Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree *

Dongseop Kwon Sangjun Lee Sukho Lee
School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea
{subby,freude}@db.snu.ac.kr, shlee@cse.snu.ac.kr

Abstract

With the rapid advances of wireless communications and positioning techniques, tracking the positions of moving objects is becoming increasingly feasible and necessary. Traditional spatial index structures are not suitable for storing these positions because of numerous update operations. To reduce the number of update operations, many existing approaches use a linear function to describe the movements of objects. In many real applications, however, the movements of objects are too complicated to be represented as a simple linear function. In this case, such approaches based on a linear function cannot reduce update cost efficiently. In this paper, we propose a novel R-tree based indexing technique called LUR-tree. This technique updates the structure of the index only when an object moves out of the corresponding MBR (minimum bounding rectangle). If a new position of an object is in the MBR, it changes only the position of the object in the leaf node. It can update the position of the object quickly and reduce update cost greatly. Since it is based on the R-tree, the LUR-tree also uses the same algorithms to process various types of queries as the R-tree. We present the experimental results which show that our technique outperforms other techniques

1. Introduction

Mobile computing is now a reality owing to the rapid advances of wireless communication technologies and electronic technologies. It is estimated that there will be more than 500 million mobile phones in use by year 2002, and 1 billion by 2004 [12, 14]. The appearance of powerful portable computers and the development of fast reliable wireless networks change computing paradigms. The mobility will create an entire new class of applications and possibly new massive markets combining personal computing and consumer electronics [3]. With the advances of positioning systems such as Global Positioning System (GPS) and the progress of mobile computing environments, tracking the changing positions of moving objects is becoming increasingly feasible and necessary. The need for storing and processing continuously moving data arises in a wide range of applications, including traffic control or monitoring, transportation and supply chain managements, digital battlefields, and mobile e-commerce [17].

Since traditional database systems assume that data stored in the database remain constant unless it is explicitly modified, they are not appropriate for representing, storing, and querying dynamic attribute such as moving object because the database has to be updated continuously. Moreover, if traditional multi-dimensional index structures such as R-tree [7] are used for indexing moving objects, similar problems still remain. The modification of location may need to split or merge the nodes. This requires additional overheads. In the worst case, frequent index rebuilding may be required. For these reasons, R-tree based index structures are not proper to index dynamic attributes such as the positions of continuously moving objects.

To reduce the number of update operations, many existing approaches use a simple linear function to express the movements of objects. In these approaches, the database is updated only when the parameters of the function change. In many real application, however, a good function for describing the movements hardly exists. If the movements of objects are very complicated, a simple linear function is not suitable for describing the movements. In this situation, a lot of update operations are needed. Although threshold technique can be used for preventing numerous updates, it makes the data inaccurate. Therefore, it is not proper to the applications that require a high accuracy.

In this paper, to solve these problems, we propose a new indexing technique, the Lazy Update R-tree (LUR-tree), which efficiently indexes the current positions of moving objects and reduces update cost greatly. In this technique, we remove unnecessary modification of the tree while updating the positions. This technique updates the index structure only when an object moves out of the corresponding

*This work was supported by the Brain Korea 21 Project.

MBR (minimum bounding rectangle). If a new position of an object is in the MBR, the LUR-tree changes only the position of the object in the leaf node. It can update the position of the object quickly and reduce update cost greatly. To increase the performance of LUR-tree, we also use an Extended MBR (EMBR). We extend the size of an MBR to keep the objects a little longer. Since the LUR-tree naturally extends the R-tree, it can use the same algorithms to process various types of queries (e.g. range queries, k-nearest queries, spatial join queries) as the R-tree. We implement the LUR-tree based on R*-tree [5] and carry out experiments under various environments.

The rest of this paper is organized as follows: In Section 2, we review related work, and discuss their advantages and disadvantages. Section 3 presents the problem being addressed. In Section 4, we propose a novel, R-tree based indexing technique to reduce update cost. In Section 5, we present the experimental results to compare our technique with existing approaches. Finally, conclusions and future work are discussed in Section 6.

2. Related Work

A detailed summary of work on spatio-temporal databases can be found in [1]. Developing efficient index structure is an important research issue of spatio-temporal databases. As the simplest approach, multi-dimensional spatial index structures can be utilized for indexing the positions of moving objects. A number of index structures have been proposed for handling multi-dimensional data, and a survey of these indexing methods can be found in [6]. Although traditional spatial index structures can be used, they are not suitable for storing the positions of moving objects because of numerous update operations. In case of the R-tree, this update can cause a node splitting or merging. It is clearly an inefficient and infeasible solution to index moving objects.

