

Short Paper

Efficient Update Method for Indexing Locations of Moving Objects*

DONGSEOP KWON, SANGJUN LEE⁺ AND SUKHO LEE

School of Electrical Engineering and Computer Science

Seoul National University

Seoul 151-742, Korea

⁺*School of Computing*

Soongsil University

Seoul 156-743, Korea

Tracking the changing positions of moving objects is becoming increasingly feasible and necessary. However, traditional spatial index structures are not suitable for storing these positions because of numerous update operations. In this paper, we propose an efficient update method for indexing the locations of moving objects based on the R-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 the leaf node only. Using a secondary access path to find the corresponding leaf node, the proposed technique can update the position of the object quickly and reduce update the cost greatly. In addition, our technique can be adopted in diverse variants of the R-tree and various applications that use the R-tree since it is based on the R-tree and it guarantees the correctness of the R-tree. We also present experimental results which show that our technique outperforms other techniques.

Keywords: indexing, access method, data structures, database, moving object, R-tree

1. INTRODUCTION

With advances in positioning systems, such as the Global Positioning System (GPS), and progress in mobile computing environments, tracking the changing positions of moving objects is becoming increasingly feasible and necessary. The need to store and process continuously moving data arises in a wide range of applications, including traffic control or monitoring, transportation and supply chain management, digital battlefields, and mobile e-commerce [1]. However, traditional database management systems assume that data stored in a database remain constant unless they are explicitly modified. This static model is appropriate when data change in discrete steps. However, it is not appropriate for representing, storing, and querying dynamic attributes, such as the positions of

Received October 11, 2002; revised June 29 and October 13, 2004; accepted October 27, 2004.

Communicated by Ming-Syan Chen.

* This work was supported in part by the Brain Korea 21 Project and in part by the Information Technology Research Center (ITRC) Support Program in 2004.

continuously moving objects, because the database has to be updated continuously. This is inefficient and infeasible due to the resulting large update overhead. This overhead becomes more serious as the number of moving objects increases. The R-tree [2], which is one of index structures most widely used to index multidimensional data in traditional spatial databases, also has serious problems with indexing the positions of moving objects. A modification in the leaf node may split or merge the nodes. This requires additional overhead. In the worst case, frequent index rebuilding may be required. Therefore, the standard R-tree is not suitable for indexing dynamic attributes, such as the positions of continuously moving objects.

In this paper, to solve these problems, we propose a new indexing technique, the Leaf-prior Update R-tree (LUR-tree), which efficiently indexes the current positions of moving objects and reduces the update cost greatly. In this technique, we remove unnecessary modification of the tree while updating the positions. This technique updates the structure of the LUR-tree 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, the LUR-tree changes only the position of the object in the leaf node. To find the leaf node to which an object belongs, we use another access method called DirectLink. It is a secondary index used to find the corresponding leaf node from an object's id. With DirectLink, the position of the object can be found quickly. The main goal of this paper is to improve the R-tree index structure for updating dynamic attributes, such as the positions of moving objects, because the R-tree is known to be efficient and is generally used in this area [3]. Since the LUR-tree naturally extends the R-tree, it uses the same algorithms to process various types of queries (e.g., range queries, k-nearest queries, and spatial join queries) as does the R-tree. We have implemented the LUR-tree based on R*-tree [4] and carried out experiments in various environments.

The rest of this paper is organized as follows: In section 2, we review related works 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 the update cost. In section 5, we present experimental results to compare our technique with existing approaches. Finally, conclusions and future work are discussed in section 6.

2. RELATED WORKS

Various index structures have been proposed for multidimensional data, and a survey for these can be found in [5]. However, these multidimensional index structures cannot be used directly to index moving objects because it is hard for these index structures to handle heavy update loads efficiently.

In recent times, many new index structures have been proposed to overcome the drawbacks of spatial indexing techniques. Depending on the type of data being stored, the following two categories types of structures exist: (a) those that index the past positions of objects (that is trajectories) and (b) those that index the current positions of objects. Our technique belongs to the latter category.

With structures 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 adding time into the same space. In [6], the authors introduced two access methods called the Spa-

tio-Temporal R-tree (STR-tree) and Trajectory-Bundle tree (TB-tree). They claimed that these two tree structures worked better than the traditional R-tree family of structures for trajectory-based queries. Recently, Tao *et al.* [7] proposed the Multi-version 3D R-tree (MV3R-tree), a structure that combines the concepts of multi-version B-trees [8] and 3D-Rtrees.

