Update-efficient Indexing of Moving Objects in Road Networks

Jidong Chen   Xiaofeng Meng   Yanyan Guo   Zhen Xiao

School of Information, Renmin University of China
{chenjd, xfmeng, guoyy, xiaozhen}@ruc.edu.cn

Abstract

Recent advances in wireless sensor networks and positioning technologies have boosted new applications that manage moving objects. In such applications, a dynamic index is often built to expedite evaluation of spatial queries. However, development of efficient indexes is a challenge due to frequent object movement. In this paper, we propose a new update-efficient index method for moving objects in road networks. We introduce a dynamic data structure, called *adaptive unit*, to group neighboring objects with similar movement patterns. To reduce updates, an adaptive unit captures the movement bounds of the objects based on a prediction method, which considers the road-network constraints and stochastic traffic behavior. A spatial index (e.g., R-tree) for the road network is then built over the adaptive unit structures. Simulation experiments, carried on two different datasets, show that an adaptive-unit based index is efficient for both updating and querying performance.

Keywords   Spatial-Temporal Databases, Moving Objects, Index Structure, Road Networks

1   Introduction

Recent advances in wireless sensor networks and positioning technologies have enabled a variety of new applications such as traffic management, fleet management, and location-based services that manage continuously changing positions of moving objects [11, 12]. In such applications, a dynamic index is often built to expedite evaluation of spatial queries. However, existing dynamic index structures (e.g. B-tree and R-tree) suffer from poor performance due to the large overhead of keeping the index updated with the frequently changing position data. Development of efficient indexes to improve the update performance is an important challenge.

Current work on reducing the index updates of moving objects mainly contains three kinds of approaches. First, most efforts [4, 9, 10, 15] focus on the update optimization of the existing multi-dimensional index structures especially the adaptation and extension of the R-tree [6]. To avoid the multiple paths search operation in the R-tree during the top-down update, recent work proposes the bottom-up approach [9, 10] and memo-based [15] structure to reduce the updates of the R-tree. Another method [4] exploits the change-tolerant property of the index structure to reduce the number of updates that cross the MBR boundaries of R-tree.

However, the indexes based on MBRs exhibit high concurrency overheads during node splitting, and each individual update is still costly. Therefore, some index methods based on a low-dimensional index structure (e.g. B^+-tree) are proposed [7, 16], which construct the second category of index methods. They combine the dimension reduction and linearization technique with a single B^+-tree to efficiently update the index structure.

The third kind of approaches use a prediction method with a time-parameterized function to reduce the index updates [12, 13, 14]. They describe a moving object's location by a linear function and the index is updated only when the parameters of the function change, for example, when the moving object changes its speed or direction. The MBRs of the index vary with the time as a function of the enclosed objects. However, the linear prediction is hard to reflect the movement in many real application and therefore leads to low prediction accuracy and frequent updates.

Though these index structures solve the problem of index updates to some extent, they are designed to index objects performing free movement in a two-dimensional space. We focus on the index update problem in real life environments, where the objects move within constrained networks, such as vehicles on roads. In such setting, the spatial property of objects' movement is captured by the network. Therefore, the

spatial location of moving objects can be indexed by means of the road-network index structure. For example, moving objects can be accessed by each road segment indexed by the R-tree. Since the road network seldom change and objects just move from one part to the other part of the network, the R-tree in this case remains fixed. Existing index work that handles network-constrained moving objects [1, 5, 11] is based on this feature. They separate spatial and temporal components of the moving objects' trajectories and index the spatial aspect by the network with a R-tree. However, they are mostly concerned with the historical movement and therefore they do not consider the problem of index updates.

In this paper, we address the problem of efficient indexing of moving objects in road networks to support heavy loads of updates. We exploit the constraints of the network and the stochastic behavior of the real traffic to achieve both high update and query efficiency. We introduce a dynamic data structure, called *adaptive unit* (AU for short) to group neighboring objects with similar movement patterns in the network. A spatial index (e.g., R-tree) for the road network is then built over the adaptive units to form the index scheme for moving objects in road networks. The index scheme optimizes the update performance for the following reasons: (1) An AU functions as a one-dimensional MBR in the TPR-tree [13], while it minimizes expanding and overlaps by considering more movement features. (2) The AU captures the movement bounds of the objects based on a prediction method, which considers the road-network constraints and stochastic traffic behavior. (3) Since the movement of objects is reduced to occur in one spatial dimension and attached to the network, the update of the index scheme is only restricted to the update of the AUs. We have carried out extensive experiments based on two datasets. The results show that an adaptive-unit based index not only improves the efficiency of each individual update but also reduces the number of updates and is efficient for both updating and querying performance.