In recent times, many new index structures have been proposed to overcome the problems of spatial indexing techniques. According to the type of data being stored, the following two categories exist: (a) index the past positions of objects (i.e. trajectories) and (b) index the current positions of objects. Our technique belongs to the latter category.

In the former category, the movement of one object in a d-dimensional space is described as a trajectory in a $(d+1)$-dimensional space after combining time into same space. In [10], the authors introduced two access methods called Spatio-Temporal R-tree (STR-tree) and Trajectory-Bundle tree (TB-tree). They claimed that these two tree structures worked better than traditional R-tree family for trajectory-based queries. Recently, Tao et al. [14] proposed the Multi-version 3D R-tree(MV3R-tree), a structure that combines the concepts of multi-version B-trees [4] and 3D-Rtrees.

In the latter category, most existing approaches describe each object's location as a simple linear function, and update the database only when the parameters of the function change (for example, when the speed or the direction of a car changes). Tayeb et al. [15] addressed the issue of indexing moving objects to query their present and future positions. They proposed a method to index moving objects using PMR-Quadtree. Kollis et al. [8] proposed an efficient indexing scheme, based on partition trees. Agarwal et al. [2] proposed various efficient schemes based on the duality. They also developed an efficient indexing scheme to answer approximate nearest-neighbor queries among moving points. Saltenis et al. [12] proposed the time-parameterized R-tree (TPR-tree). In their scheme, the position of a moving point was represented by a reference position and a corresponding velocity vector. When splitting nodes, the TPR-tree considered not only the positions of the moving points but also their velocities. The problem of all these techniques is that a good function for describing the movements hardly exists. In many real applications, the movement of objects is complicated, not linear. In this situation, such approaches based on a linear function cannot reduce the update cost efficiently. To reduce the update cost, approximation technique using threshold can be used. Since, however, this approximation technique can decrease the accuracy, it is not suitable for the applications that need a high accuracy.

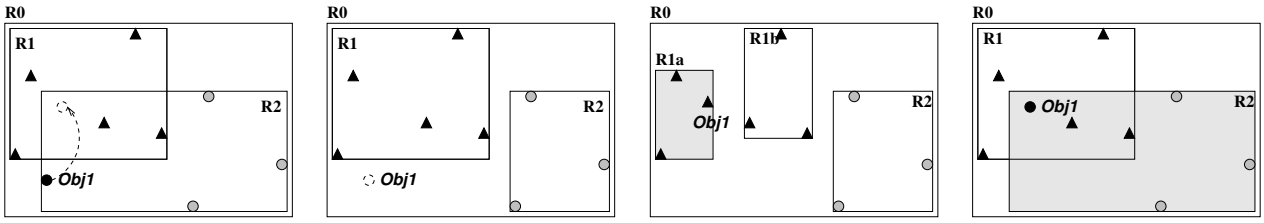
Recently, Song and Roussopoulos [13] proposed a new idea based on hashing to solve this problem. They inserted a new filter layer between the position information collectors and the database. The database stores the bucket information of each object instead of the exact location of the object. The database does not make any change until an object moves into a new bucket. Only the filter layer stores exact locations and changes the locations of objects. This approach is simple and intuitive. However, since it is based on a simple hashing, it can make long chains of overflow pages. This can affect performance because all pages in bucket have to be searched. If the distribution of data is skewed, the problem could become more serious.

3. Problem Description

In this section, we describe the problem that we discuss and solve in this paper. First, we will present a sample application scenario, the system for tracking automobiles. Then, we will introduce the problem setting and other assumptions.