With structures in the latter category, most of the existing approaches abstract 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.* [9] addressed the issue of indexing moving objects to query their present and future positions. They proposed a method to index moving objects using the PMR-Quadtree. Kollis *et al.* [10] proposed an efficient indexing scheme based on partition trees. They used a linear function to describe the trajectories of objects. Since it was very hard to index a line in most spatial databases, the authors mapped a line into a point in the dual plane. The duality transformation formulated the problem in a more intuitive manner. In dual space, however, a rectangle query range becomes a polygon. For this reason, the query becomes more difficult to execute. Agarwal *et al.* [11] proposed various efficient schemes based on this duality. They also developed an efficient indexing scheme for answering 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 is represented by a reference position and a corresponding velocity vector. When nodes are split, the TPR-tree considers not only the positions of the moving points but also their velocity. The problem with all these techniques is that a good function for describing movements doesn't exist. In many real applications, the movements of objects are complicated, not linear. In this situation, approaches based on a linear function cannot reduce the update cost efficiently.

As an alternative way to solve this problem, Song and Roussopoulos [13] proposed a hash-based indexing method. This approach is simple and intuitive. It can effectively reduce the update cost. Since, however, they use a simple static hashing scheme, their approach suffers from the overflow problem. If the distribution of data is skewed, this problem could become more serious.

3. PROBLEM DESCRIPTION

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

3.1 Application Scenario

The optimization of transportation, especially in highly populated areas, is a very challenging task that can be supported by an information system. Consider the fleet management system. At the central site, there is a database system for managing all the data of vehicles. Each vehicle is equipped with a GPS device. Vehicles measure their current locations with the GPS devices and transmit their positions to the central server using either radio communication links or cellular phones. The server stores the current posi-

tions of all the vehicles and processes queries. Example queries occurring in such an application are as follows:

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

3.2 Problem Setting

The problem we will focus on here is that of reducing the enormous update cost involved in indexing moving objects using a multidimensional index structure. Suppose that there are N moving objects in a d -dimensional space. The position of the i th object is given by $o_i = (p_1, p_2, \dots, p_d)$, 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 location sampling). 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 [14]. 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 that there are no errors in the network layer. We also ignore the uncertainties in the network. We ignore all the other uncertainties under a similar assumption.

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

4. LEAF-PRIOR UPDATE R-TREE

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

4.1 Basic Idea

As stated in section 2, using a spatial index, such as the R-tree, to index the positions of continuously moving objects leads to some problems. Consider the situation shown in Fig. 1. We use the R-tree to index 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 that the maximum cardinality of entries in the leaf node is 5. $R1$ has 5

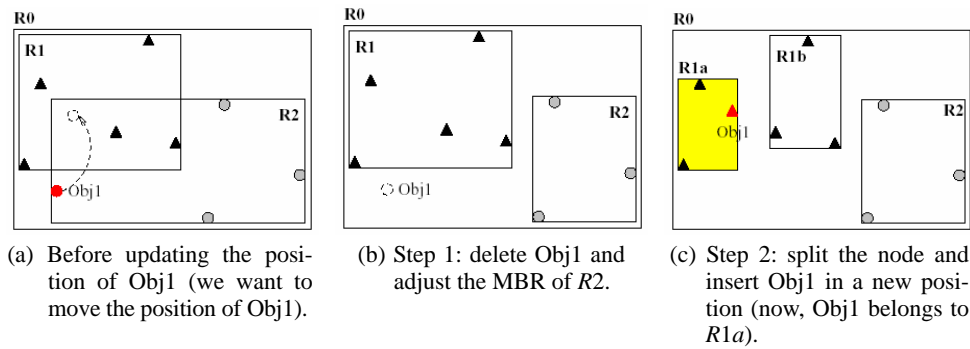


Fig. 1. Example of updating in the traditional R-tree.

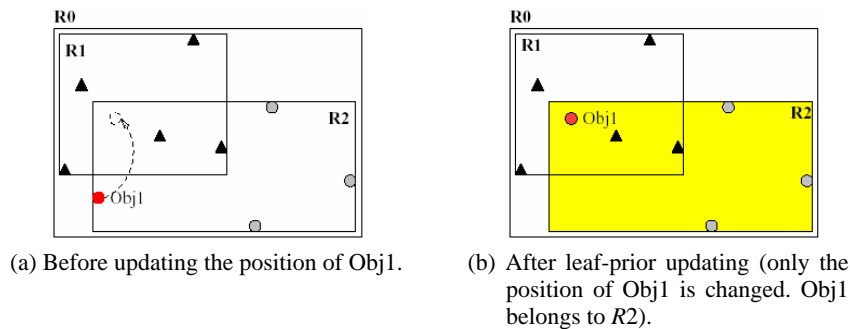


Fig. 2. Example of updating in the leaf-prior update approach.

