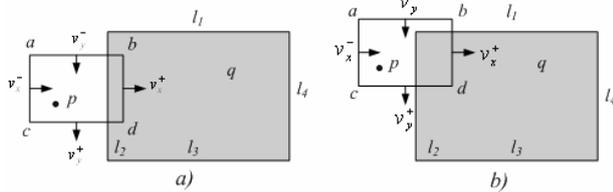

Figure 5 Topological relations between R and q

Lemma 2 Given a moving rectangle R with fixed edge velocities and a static query rectangle q, let $q \cap R = \Phi$. If the four vertices of R lie in the query area q at different future timestamps, then the moving points enclosed in R will lie in the query area q during the future time.

Proof: Supposing vertex d to be the first point that will lie in the query q, then at this timestamp the topological relationship between R and q can be shown as in Figure 5. So according to Lemma 1, we can deduce Lemma 2.

Heuristic 1: Given an intermediate entry E and a PRA query $q(q_R, q_T)$, the subtree of E totally satisfy query q if the vertices of MBR in entry E lie in query area q_R during future time q_T. If true, the query algorithm only returns the aggregate information of E without accessing its subtree.

Heuristic 1 reduces the searching cost considerably, while incurring rather small computational overhead. Specifically, the algorithm first scans the velocity bucket queue, according to the current MBR and VBR of this bucket, the future timestamps when each vertex lies in the query area q_R can computed. If all the vertices can lie in q_R during the query window q_T, then the algorithm only need to return the aggregate information of this bucket; or else if none of the vertices will lie in q_R during the query window q_T, then the algorithm skips this bucket; otherwise the bucket is explored further. Similarly, when searching the aTPR-tree, from the root point the timestamps when four vertices of MBR in each node lie in q_R is computed. If all the vertices can lie in q_R during the query window q_T, then the algorithm only need to return the aggregate information of this node; or else if none of the vertices will lie in q_R during the query window q_T, then the algorithm skips this node; otherwise this subtree is explored further.

The Algorithm 1 describes the pseudo-code for processing PWRA query as follows:

Algorithm 1
Input: a PWRA query $q(q_R, q_T)$, output: $Agg(q)$
1. Initialize $Agg(q)$, set $Agg(q) \leftarrow 0$
2.   For each item E in the velocity bucket queue
3.   Compute the timestamps when each vertex lies in q_R;
4.   If all the timestamps lie between q_T
5.   Then $Agg(q) \leftarrow Agg(q) + E.Agg$;
6.   Else if none of the timestamps lies between q_T
7.   Then skip E;
8.   Else get the aTPR-tree pointed by this bucket item E;
9.   from the root point, for each entry E in this node
10.   Compute the timestamps when each vertex lies in q_R;
11.   If all the timestamps lie between q_T
12.   Then $Agg(q) \leftarrow Agg(q) + E.Agg$
13.   Else if none of the timestamps lies between q_T
14.   then skip E
15.   Else explore the subtree recursively until leaf level
16.   Compute the aggregate information in leaf node E
17.   Set $Agg(q) \leftarrow Agg(q) + E.Agg$
18.   End if
19.   End for
20.   End if
21. End for
End algorithm 1

6 Experimental Results and Performance Analysis

6.1 Experimental Setting and Details

In this section, we evaluate the query and update performance of PRA-tree with aTPR-tree and TPR*-tree. We use the Network-based Generator of Moving Objects [11] to generate 100k moving objects. The input to the generator is the road map of Oldenburg (a city in Germany). An object appears on a network node, and randomly chooses a destination. When the object reaches its destination, an update is reported by randomly selecting the next destination. When normalize the data space to 10000×10000, the default velocity of objects is

equal to 20 per timestamp. At each timestamp about 0.8 % moving objects update their velocities. The PRA queries are generated as follows: i) the query spatial extent *qRlen* is set as $100\times100, 400\times400, 800\times800, 1200\times1200, 1600\times1600$ respectively, and the starting point of its extent randomly distributes in $(10000-qRlen)\times(10000-qRlen)$.ii) the query time window is [*Tisu，Tisu+qTlen*] (*Tisu* is the time when the query is presented), where *qTlen*=20,40,60,80,100 respectively. The query performance is measured as the average number of node accesses in processing ten PRA queries with the same parameters. The update performance is measured as the average node accesses in executing 100 moving objects updates.

Table 1 summarizes the parameters of PRA-tree exploited for the workload. For all simulations, we use a Celeron 2.4GHz CPU with 256MByte memory.