3.1 An Application Scenario

The optimization of transportation, especially in highly populated areas, is a very challenging task that can be sup-



(a) Before Update : we want to move the position of *Obj1*

(b) Traditional Update : (Step 1) delete *Obj1* and adjust the MBR of *R2*

(c) Traditional Update : (Step 2) split the node and insert *Obj1* in a new position (*Obj1* belongs to *R1*)

(e) Lazy Update : only the position of *Obj1* is changed. *Obj1* belongs to *R2*

Figure 1. Example of Traditional Update Algorithm and Lazy Update Algorithm

ported by an information system. Consider the fleet management system. At the central site, there is a database system for managing all data of vehicles. Each vehicle is equipped with a GPS device. Vehicles measure their current locations with the GPS device and transmit their positions to the central site using either radio communication links or cellular phones. The database system stores the current positions of all vehicles and processes queries. Example queries occurring in such an application are as follows:

- Which taxi is closest to the Hilton hotel? (a k-nearest neighbor search query)
- Retrieve all taxies within 500 meters of the central station. (a range search query)
- Find all pairs of taxies and hospitals that are within 1 mile of each other (a spatial join query)

3.2 Problem Setting

The problem we focus on is the reduction of enormous update cost in indexing moving objects using a multi-dimensional index structure. Suppose that there are N moving objects in a d-dimensional space. The position of the ith object is given by $o_i = (p_1, p_2, \ldots, p_d)$, which is a point in a d-dimensional space. In every t_{int} time, an object measures and transfers its position to the database server (i.e. t_{int} is the interval of sampling the location). An update message which an object sends to the server is represented as (oid, p_{new}), where oid is a unique id of an object and p_{new} is a new position of the object.

Fundamentally, the representations of moving objects have some uncertainties [9]. We, however, assume that the accuracy of the GPS device and the sampling frequency in the GPS device is high enough. Under this assumption, we ignore the uncertainties in sampling. We also assume that the transmission time from a vehicle to the server is short enough and there are no errors in the network layer. We also ignore the uncertainties in the network. We ignore all the other uncertainties under the similar assumption.

The database maintains only the current positions. Whenever an update request issues, the database updates the position corresponding to the request. The database indexes the positions of objects with a multi-dimensional index structure such as R-tree. Our objective is to reduce the update cost in the database.

4 Lazy Update R-tree

In this section, we will present a novel index structure called Lazy Update R-tree (LUR-tree) for indexing moving objects. First, we introduce the basic idea of the lazy update approach. Then, we describe the structure of the LUR-tree. We also present a lazy update algorithm for the LUR-tree. Finally, we present the notion of an extended minimum bounding rectangle (EMBR).

4.1 Basic Idea

As stated in Section 2, using a spatial index such as an R-tree for indexing the positions of continuously moving objects has some problems. Consider the situation of Figure 1. We use an R-tree for indexing moving objects. *R1* and *R2* are MBRs of leaf nodes. *R0* is the MBR of the parent node of *R1* and *R2*. *Obj1* belongs to *R2* at the initial time. Assume the maximum cardinality of entries in the leaf node is 5. *R1* has 5 entries and *R2* has 4 (including *Obj1*). Triangles mean objects which belong to *R1* and Circles mean those which belong to *R2*. The situation is like this. *Obj1* reports its location is moved to a new position. Then, we want to update the position of *Obj1* like Figure 1(a). Since there is no special update algorithm in the standard R-tree, we have to delete the old position of *Obj1* and insert the new one. This may induce some serious problems: splitting or merging nodes. Like in Figure 1(b), after deleting *Obj1*, we have to adjust the MBR of *R2*. When we insert the new position, we select *R1* for the node into which the new position will be inserted. Since there are no room for a new entry in *R1*, we have to split *R1* into *R1a* and *R1b* like

Figure 1(c). Then, we can insert the new position of *Obj1* into *R1a*. In this case, even if we ignore disk accesses for traversing the tree and adjusting the node in a path from the leaf to the root, we need at least 6 disk accesses, i.e. 2 disk access for deleting (1 read and 1 write for *R2*) and 4 disk access for splitting and inserting (1 read for *R1*, 3 writes for *R1a*, *R1b*, and *R0*). In the worst case, a series of node splitting or merging can occurr.

We, however, do not have to update like Figure 1. If a new position of an object is in the MBR to which the object belongs (like Figure 1(a)), We do not have to move *Obj1* from *R2* to *R1*. We have only to update the position of the corresponding entry in the leaf node like Figure 1(d). Figure 1(d) shows that *Obj1* still belongs to *R2* in spite of its movement. In this case, we need only 2 disk accesses (1 read and 1 write for *R2*). We can reduce 4 disk accesses. Since the the method based on the deletion and insertion requires additional disk accesses for traversing and adjusting the tree, we also can reduce more disk accesses. In Figure 1(d), the MBR of *R2* is no longer minimum bounding rectangle. Therefore, we have to adjust the MBR of *R2* in the manner of the standard R-tree. We, however, need not adjust the MBR because it does not violate the semantics of the R-tree. The R-tree is valid only if the region of a parent node includes all regions of its child nodes.