entries, and $R2$ has 4 (including $Obj1$). Triangles indicate objects which belong to $R1$, and circles indicate those which belong to $R2$. The situation is as follows. $Obj1$ reports that its location has moved to a new position. Thus, we want to update the position of $Obj1$ as shown in Fig. 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 lead to some serious problems: the splitting or merging of nodes. As in Fig. 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. There is, however, no room for a new entry in $R1$. Thus, we have to split $R1$ into $R1a$ and $R1b$ as shown in Fig. 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 events can occur.

We, however, do not have to update as in Fig. 1. If a new position of an object is in the MBR to which the object belongs (as in Fig. 2 (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 as in Fig. 2 (b). Fig. 2 (b) 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 eliminate 4 disk accesses. Since the method based on deletion and insertion requires ad-

ditional disk accesses for traversing and adjusting the tree, we also can reduce the number of disk accesses. In Fig. 2 (b), the MBR of $R2$ is no longer a minimum bounding rectangle. Thus, we have to adjust the MBR of $R2$ in the same manner as in 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 the regions of its child nodes.

4.2 Index Structure

Based on the basic idea described above, we propose a new index structure, the leaf-prior 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 leaf-prior update algorithm is appended. Algorithms and data structure in the R-tree are slightly modified.

An update request basically consists of a pair composed of an object id and a new position. An example is “change the position of a car whose id is ‘47’ to a new position (150, 240).” However, we cannot perform the update operation directly for this request. In the case of the original R-tree, we need to know the old position of the object in order to delete it from the tree. Therefore, in addition to the R-tree, we need a way to find the previous position of the object by means of an id, such as a hash table. Similarly, in the case of the LUR-tree, we cannot find the object directly in the LUR-tree because the key in the LUR-tree is a position, not an object id. Therefore, we should scan all the nodes in the tree to find the corresponding item.

To avoid this situation and find the leaf node quickly, 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. 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 a hash index or B-tree index, can be used for the DirectLink. Fig. 3 illustrates the structure of the LUR-tree and the DirectLink. Although the DirectLink has an additional space overhead, it can reduce the update cost considerably because it helps the system to access a node without performing tree traversals.

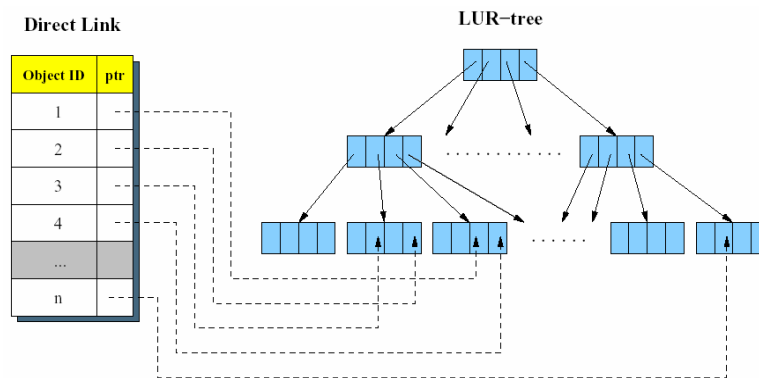


Fig. 3. Structure of the LUR-tree and the DirectLink.

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 the entries of the node.

4.3 Leaf-Prior Update Algorithm

Algorithm 1 describes the algorithm of leaf-prior 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 leaf-prior update is possible; i.e., whether 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. Leaf-prior Update Algorithm

Procedure LeafPriorUpdate (*oid*, *newpos*)

Input: an object's id *oid*, new position *newpos*

begin

1. *blockid* \leftarrow ReadDirectLink(*oid*);
2. *N* \leftarrow ReadNode(*blockid*);
3. *E* \leftarrow GetEntry(*N*, *oid*);
4. **if** *newpos* \in *N*.*MBR* **then**
5. *E*.*pos* \leftarrow *newpos*;
6. WriteNode(*N*, *blockid*);
7. **else**
8. Delete(*oid*, *E*.*pos*);
9. *new_blockid* \leftarrow Insert(*oid*, *newpos*);
10. UpdateDirectLink(*oid*, *new_blockid*);

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 the simplest way and has been used in the original R-tree. It is expensive because it may cause splitting or merging.
- Dynamic extension of the MBR: Instead of deleting the object, this method extends the MBR. However, this method may lead to additional cost for adjusting all the MBRs of its parent nodes.
- Reinsertion into the parent node: Instead of inserting the new position from the root, we can reinsert it from the parent node to reduce the insertion cost.

