An Efficient Location Index for the Semantic Search of Moving Objects

Dong-Oh Kim, Jung-Su Shin, Hong-Koo Kang, Ki-Joon Han

School of Computer Science & Engineering, Konkuk University,
1, Hwayang-Dong, Gwangjin-Gu, Seoul 143-701, Korea
{dokim, jssinn, hkkang, kjhan}@db.konkuk.ac.kr

Abstract. In moving object databases, researches on the spatio-temporal access method are very important for the efficient search of moving object location in ITS, LBS, and Telematics. Recently, researches are being made actively on the efficient management of the current location of moving objects and on the estimation of future location using information such as the current location and moving pattern of moving objects. In this paper, we propose Map-Based R-tree(MBR-tree), which is a new current location index structure for indexing the current location of moving objects in an urban area, a 2-dimentional space. MBR-tree is an index which forms the MBR(Minimum Bounding Rectangle) of R-tree nodes using static objects(or fixed objects) on the map. Because moving objects generally moves within a static object, if the MBR is formed using static objects, we can reduce the cost of updating the index of the current location of moving objects. In addition, it shows superior performance in semantic search that searches in a specific building or place (e.g. “Who are in Konkuk university?”) rather than in an arbitrary area. Finally, to test the index proposed in this paper, we compared its performance with that of hashing technique and Lazy Update R-tree using various datasets and proved the superiority of its performance.

Keywords: Location Index, Semantic Search, Moving Object, MBR, MBR-tree

1 Introduction

With the development of location positioning systems such as GPS(Global Positioning System), application systems using the location of moving objects are widely used including ITS(Intelligent Transportation System), LBS(Location Based Service) and Telematics. In addition, a moving object database has emerged to manage the location of moving objects efficiently in application systems. In particular, researches are being made actively on the spatio-temporal access method for the efficient search of moving object location[2,4].

In general, spatio-temporal access methods are divided into past location index, current location index and future location index according to the type of query. Recently, hashing technique[6], Lazy Update R-tree(LUR-tree)[1], etc. have been proposed for the efficient management of the current location of moving objects. However, hashing technique, though its update cost is low, has low search performance due to node chaining that takes place on overflow. LUR-tree improves the update performance of R-tree, which is superior in the search performance, but still has high update load from node reconstruction.

Thus, we propose Map-Based R-tree(MBR-tree), which is a new current location indexing technique for indexing the current location of moving objects(e.g. persons) in an urban area, a 2-dimensional space. MBR-tree is an index that forms the MBR(Minimal Bounding Rectangle) of R-tree[3] nodes using static objects(e.g. buildings) on the map. Because moving objects generally move within a static object, if the MBR is formed using static objects, we can reduce the cost of index update for the current location of moving objects.

In addition, MBR-tree shows superior performance in semantic search that searches in a specific building or place(e.g. "Who are in Konkuk University?") rather than in an arbitrary area. Lastly, to test the index proposed in this paper, we compare its performance with that of hashing technique and LUR-tree using various datasets[7]. According to the results, MBR-tree is superior in the search and update performance and particularly excellent in semantic search.

The structure of this paper is as follows. Chapter 2 reviews related works, examining hashing technique and LUR-tree as well as semantic search. Chapter 3 explains MBR-tree in detail, and Chapter 4 analyzes the results of performance evaluation using various datasets. Finally, Chapter 5 draws conclusions.

2 Related Works

This chapter reviews hashing technique and LUR-tree, whose performance will be compared with that of MBR-tree. In addition, it examines semantic search.

2.1 Hashing Technique

In order to reduce the cost of update, hashing technique uses a location pre-processing module that plays the role of a filter between a database and moving objects reporting their locations[6]. The location pre-processing module synchronizes its own hashing function with the hashing function used in the database and stores the location of moving objects into the database using bucket information obtained by entering the location of moving objects into the hashing function. When moving object location is updated, if new bucket information obtained using the hashing function is the same as existing bucket information, the new information is not reported to the database but is recorded only in the location pre-processing module.

Hashing functions used in hashing technique are composed of an overlap-free space partition function that removes redundant hashing nodes while maintaining constant the number of moving objects managed in the bucket, an augmented space partition function that allows the overlapping of hashing nodes and expands hash nodes to the specified size, a quad-tree hashing function that utilizes quad-tree division to resolve the uneven distribution of moving objects, etc.

Hashing technique improves the scalability and the performance of update because it is possible to do distributed processing of moving objects using multiple location pre-processing modules, but if the number of moving objects managed in a bucket is large, the search performance is lowered due to node chaining on overflow.

2.2 LUR-tree

LUR-tree is an index that can reduce update cost by improving an update algorithm and, as a result, reducing the number of times of index reconstruction in R-tree[1]. LUR-tree is composed of R-tree for indexing the current location of moving objects and Direct Link for direct reference to leaf nodes of the index where moving objects are stored. Direct Link, which is an auxiliary index that uses the ID of moving objects as the key, refers to the leaf node in R-tree where the moving object of the corresponding ID is stored. Therefore, it effectively reduces the cost of tree search caused by update of R-tree.