4.2 Index Structure

Based on the basic idea, we propose a new index structure, the lazy update R-tree (LUR-tree). The LUR-tree is based on the R-tree. All algorithms and index structures are similar to those which are used in other R-tree variants. Only the lazy update algorithm is appended. Algorithms and data structure in the R-tree are slightly modified.

We assume that an update request consists of a pair of a object id and a new position (e.g. (oid, p_{new})). Therefore, we append an additional access path to find the leaf node which contains the object whose id is oid. We call this access path the *DirectLink*. The DirectLink is a kind of a secondary index structure. It is, however, somewhat different from other traditional secondary index structures. In the DirectLink, the key of the index structure is oid. Each index entry has a pointer to the corresponding entry in a leaf node. Any secondary index structure such as hash index or B-tree index can be used for the DirectLink. Figure 2 illustrates the structure of the LUR-tree and the DirectLink.

When an object is inserted or deleted, the leaf node offset of the object can be changed. When a node is split or merged, the leaf node offsets of several objects can be changed. In this case, the DirectLink must be updated to maintain the valid pointer to the leaf node. To simplify implementation, each leaf node can have its own MBR. But, this is not absolutely necessary because we can calculate the MBR from entries of the node.

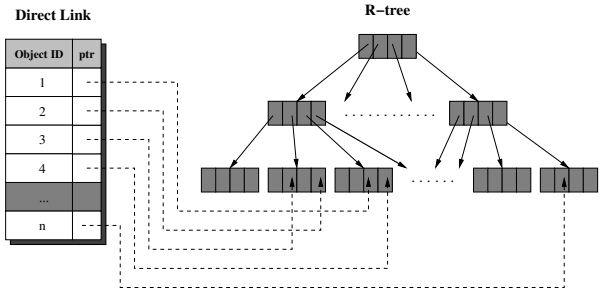


Figure 2. Structure of LUR-tree and DirectLink

4.3 Lazy Update Algorithm

Algorithm 1 describes the algorithm of the lazy update. In the first step, we find a leaf node to which an object belongs using the DirectLink. We read the leaf node and find an entry whose object id is equal to a given id. Then, we check whether or not the lazy update is possible, i.e. the MBR of the leaf node contains a new position or not. If the new position is in the MBR, we modify only the position of the object in the entry. Then, we write the node.

Algorithm 1: Lazy Update Algorithm

```
   Procedure LazyUpdate (oid, newpos)
   Input: an object's id oid, new position newpos
   begin
1:     blockid ← ReadDirectLink(oid);
2:     N ← ReadNode(blockid);
3:     E ← GetEntry(N, oid);
4:     if newpos ∈ N.MBR then
5:         E.pos ← newpos;
6:         WriteNode(N, blockid);
       else
7:         Delete(oid, E.pos);
8:         new_bid ← Insert(oid, newpos);
9:         UpdateDirectLink(oid, new_bid);
       end
   end
```

If a new position is not in the MBR, we need to update the tree in a different way. To process this type of update, we could use several methods as follows:

- **Deletion and Insertion** : This is an intuitive method that has been used in the R-tree. We delete an old position and insert a new one into the LUR-tree. This method is the same as the one used in the standard R-tree. It can cause splitting or merging problems.
- **Extension of MBR** : Instead of deleting an old position, we can just extend the MBR to include a new

position. Then, we adjust the MBR of its parent node. This method is applicable only if the new position is not far from the MBR. Otherwise, the MBR becomes too large, so that the search performance will be degraded.

- **Reinsertion into the Parent Node** : After deleting the old position, we insert the new one into the parent node, instead of inserting it into the leaf. If the parent node is full, we try to insert it into the parent node of the parent node. This method can reduce the number of disk access. However, each node has to keep the pointer of the parent node, so the fanout of the tree decreases.

For simplicity, we have used only the deletion and insertion method in Algorithm 1.