The main contributions of this paper are:

- The introduction of Adaptive Units that optimize for frequent index updates of moving objects in road networks.

- An experimental evaluation and validation of the efficient update as well as query performance of the proposed index structure.

The rest of the paper is organized as follows. Section 2 surveys related work and introduces underlying model. Section 3 describes the structure and algorithms of adaptive units for efficient updates. Section 4 contains algorithm analysis and experimental evaluation. We conclude and propose the future work in Section 5.

2 Related Work and Underlying Model

2.1 Related Work

There are lots of efforts at reducing the need for index updates of moving objects. In summary, they can be classified into three categories.

First, most work focuses on the update optimization of existing multi-dimensional index structures especially the adaptation and extension of the R-tree [6]. The top-down update of R-tree is costly since it needs several paths for searching the right data item considering the MBR overlaps. In order to reduce the overhead, Kwon et al. [9] develop the Lazy Update R-tree, which is updated only when an object moves out of the corresponding MBR. With adding a secondary index on the R-tree, it can perform the update operation in the bottom-up way. Recently, by exploiting the change-tolerant property of the index structure, Cheng et al. [4] present the CTR-tree to maximize the opportunity for applying lazy updates and reduce the number of updates that cross MBR boundaries. [10] extends the main idea of [9] and generalizes the bottom-up update approach. However, they are not suitable to the case where consecutive changes of objects are large. Xiong and Aref [15] present the RUM-tree that processes R-tree updates in a memo-based approach, which eliminates the need to delete the old data item during an index update. Therefore, its update performance is stable with respect to the changes between consecutive updates. In our index structure, however, the R-tree remains fixed since it indexes the road network and only the adaptive units are updated.

The second type of methods are based on the dimension reduction technique [11] and a low-dimensional index [7, 16] (e.g. B^+-tree). The B^x-tree [7, 16] combine the linearization technique with a single B^+-tree to efficiently update the index structure. They uses space filling curves and a pre-defined time interval to partition the representation of the locations of the moving objects. This makes the B^+-tree capable to index the two-dimensional spatial locations of moving objects. Therefore, the cost of individual update of index is reduced. However, the B^x-tree imposes discrete representation and may not keep the precise values of location and time during the partitioning. For our setting, the two-dimensional spatial locations of moving objects can be reduced to the 1.5 dimensions [8] by the road network where objects move.

The techniques in third category use a prediction method represented as the time-parameterized function to reduce the index updates [12, 13, 14]. They store the parameters of the function, e.g. the velocity and the starting position of an object, instead of the real positions. In this way, they update the index structure only when the parameters change (for example, the speed or the direction of a moving object changes). The Time-Parameterized R-tree (TPR-tree) [13] and its variants (e.g. TPR*-tree) [12, 14] are

the examples of this type of index structures. They all use a linear prediction model, which relates objects' positions as a linear function of time.However, the linear prediction is hard to reflect the movement in many real application especially in traffic networks where vehicles change their velocities frequently. The frequent changes of the object's velocity will incur repeated updates of the index structure. Our technique also fall into this category and apply an accurate prediction method we proposed in [3] by considering more transportation features.

Several methods have been proposed for indexing moving objects in spatially constrained networks. Pfoser et al. [11] propose to convert the 3-dimensional problem into two sub-problems of lower dimensions through certain transformation of the networks and the trajectories. Another approach, known as the FNR-tree [5], separates spatial and temporal components of the trajectories and indexes the time intervals that each moving object spends on a given network link. The MON-tree approach [1] further improves the performance of the FNR-tree by representing each edge by multiple line segments (i.e. polylines) instead of just one line segment. However, they all focus on the historical movement and cannot support frequent index updates. To the best of our knowledge, there is no current index method to support efficient updates of moving objects in road networks.

2.2 Underlying Model

We use the GCA model we proposed in [3] to model the network and moving objects. A road network is modeled as a graph of cellular automata (GCA), where the nodes of the graph represent road intersections and the edges represent road segments with no intersections. Each edge consists of a cellular automaton (CA), which is represented, in a discrete mode, as a finite sequence of cells.