For simplicity, we use 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 as shown in Fig. 4, deletions and insertions can occur continuously. This can affect the update performance. To prevent this problem, we can use a slightly larger bounding rectangle instead of an MBR. We call it the Extended MBR (EMBR). The EMBR is used only for leaf nodes. With the EMBR, we can reduce any unnecessary update cost. The EMBR is larger than the corresponding MBR. Therefore, if we use the EMBR, the search performance may be degraded since the overlap between leaf nodes is increased. In this case, however, the gain in update performance is greater than the loss in search performance. There is a trade-off between the gain in update performance and the loss in search performance. The EMBR is defined as follows:

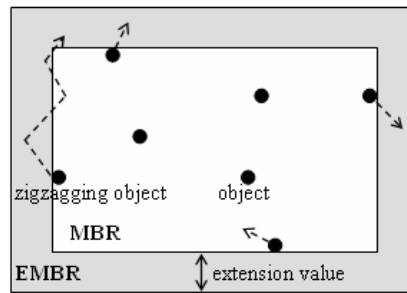


Fig. 4. Concept of the Extended MBR.

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

$$EMBR = (EI_1, EI_2, \dots, EI_d). \quad (3)$$

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

$$EI_i = \left[\min_{1 \leq k \leq K} (p_k^i - \varepsilon), \max_{1 \leq k \leq K} (p_k^i + \varepsilon) \right], \quad (4)$$

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 the update cost more. However, overlaps between leaves also become larger. Thus, the search performance is also degraded. Therefore, we should choose an appropriate extension value by considering the

distribution and the movement pattern of objects. Note that the update operation does not enlarge the size of the EMBR during execution. When an object moves out of the EMBR, the LUR-tree invokes the original insert and delete algorithm of the R-tree instead of adjusting the EMBR to keep the object.

5. PERFORMANCE EVALUATION

In this section, we will present the results of some experiments performed 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 will be described first, and the results of experiments will be given next.

5.1 Experimental Setup

We implemented both the LUR-tree and the R*-tree in C++ on a Linux machine (Redhat 7, kernel version 2.4.2-2), a Pentium III 1GHz CPU with 768MB of memory and a 40GB HDD. The size of the disk page was set to be 4KB. We did not consider the effect of disk caches in the operating system. We used Katayama's R*-tree source [15]. We modified it in order to implement the LUR-tree. The R*-tree used the deletion and insertion algorithm to update positions. In our implementation of the R*-tree and the LUR-tree, the fanout of the R*-tree was 102 for leaf nodes and 113 for internal nodes, and that of the LUR-tree was 101 for leaf nodes and 113 for internal nodes. Owing to the lack of real data, we used synthetic datasets generated by "General Spatio-Temporal Data" (GSTD) [16]. GSTD has been widely adopted as a benchmarking data generator for moving objects (e.g., [6, 7, 13]). We generated test datasets with two initial data distributions and two movement patterns. Generated datasets consisted of from 1,000 to 10,000 objects in a working space, which was the unit square (i.e., $[0, 1]^2$). No objects disappeared, and no new objects appeared during each experiment. Initial data distributions included uniform distribution and Gaussian distribution. Objects were uniformly distributed in the working space in the case of uniform distribution and clustered around the center of the working space in the case of Gaussian distribution. When we generated the datasets using Gaussian distribution, the standard deviation of the dataset was set to be 0.1. The movement patterns included random movement and directed movement. Table 1 summarizes the movement pattern settings used in the experiments. v is the velocity of an object, and \bar{v} is the average velocity of all the objects. $\sigma(v)$ is the standard variation of v . At each time slice, the objects moved according to a given movement pattern. Table 2 summarizes the abbreviations for datasets used in the experiments. In the experiments, we used three variants of the LUR-tree, obtained by varying the extension value. Table 3 describes these variants. Note that the value 0.005 in the LUR-tree(0.005) is the average velocity, \bar{v} , of the directed movement pattern.

In all the 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

Table 1. Settings for two movement patterns.

	\bar{v}	$\sigma(v)$	$\min_x(v)$	$\min_y(v)$	$\max_x(v)$	$\max_y(v)$
Random movement	0	0.005	-0.005	-0.005	0.005	0.005
Directed movement	0.005	0.005	0	0	0.01	0.01

Table 2. Abbreviations for datasets.