4.4 Extended MBR

An object that is on the boundary of an MBR can easily move out of the MBR. If an object zigzags along the boundary of an MBR like in Figure 3, deletions and insertions can occur continuously. This can affect the update performance. To prevent this problem, we can use a slightly large bounding rectangle instead of an MBR. We call it an Extended MBR(EMBR). The EMBR is used only for leaf nodes. With the EMBR, we can reduce unnecessary update cost. The EMBR is larger than the corresponding MBR. Therefore, if we use the EMBR, the search performance can be degraded since the overlap between leaf nodes is increased. In this case, however, the gain of the update performance is bigger than the loss of the search performance. There is a trade-off between the gain of the update performance and the loss of the search performance.

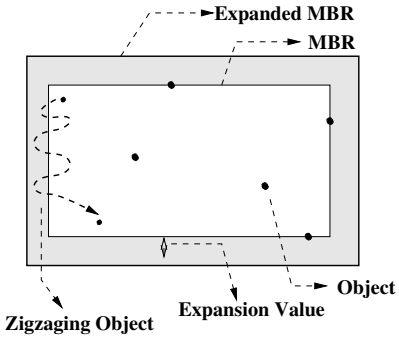


Figure 3. Concept of the Extended MBR

The following is the definition of the EMBR.

Definition 1 (Extended MBR) *Let O be the set of objects which an EMBR includes, and O be defined as $O = \{o_1, o_2, \ldots, o_K\}$. The position of each object is represented by a d-dimensional point as $o_k = (p_k^1, p_k^2, \ldots, p_k^d)$, where $1 \leq k \leq K$. In a d-dimensional space, the EMBR is defined as*

$$EMBR = (EI_1, EI_2, \ldots EI_d)$$

$EI_i (1 \leq i \leq d)$ means an interval of the ith dimension. EI_i is defined as

$$EI_i = [\min_{1 \leq k \leq K} (p_k^i - \epsilon), \max_{1 \leq k \leq K} ((p_k^i + \epsilon))]$$

Where ϵ is the extension value.

The extension value, ϵ is a constant value. If ϵ is zero, the EMBR is equal to the corresponding MBR. In this case, the search performance of the LUR-tree is similar to that of the R-tree. As ϵ becomes larger, we can reduce more update cost. Since, however, overlaps between leaves also become larger, the search performance will be degraded. Therefore, we should choose an appropriate extension value by considering the distribution and the movement pattern of objects.

5 Performance Evaluation

In this section, we will present the results of some experiments to analyze the performance of the LUR-tree with respect to the update performance and the search performance. To verify the effectiveness of the proposed LUR-tree, we compared the LUR-tree with the conventional R*-tree in terms of the number of page accesses. The experimental settings are described first and the results of experiments are given next.

5.1 Experimental Setup

We implemented both the LUR-tree and the R*-tree in C++ on a Linux machine (Redhat 6, kernal version 2.2.14) with Celeron 366MHz CPU, 128MB of memory and 18GB HDD. The size of disk page is set to 4 KB. We did not consider the effect of disk caches in the operating system. We used the Katayama's R*-tree source codes[1]. The R*-tree used the deletion and insertion algorithm for updating the position. The fanout of the R*-tree is 102 for leaf nodes and 113 for internal nodes, and that of the LUR-tree is 101 for leaf nodes and 113 for internal nodes. Owing to the lack of real data, we used synthetic datasets generated by the "General Spatio-Temporal Data" (GSTD)[16], which has been widely adopted as a benchmarking data generator for moving objects (e.g. [10, 13, 14]). Table 1 summarizes the datasets used in experiments. We generated the

[1]The Katayama's R*-tree source codes can be retrieved from `http://research.nii.ac.jp/ katayama/homepage /research/srtree/English.html`

datasets with two initial data distributions and two movement patterns. When we generate the datasets using Gaussian distribution, the standard deviation of dataset is set to 0.1. The movement patterns include random movement and directed movement. Table 2 summarizes the movement patterns setting. Generated datasets consist of from 1,000 to 10,000 objects in a working space, which is the unit square (i.e. $[0, 1]^2$). No objects disappeared and no new objects appeared during each experiment.

Table 1. Datasets used in the experiments

	initial distribution	movement
U-R dataset	uniform	random
U-D dataset	uniform	directed
G-R dataset	Gaussian	random
G-D dataset	Gaussian	directed

Table 2. Setting for two movement patterns

	$\bar{v}$	$\sigma(v)$	working space
Random	0	0.005	$[-0.005, 0.005]^2$
Directed	0.005	0.005	$[0, 0.01]^2$

For the experiments, we used three variants of the LUR-tree varying the extension value. Table 3 describes these variants.