Table 1   PRA-tree parameters

Parameter	Value	Description
Number of buckets	25/50/100/1 50/200/250	The velocity domain is split into 5×5, 5×10, 10×10, 10×15, and 10×25 respectively.
Page size	1k	The page size of PRA-tree.
Fanout	21	The average fanout of PRA-tree in intermediate node.
Average height	3	The average height of PRA-tree.

6.2 Performance Analysis

We compare the query and update performance of PRA-tree, TPR*-tree and aTPR-tree by node accesses. In order to study the deterioration of the indexes with time, we measure the performance of PRA-tree, aTPR-tree and TPR*-tree, using the same query workload, after every 5k updates.
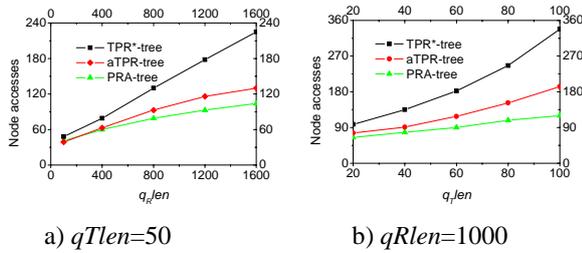

a) *qTlen*=50   b) *qRlen*=1000
Figure 6 comparison of PTRA query performance

Figure 6 a) and b) shows the PTRA query cost as a function of *qRlen* and *qTlen* respectively. In Figure 6 a) we fix the parameter *qTlen*=50 and in Figure 6 b) we fix the parameter *qRlen*=1000. As seen, the query performance of PRA-tree works best and TPR*-tree exhibits a worst performance. This is because PRA-tree is constructed both on the space and velocity domain, the area of MBRs overlap relatively less than those of aTPR-tree and TPR*-tree, thus having a good query performance. While TPR*-tree and aTPR-tree may visit many unnecessary nodes for the massive overlaps between area of MBRs with time, causing a worse query

performance. In addition, the larger the query range area, the better the PTRA query performance of PRA tree, for the intermediate nodes in PRA-tree store the aggregate information of this subtree, thus reducing the node accesses needed by TPR*-tree. The aTPR-tree holds a relative worse performance than PRA-tree for its larger MBRs with time.
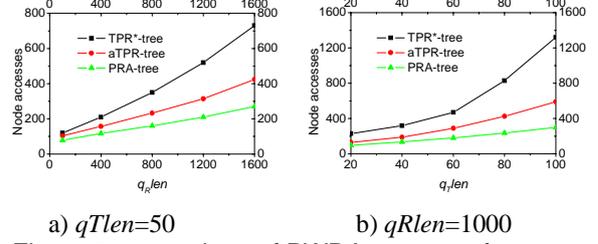

a) *qTlen*=50   b) *qRlen*=1000
Figure 7 comparison of PWRA query performance

Figure 7 a) and b) shows the PWRA query cost as a function of *qRlen* and *qTlen* respectively. In Figure 7 a) we fix the parameter *qTlen*=50 and in Figure 7 b) we fix the parameter *qRlen*=1000. As seen, the query performance of PRA-tree works best and TPR*-tree exhibits a worst performance. This is because the MBRs PRA-tree overlap relatively less than those of aTPR-tree and TPR*-tree, thus having a good query performance. While TPR*-tree and aTPR-tree may visit many unnecessary nodes for the massive overlaps between area of MBRs with time. Furthermore, PRA-tree exploits more precise branch-and-bound strategy, thus having a best performance.
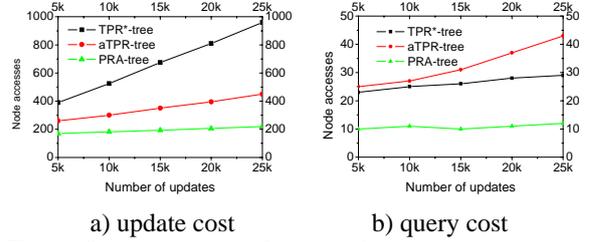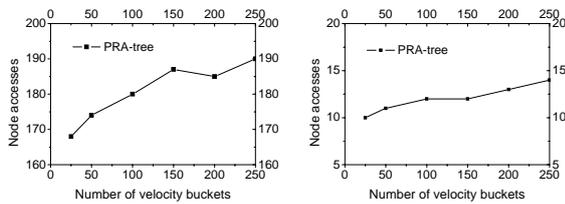

a) update cost   b) query cost
Figure 8 comparison of update& query performance

Figure 8 compares the average PRWA query and update cost as a function of the number of updates. As shown in figure 8 a), the node accesses needed in PRA-tree update execution are far less than TPR*-tree and aTPR-tree. And PRA-tree and TPR*-tree have nearly constant update cost This is because PRA-tree exploits the bottom-up deletion strategy, avoiding the excessive node accesses for deletion search, while TPR*-tree and aTPR-tree process update in top-down manner, needing more node accesses.