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

Table 3. Three variants of the LUR-tree.

LUR-tree(0)	LUR-tree whose extension value, ϵ , is 0 (i.e., does not use the EMBR)
LUR-tree(0.0025)	LUR-tree whose extension value, ϵ , is 0.0025
LUR-tree(0.005)	LUR-tree whose extension value, ϵ , is 0.005

without overflow. We needed just one disk access to read the DirectLink owing to the absence of overflow. As we mentioned in section 3, we assume that an update request consisted of an id and a new position of an object. To delete the object in the R^* -tree, we had to find an old position of the object. We needed at least one disk access to find the old position in the R^* -tree. For 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 number of page accesses needed to process 100 update operations for each object in each dataset. Thus, the total number of update operations was 100 times the number of objects in a dataset.

Fig. 5 shows the total number of page accesses with variation of the number of objects. The trend was similar for all the datasets. Clearly, as the number of objects increased, the number of disk accesses grew much faster for the R^* -tree than for the LUR-tree variants. The LUR-tree could reduce the update cost for a large number of objects since it prevented unnecessary traversals and modifications. When the number of objects was between 7,000 and 9,000, the height of the tree grew by one. Due to the increase in height, the number of disk accesses for the R^* -tree increased rapidly in that range. The performance of the LUR-tree, however, was little affected by the height. The reason was that the LUR-tree directly accessed the leaves using the DirectLink. Among the LUR-tree variants, as the extension value, ϵ , increased, the number of disk accesses decreased.

Fig. 6 shows the average number of disk accesses for each update query. The number of disk accesses for each update operation was not affected by the number of objects.

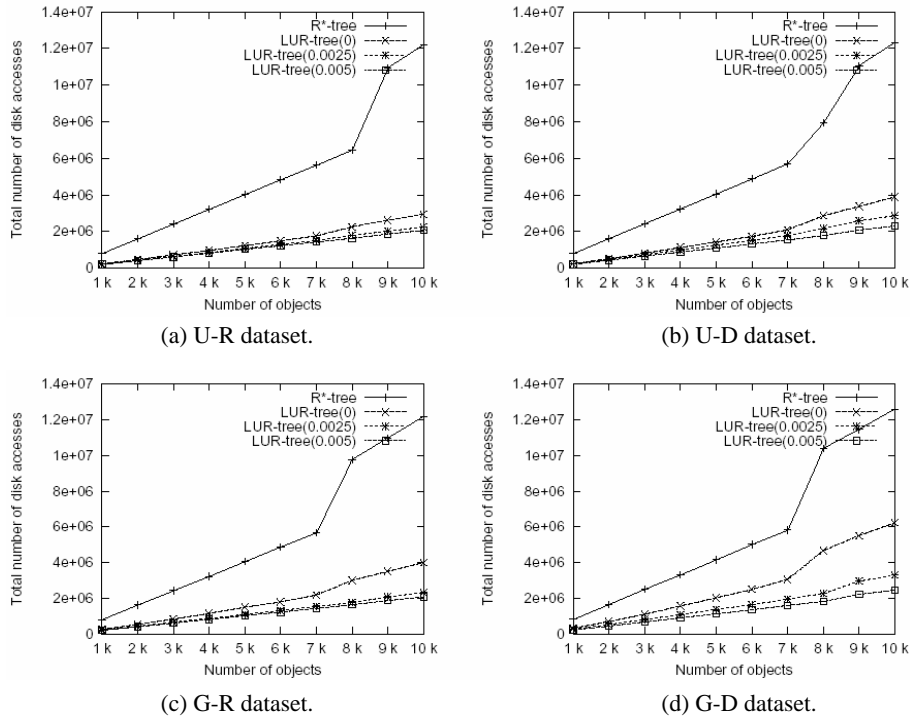


Fig. 5. Total number of disk accesses needed to process 100 update queries for each object.