In the GCA, a moving object is represented as a symbol attached to the cell and it can move several cells ahead at each time unit. Intuitively, the velocity is the number of cells an object can traverse during a time unit. The motion of an object is represented as some (time, location) information. Generally, such information is treated as a trajectory.

3 The Adaptive Unit

3.1 Structure and Storage

Conceptually, an adaptive unit is similar to a one-dimensional MBR in the TPR-tree, that expands with time according to the predicted movement of the objects it contains. However, in the TPR-tree, it is possible that an MBR may contain objects moving in opposite directions, or objects moving at different speeds. As a result, the MBR may expand rapidly, which may create large overlaps with other MBRs. The AU avoids this problem by grouping objects having similar mov-

ing patterns. Specifically, for objects in the same network edge, we use a distance threshold and a speed threshold to cluster the adjacent objects with the same direction and similar speed. In comparison, the AU has no obvious enlargement because objects in the AU move in a cluster.

We now formally introduce the AU. An AU is a 8-tuple:

$$AU = (\texttt{auID}, \texttt{objSet}, \texttt{upperBound}, \texttt{lowerBound},$$
$$\texttt{edgeID}, \texttt{enterTime}, \texttt{exitTime}, \texttt{auInitLen})$$

where `auID` is the identifier of the AU, `objSet` is a list that stores objects belonging to the AU, `upperBound` and `lowerBound` are upper and lower bounds of predicted future trajectory of the AU. The trajectory bounds will be explained in details in Section 3.3. We assume the functions of trajectory bounds as follows:

$$\texttt{upperBound}: \quad D(t) = \alpha_u + \beta_u \cdot t$$
$$\texttt{lowerBound}: \quad D(t) = \alpha_l + \beta_l \cdot t$$

`edgeID` denotes the network edge that the AU belongs to, `enterTime` and `exitTime` record the time when the AU enters and leaves the edge and `auInitLen` represents the initial length of the AU.

In the road network, multiple AUs are associated with a network edge. Since AUs in the same edge are likely to be accessed together during query processing, we store AUs by clustering on their `edgeID`. That is, the AUs in the same edge are stored in the same disk pages. To access AUs more efficiently, we create an in-memory, compact summary structure called the *direct access table* for each edge. A direct access table stores the summary information of each AU on an edge (i.e. number of objects, trajectory bounds) and pointers to AU disk pages. Each AU corresponds to an entry in the direct access table, which has the following structure (`auID`, `upperBound`, `lowerBound`, `auPtr`, `objNum`), where `auPtr` points to a list of AUs in disk storage and `objNum` is the number of objects included in the AU. In order to minimize I/O cost, we use the direct access table to filter AUs and only access the disk pages when necessary.

3.2 The Index Scheme

We build a spatial index (e.g., R-tree) for the road network over the adaptive units to form the index scheme for the network-constrained moving objects. The AU index scheme is a two-level index structure. At the top level, it consists of a 2D R-tree that indexes spatial information of the road network. On the bottom level, its leaves contain the edges representing road segments included in the corresponding MBR of the R-tree and point to the lists of adaptive units. The top level R-tree remains fixed during the lifetime of the index scheme (unless there are changes in the network). The index scheme is developed with the R-tree
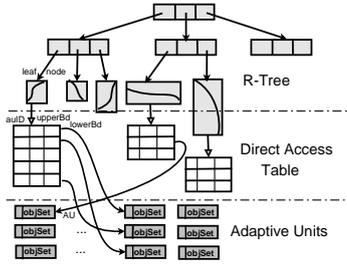
Figure 1: Structure of the AU index scheme

(a) Simulated trajectories   (b) Trajectory bounds

Figure 2: The simulation-based prediction

in this paper, but any existing spatial index can also be used without changes.
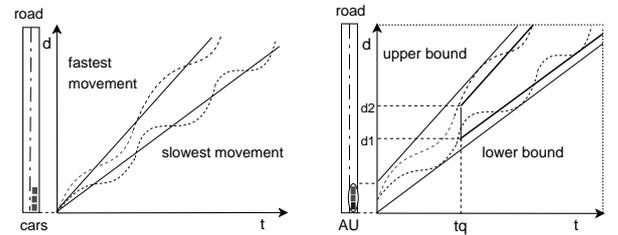
Figure 1 shows the structure of the AU index scheme, which also includes the direct access table. The R-tree and adaptive units are stored in the disk. However, the direct access table is in the main memory since it only keeps the summary information of adaptive units. In the index scheme, each leaf node of the R-tree can be associated with its direct access table by its `edgeID` and the direct access table can connect to corresponding adaptive units by `auPtr` in its entries. Therefore, we only need to update the direct access table when AUs change, which greatly enhances the performance of the index scheme.