Table 3. Three variants of the LUR-tree

	extension value (ϵ)
LUR-tree(0)	0 (not use the EMBR)
LUR-tree(0.0025)	0.0025
LUR-tree(0.005)	0.005

In all experiments, we measured the number of disk page accesses. Page accesses consist of non-leaf node accesses and leaf node accesses. Since the number of objects was fixed, we could implement the DirectLink using a static hashing technique without overflow. We needed just one disk access to read the DirectLink owing to no overflow. As we mentioned in Section 3, we assume that an update request consists of an id and a new position of an object. To delete the object in the R*-tree, we have to find an old position of the object. We needed at least one disk access to find the old position for the R*-tree. In this reason, we ignored these additional disk accesses for both of the LUR-tree and the R*-tree.

5.2 Update Performance

In the first experiment, we compared the LUR-tree variants with the R*-tree in terms of the numbers of page accesses to process 100 update operations for each object in each dataset. The total number of update operations will be 100 times the number of objects in a dataset.

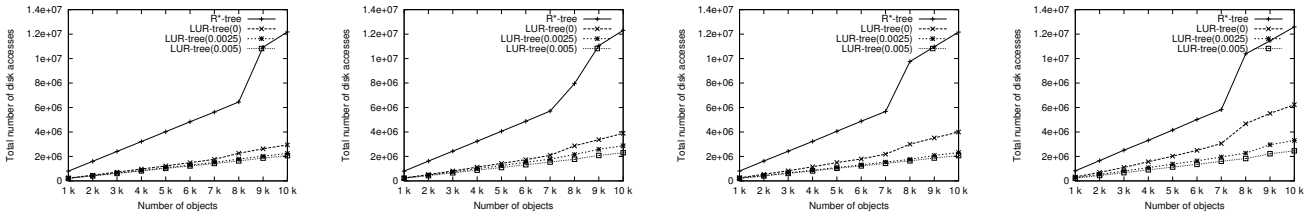
Figure 4 shows the total number of page accesses with varying the number of objects. The trend is similar for all datasets. Clearly, as the number of objects increases, the number of disk accesses grows much faster for the R*-tree than for the LUR-tree variants. The LUR-tree can reduce the update cost for the large number of objects, since it prevents unnecessary traversals and modifications. Between 7,000 and 9,000 objects, the height of the tree grows by one. Due to the growth of the height, the the number of disk accesses for the R*-tree increases rapidly in that range. The performance of the LUR-tree, however, is little affected by the height. The reason is that the LUR-tree directly accesses the leaves using the DirectLink. Among the LUR-tree variants, as the extension value ϵ increases, the number of disk accesses decreases. Figure 5 shows the average number of disk accesses per each update query. The number of disk accesses for each update operation is not affected the number of objects. Between 7,000 and 9,000 objects, we can see that the number of disk accesses for the R*-tree jumps up suddenly. This is also due to the growth of the height of the tree. In the LUR-tree, if we can process all the update queries in the lazy updating way, we need just 2 disk accesses per each query. But, as you see in Figure 5, the average number of disk accesses is bigger than 2. Figure 5(d) shows that the initial distribution and the movement pattern affect the update performance of the LUR-tree. In the highly skewed dataset (G-D dataset), the LUR-tree(0) which does not use the EMBR requires about 2 times more disk accesses than other LUR-tree variants.

5.3 Search Performance

In this experiment, we measured the performance for two categories of search. One is the range search and the other is the k-nearest neighbor search. In both categories of experiments, the number of moving objects is fixed to 10,000. After half the update operations was executed (i.e. 50 update operations per an object), we performed this experiment.