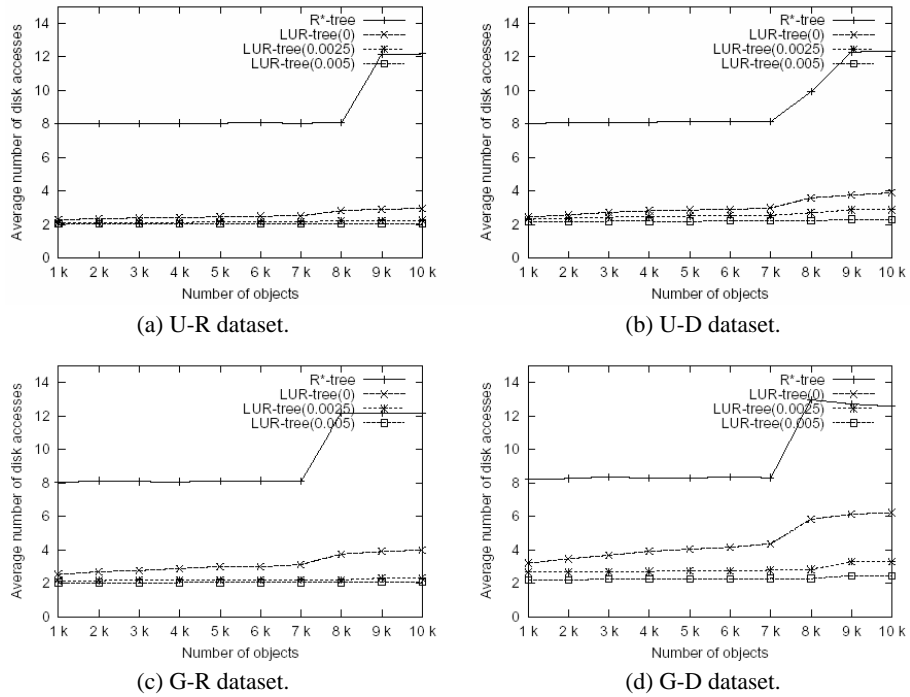


Fig. 6. Average number of disk accesses for an update query.

When the number of objects was between 7,000 and 9,000, the number of disk accesses for the R^* -tree jumped suddenly. This was also due to the growth of the height of the tree. In the LUR-tree, if we can process all the update queries through leaf-prior updating, we need just 2 disk accesses for each query. However, as shown in Fig. 6, the average number of disk accesses was larger than 2. Fig. 6 (d) shows that the initial distribution and the movement pattern affected 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, required about 2 times more disk accesses than the other LUR-tree variants.

5.3 Search Performance

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

Range Search Query: First, we compared the performance for range search queries with variation of the size of a query window. We used 5 sets of square query windows with ranges 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 total space. Each query set included 100 query windows. Query windows were uniformly distributed in the unit square (i.e., $[0, 1]^2$). Fig. 7 shows the average number of disk accesses for each range search query. Note that the x-axis is of logarithmic scale with base 10. The search performance of the LUR-tree was a little worse than that of the R^* -tree. As the extension value increased, the LUR-tree required more disk accesses. This was due to the fact that the number of overlaps between the leaves increased. As a result, the number of node accesses increased. However, compared with the gain in update performance, the loss in search performance was minor.

k -Nearest Neighbor Search Query: Next, we compared the performance for k -nearest neighbor search queries with variation of 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 [17]. We generated 100 query points in the experiment with uniform distribution. We processed 100 k -nearest neighbor search queries for each k . Fig. 8 shows the average number of disk accesses for each k -nearest neighbor search query. As the results of range search show, the LUR-tree variants required more disk accesses than the R^* -tree. But the differences among the R^* -tree, the LUR-tree(0) and LUR-tree(0.0025) were small. The R^* -tree and the LUR-tree(0) required almost the same number of disk accesses (the R^* -tree required slightly fewer). As the extension value increased, the LUR-tree required more disk accesses.

5.4 Summary

The first observation from all the 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 a little worse than that of the R^* -tree. Above all, the