3.3 Optimizing for Updates

An important feature of the AU is that it groups objects having similar moving patterns. The AU is capable of dynamically adapting itself to cover the movement of the objects it contains. By tightly bounding enclosed moving objects for some time in the future, the AU alleviates the update problem of MBR rapid expanding and overlaps in the TPR-tree like methods.

For reducing the updates further, the AU captures the movement bounds of the objects based on a prediction method we proposed in [3], which considers the road-network constraints and stochastic traffic behavior. Since objects in an AU have similar movement, we then predict the movement of the AU, as if it were a single moving object. In the following, we describe the application and adaptation of the prediction method to the AU.

We use GCAs not only to model road networks, but also to simulate the movements of moving objects by the transitions of the GCA. Based on the GCA, the *Simulation-based Prediction (SP)* method to anticipate future trajectories of moving objects is proposed. The SP method treats the objects' simulated results as their predicted positions. Then, by the linear regression, a compact and simple linear function that reflects future movement of a moving object can be obtained. To refine the accuracy, based on different assumptions on the traffic conditions we simulate two future trajectories to obtain its predicted movement function. Specifically, we extend the CA model used in traffic flow simulation for predicting the future

trajectories of objects by setting $P_d(i)$ to values that model different traffic conditions. In this setting, $P_d(i)$ is treated as a random variable to reflect the stochastic, dynamic nature of traffic system. By giving $P_d(i)$ two values (e.g. 0 and 0.1 in our experiments), we can derive two future trajectories, which describe, respectively, the fastest and slowest movements of objects. Finally, we translate the two regression lines, until all estimated future positions fall within to obtain the predicted trajectory bounds. The SP method is shown in Figure 2. Through the SP method, we obtain two predicted future trajectory bounds of objects. We apply this technique to the AU - a set of moving objects that have similar movement and are treated as one object.

The future trajectory bounds are predicted as soon as AU is created. The trajectory bounds will not be changed along the edge that the AU moves on until the objects in the AU move to another edge in the network. It is evident that the range of predicted bounds of AU will become wider with the time, which leads to lower accuracy of future trajectory prediction. However, if we issue another prediction when the predicted bounds are not accurate any more, the costs of simulation and regression are high. Considering that the movement of objects along one network edge is stable, we can assume the same trends of the trajectory bounds and adjust only the initial locations when the prediction is not accurate. Specifically, when the predicted position exceeds its actual position above the predefined accuracy, the AU treats its actual locations (the locations of the boundary objects) at that time as the initial locations of the two trajectory bounds and follow the same movement vector (e.g. slope of the bounds) as the previous bounds to provide more accurate predicted trajectory bounds. In this way, the predicted trajectory bounds can be effectively revised with few costs. Figure 2(b) shows the adaptation of the trajectory bounds. t_q is the time slice when actual locations of boundary objects in the AU exceeds the predicted bounds of the AU above precision threshold and the d_1, d_2 are the actual locations of the first object and last object respectively in the AU. The trajectory bounds are revised according to the actual locations and the original bounds' slopes. Therefore, without executing more prediction, the prediction accuracy of the objects' future trajectories can be kept high.

Since the R-Tree indexes the road network, it remains fixed, and the update of the AU index scheme restricts to the update of adaptive units. Specifically, an AU is usually created at the start of one edge and dropped at the end of the edge. Since the AU is a one-dimensional structure, it performs update operations much more efficiently than the two-dimensional indexes. We will describe these operations in details.

3.4 Update Operations

The update of an AU can be of the following form: creating an AU, dropping an AU, adding objects to an AU and removing objects from an AU.

Creating an AU

To create an AU, we first compose the $objSet$ – a list of objects traveling in the same direction with similar velocities, and in close-by locations. We then predict the future trajectories of the AU by simulation and compute its trajectory bounds. In fact, we treat the AU as one moving object (the object closest to the center of the AU) and predict its future trajectory bounds by predicting this object. The prediction starts when the AU is created and ends at the end the edge. Finally, we write the created AU to the disk page and insert the AU entry to its summary structure.