Figure 8 b) shows the node accesses in processing one PWRA query as function of an interval of 5k updates when fixing the parameters *qTlen*=50 and *qRlen*=1000. It is clear that the query cost increases with the number of updates. The PRA-tree has a slow increasing query cost, while the cost of TPR*-tree and aTPR-tree increase

significantly. This is because the MBRs of PRA-tree extend less than those of TPR*-tree and aTPR-tree, so the degradation is not much expensive.

a) effect on update cost   b) effect on query cost

Figure 9 effect of velocity bucket number

Figure 9 shows the effect of velocity buckets number on PWRA query and update performance of PRA-tree. We fix the parameters $qTlen$=50 and $qRlen$=1000. As shown in Figure9 a), the update cost of PRA-tree increase slightly with the number of velocity buckets, because the small velocity bucket window causes the shift of moving objects between buckets, thus incurs excessive node accesses. From Figure 9 b) we can see, the PWRA query performance will degrade with too many buckets, this is because of, as mentioned earlier, the uncertainty of moving objects distribution in velocity and space, the probability of overlaps between the area covered by velocity buckets and query region don't decrease linearly as expected. However, it may bring more TPR-tree node accesses. In this experiment, we set the number of velocity bucket equal to 25.

7. Conclusion

This paper investigates the problem of PRA queries. Our contribution is a novel indexing method, referred to as predicted range aggregate R-tree (PRA-tree). PRA-tree considers the distribution of moving objects both in velocity and space domain. First, we partition the velocity domain into velocity buckets, and then we use aTPR-tree to index the moving objects in each bucket. To support frequent updates a supplemented hash index on leaf nodes is added to PRA-tree. Also an extended bottom-up deletion algorithm is developed for PRA-tree.

An open problem is to extend our work to support large scale of concurrent PRA queries efficiently. Further, the PRA query results need to be revaluated for the frequent object updates, so developing novel incremental algorithms for PRA queries is also an interesting work.

References

[1] Yufei Tao and Dimitris Papadias, Range Aggregate Processing in Spatial Databases. *IEEE TKDE*, NO.12, pp.1555-1570, 2004.

[2] Ine´s Fernando Vega Lo´pez, Richard T. Snodgrass, and Bongki Moon. Spatiotemporal Aggregate Computation: A Survey. *IEEE TKDE*, NO.2, pp.271-286, 2005.

[3] Simonas Saltenis, Christian S. Jensen, et al.. Indexing the Positions of Continuously Moving Objects. *Proc. SIGMOD Conf.*, pp.331-342, 2000.

[4] Tao Y., Papadias D., and Sun J.. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. *Proc. VLDB Conf.*, pp.790-801, 2003.

[5] Jignesh M. Patel, Yun Chen, V. Prasad Chaka. STRIPES: An Efficient Index for Predicted Trajectories. *Proc. SIGMOD Conf.*, pp.635-646, 2004.

[6] Yufei Tao, Dimitris Papadias, Jian Zhai, and Qing Li. Venn Sampling: A Novel Prediction Technique for Moving Objects. *Proc. Int'l Conf. Data Eng.*, pp.680-691, 2005.

[7] Yufei Tao, Jimeng Sun, Dimitris Papadias. Selectivity Estimation for Predictive Spatio-Temporal Queries. *Proc. Int'l Conf. Data Eng.,* pp.417-428, 2003.

[8] M. Jurgens and H. Lenz. The Ra*-Tree: An Improved R-Tree with Materialized Data for Supporting Range Queries on OLAP-Data. *Proc. DEXA Workshop*, 1998.

[9] Bin Lin and Jianwen Su. On bulk loading TPR-tree. *Proc. IEEE Conf. Mobile Data Management (MDM)*, pp.114-124, 2004.

[10] M. Lee, W. Hsu, C. Jensen, B. Cui, and K. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. *Proc. VLDB Conf.*, pp.608-619, 2003.

[11] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, Vol.6 (2), pp.153-180, Kluwer, 2002.