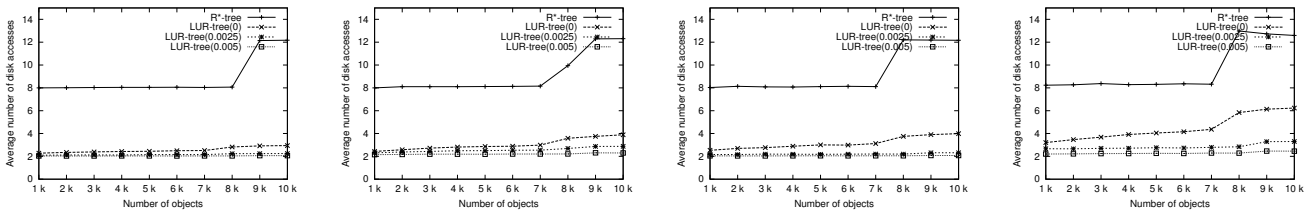
Range Search Query : First, we compared the performance for the range search query with varying the size of a query window. We used 5 sets of square query windows with a range of 0.1%, 1%, 10%, 20%, and 30% of the total range with respect to each dimension, i.e., 0.0001%, 0.01%, 1%, 4%, and 9% of the area of total space. Each query set includes 100 query windows. Query windows are uniformly distributed in the unit square (i.e. $[0, 1]^2$).

Figure 6 shows the average number of disk accesses per each range search query. Note that the x-axis is of logarithmic scale with base 10. The search performance of the LUR-tree is a little worse than that of the R-tree. As the extension value increases, the LUR-tree requires more disk accesses. This is due to that overlaps between the leaves increase. As a result, the number of node accesses increases.



(a) U-R dataset (b) U-D dataset (c) G-R dataset (d) G-D dataset

Figure 4. Total number of disk accesses to process 100 update query for each object



(a) U-R dataset (b) U-D dataset (c) G-R dataset (d) G-D dataset

Figure 5. Average number of disk accesses for a update query

k-Nearest Neighbor Search Query : Next, we compared the performance for the k-nearest neighbor search query with varying the k from 1 to 50. In both the R*-tree and the LUR-tree, we used the same k-nearest search algorithm used in [11]. We generated 100 query points for the experiment in uniform distribution. We processed 100 k-nearest neighbor search queries for each k.

Figure 7 shows the average number of disk accesses per each k-nearest neighbor search query. As the result of range search, the LUR-tree variants require more disk accesses that the R*-tree. But the gap among the R*-tree, the LUR-tree(0) and LUR-tree(0.0025) is small. The R*-tree and the LUR-tree(0) require almost the same disk accesses (The R*-tree requires slightly less). As the extension value increases, the LUR-tree requires more disk accesses.

5.4 Summary

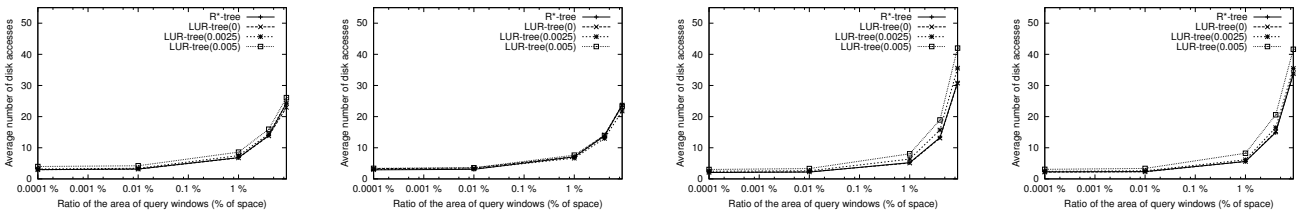
The first observation on all experimental results is that the update performance of the LUR-tree is much better than that of the R*-tree, although the search performance of the LUR-tree is little worse than that of the R*-tree. Above all, the LUR-tree is less affected by the number of objects than the R*-tree does. The relative performance gap between the LUR-tree and the R*-tree increases with an increasing number of moving objects.

Another observation is about the impact of the EMBR. Figure 5(d) shows that the update performance of LUR-tree without EMBR is about 2 times worse than that of LUR-tree with EMBR. This result shows that the EMBR can reduce the update cost efficiently in such cases. As the extension value ϵ increases, the update cost of the LUR-tree decreases but the search cost increases. It shows that there is a trade-off between the gain of the update performance and the loss of the search performance. If the extension value ϵ is too large, the search performance of the LUR-tree is extremely worse in a highly skewed dataset such as Figure 7(d). For this reason, We should choose an appropriate extension value considering the distribution and the movement pattern of objects.