Dropping an AU

When objects in an AU move out of the edge, they may change direction independently. So we need to drop this AU and create new AUs in adjacent edges to regroup the objects. When the front of an AU touches the end of the edge, some objects in the AU may start moving out of the edge. However, the AU cannot be dropped because a query may occur at that time. Only after the last object in the AU enters another edge and joins another AU, can the AU be dropped. Dropping an AU is simple. Through its entry in direct access table, we find the AU and delete it.

Adding and removing objects from an AU

When an object leaves an AU, we remove this object from the AU and find another AU in the neighborhood to check if the object can fit that AU. If it can, the object will be inserted into that AU, otherwise, a new AU is created for this object. Specifically, when adding an object into an AU, we first find the direct access table of the edge that the object lies and, by its AU entry in the table, access the AU disk storage. Finally, we insert into the objects list of the AU and update the AU entry in the direct access table. Removing an object from an AU has the similar process.

Therefore, when updating an object in the AU index scheme, we first determine whether the object is leaving the edge and entering another one. If it is moving to another edge, we delete it from the old AU (if it is the last object in the old AU, the AU is also dropped) and insert it into the nearest AU or create a new AU in the edge it is entering. Otherwise, we do not update

the AU that the object belongs to unless its position exceeds the bounds of the AU. In that case, we execute the same updates as those when it moves to another edge or only revise the predicted trajectory bounds of the AU. Factually, we find, from the experiment evaluation, that the chances that objects move beyond the trajectory bounds of its AU on an edge are very slim. The algorithm 1 shows the update algorithm of AUs.

Algorithm 1: Update($objID, position, velocity, edgeID$)

> **input** : $objID$ is the object identifier, $position$ and $velocity$ are its position and velocity, $edgeID$ is the edge identifier where the object lies

Find AU where $objID$ is included before update;
if $AU.edgeID \neq edgeID$ or (position $<$ $AU.lowerBound$ or position $> AU.upperBound$) **then**

> // The object moves to a new edge or exceeds bounds of its original AU
> Find the nearest AU AU_1 for $objID$ on $edgeID$;
> **if** $GetNum(AU_1.objSet) < MAXOBJNUM$ and $ObjectFitAU(objID, position, velocity, AU_1)$ **then**
>> InsertObject($objID, AU_1.auID, AU_1.edgeID$);
>
> **else** $AU_2 \leftarrow$ CreateAU($objID, edgeID$);
> **if** $GetNum(AU.objSet) > 1$ **then**
>> DeleteObject($objID, AU.auID, AU.edgeID$);
>
> **else** DropAU($AU.edgeID, AU.auID$);

end

In summary, updating the AU-based index is easier than updating the TPR-tree. It never invoke any complex node splitting and merging. Moreover, thanks to the similar movement features of objects in an AU and the accurate prediction of the SP method, the objects are seldom removed or added from their AU on an edge, which reduces the number of index updates.

3.5 Query Algorithm

Query processing in the AU index scheme is straightforward. Given a query, we use the top level R-tree to get the edges involved and then scan the direct access tables of the edges. With the `upperBound` and the `lowerBound` in the direct access table, we can easily find AU entries that intersect the query, and then visit the disk pages to get more information about these AUs. For space limitation, we just take window range query for example. Given a range with (X_1, Y_1, X_2, Y_2), we first perform a spatial range search in the top level R-Tree to locate the edges (e.g. $e_1, e_2, e_3, \ldots$). For each selected edge e_i, we transform the original search (X_1, Y_1, X_2, Y_2) to a 1D search range (S_1, S_2) $(S_1 \leq S_2)$, where S_1 and S_2 are the relative distances from the start vertex along the edge e_i. In the case of multiple intersecting edges, we can divide the query range into several sub-ranges by edges and apply the transformation method to each edge. The method is also applicable to the various modes

that the query and edges intersect. Here, we only illustrate the case when the query window range only intersects one edge and compute its relative distances S_1 and S_2. It can be easily extended to other cases. Suppose $X_{start}, Y_{start}, X_{end}, Y_{end}$ are the start vertex coordinates and the end vertex coordinates of the edge e_i. According to Thales Theorem about similar triangles, we obtain S_1 and S_2 as follows:

$$r = \sqrt{(X_{start} - X_{end})^2 + (Y_{start} - Y_{end})^2}$$
$$S_1 = \frac{X_1 - X_{start}}{X_{end} - X_{start}} r$$
$$S_2 = \frac{Y_1 - Y_{start}}{Y_{end} - Y_{start}} r$$

The transformed query (S_1, S_2) is then executed in each of the AUs in the direct access table of the corresponding edge e_i. By the trajectory bounds of the AU, we can determine whether the transformed query intersects the AU, thus filtering the unnecessary AUs quickly. Finally, we access the selected AUs in disk storage and return the objects satisfying the query window. In summary, the query processing is efficient due to the grouping of similar objects in AUs and the dimensionality reduction of the query.