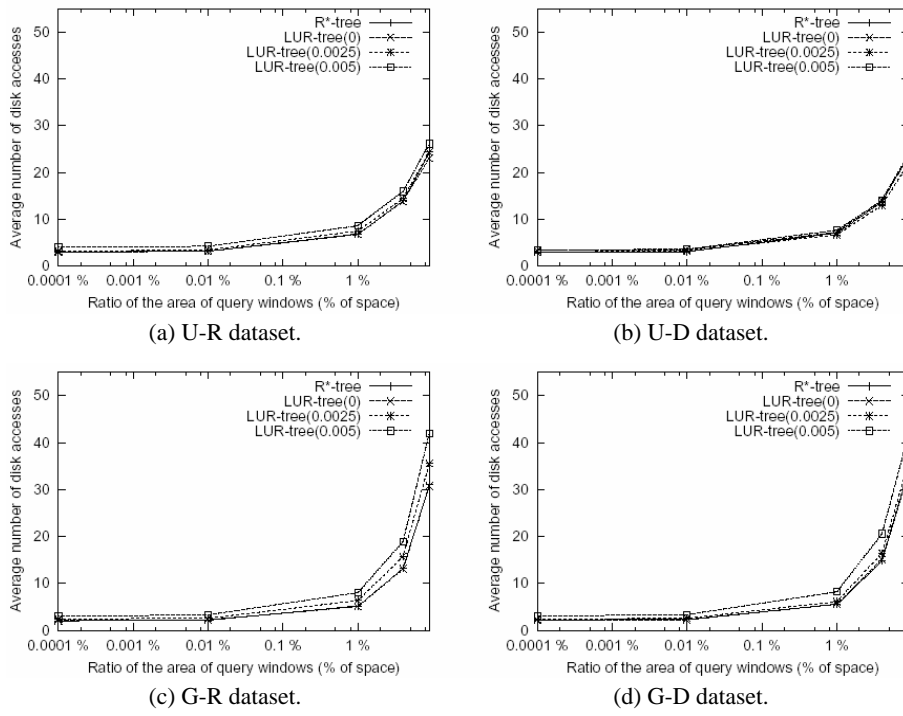


Fig. 7. Average number of disk accesses for a range search query.

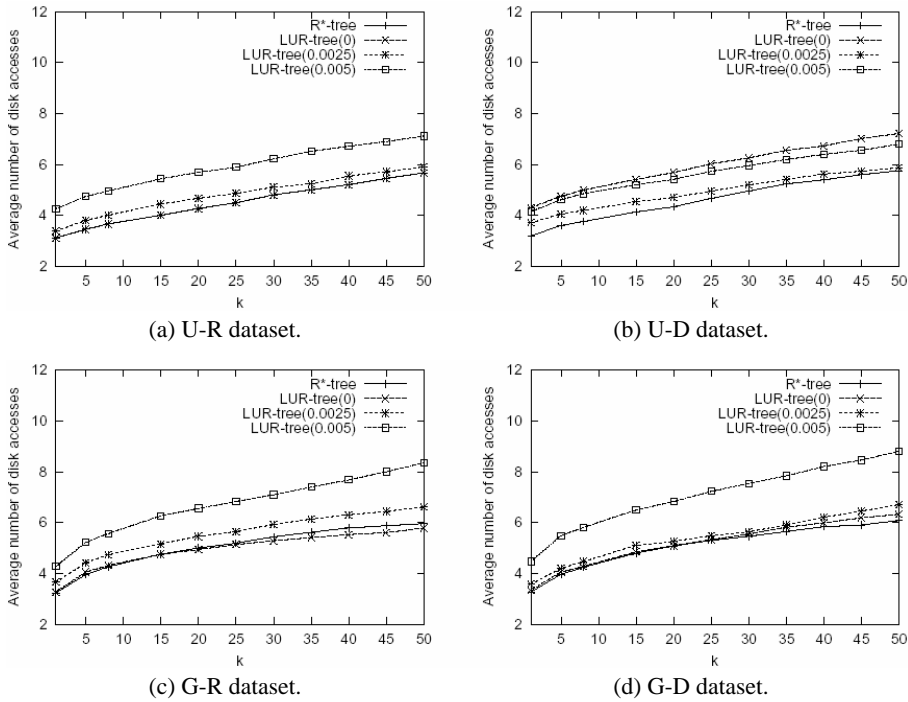


Fig. 8. Average number of disk accesses for a k -nearest neighbor search query.

LUR-tree is less affected by the number of objects than the R^* -tree is. The relative performance difference between the LUR-tree and the R^* -tree increases as the number of moving objects increases.

Another observation concerns the impact of the EMBR. Fig. 6 (d) shows that the update performance of the LUR-tree without the EMBR is about 2 times worse than that of the LUR-tree with the 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. This shows that there is a trade-off between the gain in update performance and the loss in search performance. If the extension value, ε , is too large, the search performance of the LUR-tree is much worse for a highly skewed dataset as shown in Fig. 7 (d). For this reason, we should choose an appropriate extension value based on consideration of the distribution and the movement pattern of objects.

6. CONCLUSIONS

Traditional databases cannot support dynamic updated values, such as the positions of continuously moving objects. Although the R-tree is most widely used for multidimensional data, it is not efficient for indexing dynamic values because updating can cause a node to split or merge. To solve this problem, we have proposed a novel R-tree based index structure called the Leaf-prior Update R-tree (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. Using the DirectLink, it can update the position of the object quickly and reduce the update cost greatly. We have also proposed the Extended MBR (EMBR), which can reduce the update cost even more. Since it is based on R-tree, the LUR-tree uses the same algorithms for various query types as the R-tree does. We have presented experimental results showing that our technique outperforms other techniques.