6 Conclusion

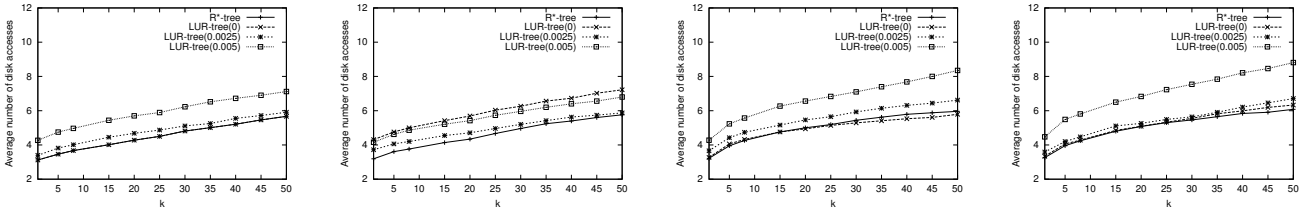
Traditional databases cannot support dynamic updated values such as the positions of continuously moving objects. If we use the R-tree variants for indexing moving objects, we would have to update index structure whenever an update request issues, and this update can cause a node splitting or merging. Another approaches which represent the movements of objects as a simple linear function, are not appropriate for complex movements of objects in many real applications. To solve this problem, we have proposed the novel, R-tree based index structure called LUR-tree. The LUR-tree updates the structure of the index only when an object moves out of the corresponding MBR (minimum bounding rectangle). If the new position of an object is in the MBR, the LUR-tree changes the position of the object in the leaf node. It can update the position of the object quickly and reduce update cost greatly. We also proposed the concept of an Extended MBR (EMBR), which can reduce more update cost. Since it is based on R-tree, the LUR-tree uses the same algorithms to support various query types as the R-tree. We present the experimental results shows that our technique outperforms other techniques.

Future research includes the following. First, we want to apply the lazy update approaches to other spatial index



(a) U-R dataset (b) U-D dataset (c) G-R dataset (d) G-D dataset

Figure 6. Average number of disk accesses for a range search query



(a) U-R dataset (b) U-D dataset (c) G-R dataset (d) G-D dataset

Figure 7. Average number of disk accesses for a k-nearest neighbor search query

structures (e.g. the quad tree, the K-D-B tree). We will also study a hybrid technique which will combine our lazy update technique with existing approaches based on a linear function. Finally, we plan to find a new algorithm for indexing historical movements such as trajectories of moving objects.

Acknowledgements The authors would like to thank Bongki Moon, the assistant professor in the Department of Computer Science at the University of Arizona, for many helpful advice and assistance.

References

[1] T. Abraham and J. F. Roddick. Survey of spatio-temporal databases. *GeoInformatica*, 3(1):61–99, 1999.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *In Proc. of the 19th ACM Symp. on Principles of Database Systems*, pages 175–186, 2000.

[3] D. Barbará. Mobile computing and databases–a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *In Proc. of the 1990 ACM SIGMOD Int'l. Conf. on Management of Data*, pages 322–331, 1990.

[6] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *In Proc. of the 1984 ACM SIGMOD Int'l. Conf. on Management of Data*, pages 47–57, 1984.

[8] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *In Proc. of the 18th ACM Symp. on Principles of Database Systems*, pages 261–272, 1999.

[9] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *In Proc. of the 6th Int'l. Symp. on Spatial Databases*, pages 111–132, 1999.

[10] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *In Proc. of 26th Int'l. Conf. on Very Large Data Bases*, pages 395–406, 2000.

[11] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *In Proc. of the 1995 ACM SIGMOD Int'l. Conf. on Management of Data*, pages 71–79, 1995.

[12] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *In Proc. of the 2000 ACM SIGMOD Int'l. Conf. on Management of Data*, pages 331–342, 2000.

[13] Z. Song and N. Roussopoulos. Hashing moving objects. In *In Proc. of the 2nd Int'l. Conf. on Mobile Data Management*, pages 161–172, 2001.

[14] Y. Tao and D. Papadias. MV3R-tree: a spatio-temporal access method for timestamp and interval queries. In *In Proc. of 27th Int'l. Conf. on Very Large Data Bases*, 2001, to appear.

[15] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[16] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In *In Proc. of the 6th Int'l. Symp. on Spatial Databases*, pages 147–164, 1999.

[17] O. Wolfson, B. Xu, S. Chamberlaina, and L. Jiang. Moving objects databases: Issues and solutions. In *In Proc. of 10th Int'l. Conf. on Scientific and Statistical Database Management*, pages 111–122, 1998.