4 Performance Analysis

We evaluate the AU index scheme (denoted as "AU index") by comparing it with the TPR-tree and the AU index scheme when the direct access table is not used (denoted as "AU index without DT"). We measure their their update performance with the individual update, update frequency and total update costs and their query performance.

4.1 Datasets

We use two datasets for our experiments. The first is generated by the CA simulator, and the second by the Brinkhoff's Network-based Generator [2]. We use the CA traffic simulator to generate a given number of objects in a uniform network of size 10000×10000 consisting of 500 edges. Each object has its route and is initially placed at a random position on its route. The initial velocities of the objects follow a uniform random distribution in the range $[0, 30]$. The location and velocity of every object is updated at each time-stamp. The Brinkhoff's Network-based Generator is used as a popular benchmark in many related work. The generator takes a map of a real road network as input (our experiment is based on the map of Oldenburg including 7035 edges). The positions of the objects are given in two dimensional X-Y coordinates. We transform them to the form of (`edgeid`, `pos`), where `edgeid` denotes the edge identifier and `pos` denotes the object relative position on the edge. The generator places a given number of objects at random positions on the road network, and updates their locations at each time-stamp.

Table 1: Parameters and their settings

Parameters	Settings
Page size	4K
Node capacity	100
Numbers of queries	200
Numbers of mo(cars)	10K, ... , 50K, ... , **100K**
Numbers of updates	**100K**, ... , 500K, ... , 1M
Dataset Generator	CA Simulator, Network-based Generator

We implemented both the AU index scheme and the TPR-tree in Java and carried out experiments on a Pentium 4, 2.4 GHz PC with 512MB RAM running Windows XP. To improve the performance of the index structure, we employed a LRU buffer of the same size as the one used in the TPR-tree [13]. We summarize workload parameters in Table 1, where values in bold are default values.

4.2 Update Cost

We compare the cost of index update for the AU index and the TPR-tree in terms of the average individual update cost, update frequency and total update cost.

Individual Update Cost

We study the individual update performance of the index while varying the number of moving objects and updates. Figure 3 shows the average individual update cost when increasing the data size from 10K to 100K. Figure 4 shows how the performance varies over time. Clearly, updating the TPR-tree tends to be costly, and the problem is exacerbated when the data size increases. In each case of different data size and different number of updates, the AU index has much lower update cost than the TPR-tree. The main reason can be explained as follows. Each update of the TPR-tree involves the search of an old entry and a new entry, as well as the modification of the index structure (node splitting, merging, and the propagating of changes upwards). The cost increases with larger data size due to more overlaps among MBRs. The changes of index structure with the increase of data updates also affect the performance of the TPR-tree. However, the AU index has better performance because its update only restricts to the AU's update and as a one-dimensional access structure, the AU has few overlaps and incurs no cost associated with node splitting and the propagation of MBR updates.

The direct access table of the AU index has a significant contribution in improving update performance. This is because the search of the specific AU is accelerated by the in-memory structure.

Update Frequency

Frequent updates of moving objects (a.k.a. data updates) may lead to frequent updates of index. When an object's position exceeds the MBR or AU, the index needs to be updated to delete the object from the old MBR or AU and insert it to another one. In this experiment, we measure the index update rate, which is
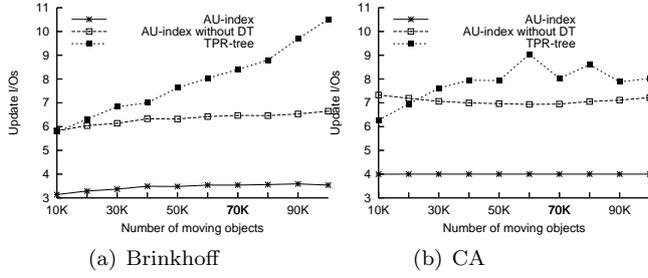
(a) Brinkhoff