Future research will include the following. First, we want to apply the leaf-prior update approaches to other spatial index structures (e.g., the quad tree and the K-D-B tree). We will also study a hybrid technique which will combine our leaf-prior update technique with existing approaches based on a linear function. Finally, we plan to find a new algorithm for indexing historical movements, such as the trajectories of moving objects.

REFERENCES

1. O. Wolfson, B. Xu, S. Chamberlaina, and L. Jiang, "Moving objects databases: issues and solutions," in *Proceedings of 10th International Conference on Scientific and Statistical Database Management*, 1998, pp. 111-122.
2. A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1984, pp. 47-57.
3. K. V. R. Kanth, S. Ravada, and D. Abugov, "Quadtree and R-tree indexes in oracle

- spatial: a comparison using GIS data,” in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2002, pp. 546-557.
4. N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, “The R*-tree: an efficient and robust access method for points and rectangles,” in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1990, pp. 322-331.
 5. T. Abraham and J. F. Roddick, “Survey of spatio-temporal databases,” *GeoInformatica*, Vol. 3, 1999, pp. 61-99.
 6. D. Pfoser, C. S. Jensen, and Y. Theodoridis, “Novel approaches in query processing for moving object trajectories,” in *Proceedings of 26th International Conference on Very Large Data Bases*, 2000, pp. 395-406.
 7. Y. Tao and D. Papadias, “MV3R-tree: a spatio-temporal access method for timestamp and interval queries,” in *Proceedings of 27th International Conference on Very Large Data Bases*, 2001, pp. 431-440.
 8. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, “An asymptotically optimal multiversion b-tree,” *The International Journal on Very Large Data Bases*, Vol. 5, 1996, pp. 264-275.
 9. J. Tayeb, Ö. Ulusoy, and O. Wolfson, “A quadtree-based dynamic attribute indexing method,” *The Computer Journal*, Vol. 41, 1998, pp. 185-200.
 10. G. Kollios, D. Gunopulos, and V. J. Tsotras, “On indexing mobile objects,” in *Proceedings of 18th ACM Symposium on Principles of Database Systems*, 1999, pp. 261-272.
 11. P. K. Agarwal, L. Arge, and J. Erickson, “Indexing moving points,” in *Proceedings of 19th ACM Symposium on Principles of Database Systems*, 2000, pp. 175-186.
 12. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, “Indexing the positions of continuously moving objects,” in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2000, pp. 331-342.
 13. Z. Song and N. Roussopoulos, “Hashing moving objects,” in *Proceedings of 2nd International Conference on Mobile Data Management*, 2001, pp. 161-172.
 14. D. Pfoser and C. S. Jensen, “Capturing the uncertainty of moving-object representations,” in *Proceedings of 6th International Symposium on Spatial Databases*, 1999, pp. 111-132.
 15. <http://research.nii.ac.jp/~katayama/homepage/research/srtree/English.html>.
 16. Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento, “On the generation of spatio-temporal datasets,” in *Proceedings of 6th International Symposium on Spatial Databases*, 1999, pp. 147-164.
 17. N. Roussopoulos, S. Kelley, and F. Vincent, “Nearest neighbor queries,” in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1995, pp. 71-79.

Dongseop Kwon is a Ph.D. candidate in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea. He received his M.S. and B.S. degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1998 and 2000, respectively. His current research interests include spatio-temporal databases, high dimensional index structures, mobile data managements, and time series databases.

Sangjun Lee received his M.S. and B.S. degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1996 and 1998, respectively and Ph.D. in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea in 2004. He is currently a full-time lecturer of the School of Computing, Soongsil University, Seoul, Korea. His current research interests include high dimensional index structures, mobile data managements, and multimedia databases.

Sukho Lee received his B.A. degree in Political Science and Diplomacy from Yonsei University, Seoul, Korea, in 1964 and his M.S. and Ph.D. in Computer Sciences from the University of Texas at Austin in 1975 and 1979, respectively. He is currently a professor of the School of Computer Science and Engineering, Seoul National University, Seoul, Korea, where he has been leading the Database Research Laboratory. He served as the president of Korea Information Science Society in 1994. His current research interests include database management systems, spatial database systems, and multimedia database systems.