(b) CA

Figure 3: Individual Update Cost with Different Datasize
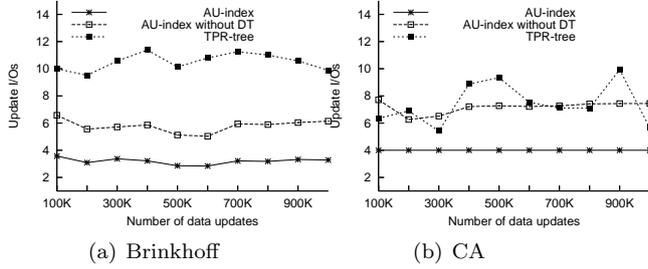

(a) Brinkhoff

(b) CA

Figure 4: Individual Update Cost over Time

the ratio between number of index updates and number of data updates, for every 100K data updates and different data size. Figure 5 and 6 show that the update rate of the TPR-tree is nearly 4 to 5 times more than that of the AU index. The index update rate depends on the prediction method. In the AU index, the future positions of the object are predicted more accurately, so the object is likely to remain in its AU, which leads to fewer index updates.

Total Update Costs

The total update costs depend on the update frequency and the average individual update cost, and it can reflect the index update performance more accurately. From both Figure 7 and 8, we can see that although the AU index has to deal with the creation and dropping of AUs, the TPR-tree incurs much higher update costs than the AU index and its performance deteriorates dramatically as data size increases. This is mainly due to the inaccuracy of the linear prediction model and the complex reconstruction of the TPR-tree (e.g. splitting and merging).
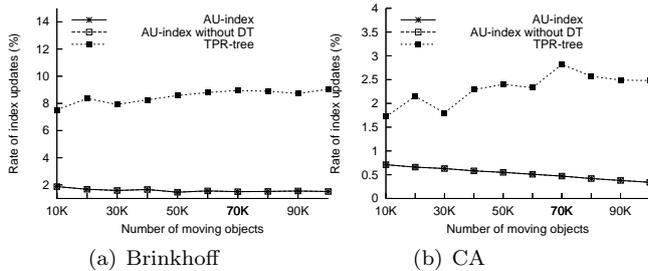

(a) Brinkhoff

(b) CA

Figure 5: Index Update Frequency with Different Datasize

(a) Brinkhoff

(b) CA

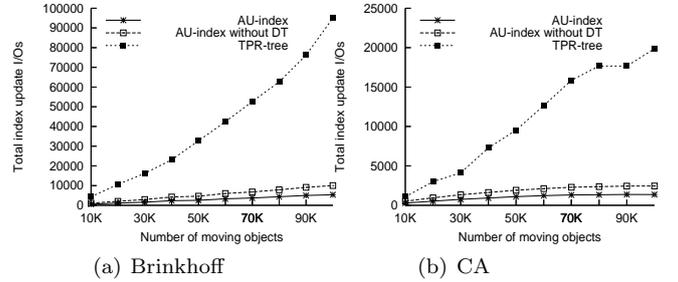*Figure 6:* Index Update Frequency over Time

(a) Brinkhoff

(b) CA

Figure 7: Total Update Cost with Different Datasize

For each data size, the update costs of the two indexes in the Brinkhoff's dataset are both higher than those in the CA dataset due to the higher complexity of road network and skewed spatial distribution of objects in the Brinkhoff's dataset.

4.3 Query Cost

We study the window range query performance of the TPR-tree and the AU index with different update settings. We increase the number of updates from 100K to 1M to examine how query performance is affected. We issued 200 range queries for every 100K updates in a 1M dataset. Figure 9 shows that the cost of the TPR-tree increases much faster as the number of updates increases. The cost of the AU index is considerably lower and is less sensitive to the number of updates. This is because the adaptive units in the AU index have much less overlaps than the MBRs in the TPR-tree, and the overlaps to a large extent determine the range query cost. Besides, as objects move apart, the amount of dead space in the TPR-tree increases,
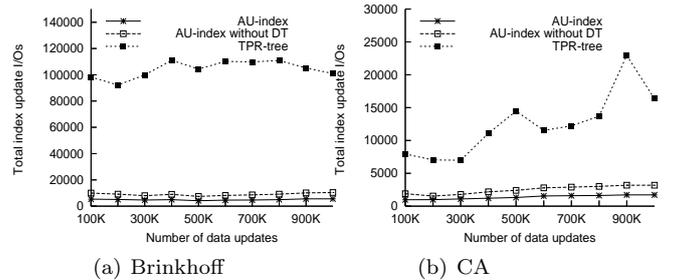

(a) Brinkhoff

(b) CA

Figure 8: Total Update Cost over Time

15

which makes false hits more likely. Also, updates lead to the expanding and overlaps of MBRs, which further deteriorate the performance of the TPR-tree. For the AU index, the increase of the updates hardly affect the total number of AUs, and the chances of overlaps of different AUs are very slim.

We also study the query performance while varying the number of moving objects and query window size. For the space limitation, we do not report the experimental results. Also, in each case, the AU index has lower query cost than the TPR-tree and scales well.
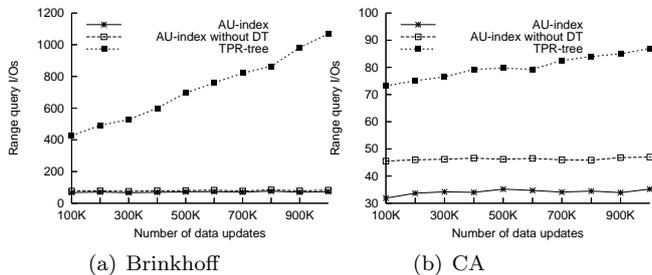

(a) Brinkhoff   (b) CA

Figure 9: Effect of Updates on Query Performance

5   Conclusions and Future Work

Indexing objects moving in a constrained network especially the road network is a topic of great practical importance. We focus on the index update issue for the current positions of network-constrained moving objects. We introduce a new access structure, adaptive unit that exploits as much as possible the characteristics of the movements of objects. The updates of the structure are minimized by an accurate prediction method which produces two trajectory bounds based on different assumptions on the traffic conditions. The efficiency of the structure also results from the possible reduction of dimensionality of the trajectory data to be indexed. Our experimental results performed on two datasets show that the efficiency of the index structure is one order of magnitude higher than the TPR-tree.

In the future, we will compare the update performance with the work of the R-tree-based updating optimization such as RUM-tree [15] and CTR-tree [4]. On the other hand, since the adaptive units contain the predicted future trajectories of moving objects, the predictive query algorithms can be developed naturally based on the adaptive unit-based index. Furthermore, we will extend the query algorithms to support the KNN query and continuous query for moving objects in the road network.

Acknowledgments

References

[1] V. T. Almeida, R. H. Güting. Indexing the Trajectories of Moving Objects in Networks (Extended Abstract). In SSDBM, 2004, 115-118.

[2] T. Brinkhof. A framework for generating network-based moving objects. In GeoInformatica, 6(2), 2002, 153-180.

[3] J. Chen, X. Meng, Y. Guo, S. Grumbach, H. Sun. Modeling and Predicting Future Trajectories of Moving Objects in a Constrained Network. In MDM, 2006, 156 (MLASN workshop).

[4] R. Cheng, Y. Xia, S. Prabhakar, R. Shah. Change Tolerant Indexing for Constantly Evolving Data. In ICDE, 2005, 391-402.

[5] E. Frentzos. Indexing objects moving on Fixed networks. In SSTD, 2003, 289-305.

[6] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In SIGMOD, 1984, 47-57.

[7] C. S. Jensen, D. Lin, B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In VLDB, 2004, 768-779.

[8] G. Kollios, D. Gunopulos, V. J. Tsotras. On indexing mobile objects. In PODS, 1999, 261-272.

[9] D. Kwon, S. J. Lee, S. Lee. Indexing the current positions of moving objects using the lazy update R-tree. In MDM, 2002, 113-120.

[10] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In VLDB, 2003, 608-619.

[11] D. Pfoser, C. S. Jensen. Indexing of network constrained moving objects. In ACM-GIS, 2003, 25-32.

[12] S. Saltenis, C. S. Jensen. Indexing of Moving Objects for Location-Based Service. In ICDE, 2002, 463-472.

[13] S. Saltenis, C. S. Jensen, S. T. Leutenegger, M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In SIGMOD, 2000, 331-342.

[14] Y. Tao, D. Papadias, J. Sun. The TPR*-Tree: An Optimized Spatiotemporal Access Method for Predictive Queries. In VLDB, 2003, 790-801.

[15] X. Xiong, W. G. Aref. R-trees with Update Memos. In ICDE, 2006, 22.

[16] M. L. Yiu, Y. Tao, N. Mamoulis. The $B^{dual-Tree}$: Indexing Moving Objects by Space-Filling Curves in the Dual Space. To appear in Very Large Data Base Journal, 2006.