In addition, LUR-tree reduces the cost of updating moving objects that travel zigzag using the extended MBR, which extended the MBR value of index nodes. When updating the location of a moving object, LUR-tree can directly refer to the index node containing the corresponding object using Direct Link. Thus, if the new location of the moving object is in the extended MBR of the current extended node, R-tree is not reconstructed and only information in the node is changed and, in this way, the cost of update can be reduced. LUR-tree has lower update cost than R-tree but its search performance is lowered by node redundancy and its update load increases due to node reconstruction.

2.3 Semantic Search

A semantic space can correspond to a physical space expressed with one or more coordinates, and the expression of the semantic space is easily understandable to users. That is, it has a logical name like “Konkuk University” or “National Road No. 13” corresponding to a physical space expressed with coordinates like “15,13,18,17”. The semantic space is often used as a search keyword in the database[5]. Figure 1 shows examples of correspondence between physical spaces and semantic spaces.

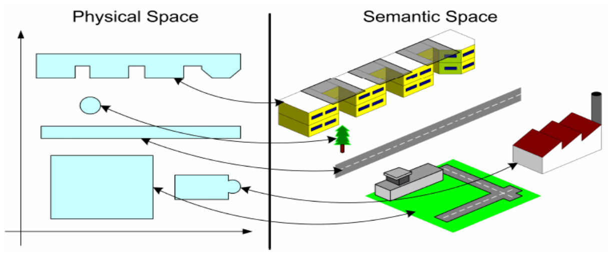

Fig. 1. Correspondence between Physical Spaces and Semantic Spaces

Semantic search is generally used to execute a query which contains semantic spaces in the query condition. Examples of query for semantic search are “Who are in Konkuk University?”, “What cars are on the road where car K is running?”, “Who are passing by Konkuk University?”, etc.

3 MBR-tree

This chapter explains motivation for MBR-tree proposed in this paper as well as its index structure and data structure. Lastly, it examines the insert, update, delete, and search algorithms of MBR-tree in detail.

3.1 Motivation

Real moving objects do not move in a free space as in Figure 2(a) but their movement is restricted by surrounding environments as in Figure 2(b). That is, the movement of a moving object is restricted by buildings and roads on the map. Thus, if the MBR of nodes in R-tree is formed with the MBR of static objects on the map, the index on the location of moving objects is not updated in case the moving object moves within a static object.

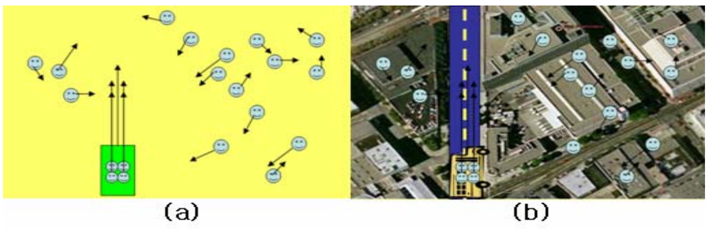

Fig. 2. Movement of Moving Objects

Figure 3 shows differences in insert and update between R-tree and MBR-tree. Figure 3(b) shows the result of inserting O_1 to R-tree in Figure 3(a). In Figure 3(b), R_1 is expanded to minimize the MBR of the node to which O_1 was inserted. Figure 3(f) shows the result of creating the MBR in MBR-tree with static objects in Figure 3(e) and inserting O_1. Because the MBR is fixed in MBR-tree, the index is not reconstructed.

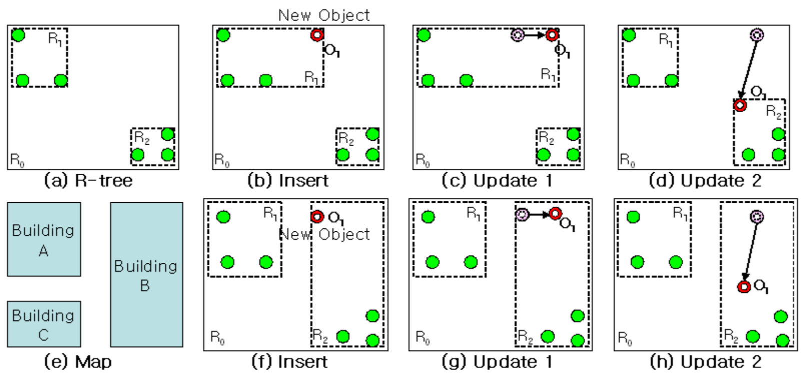

Fig. 3. Insert and Update in R-tree and MBR-tree

In the same way, R-tree is reconstructed as location is updated as in Figure 3(c) and Figure 3(d), but only the location of O_1 is updated in MBR-tree without changing the information of R_2 when updating the location of O_1 within a static object as in Figure 3(g) and Figure 3(h) and, by doing so, it can reduce the cost of update. In addition, because the MBR that manages moving objects uses static objects that have semantic spaces, MBR-tree can be more efficient in semantic search than R-tree.

3.2 Structure of MBR-tree

Figure 4 shows the overall structure of MBR-tree. MBR-tree is composed of **Base R-tree** and **Quad-tree**. Base R-tree is an index constructing the nodes of R-tree using the MBR of static objects on the map. Quad-tree is an index, connected to Base R-tree, to store the ID and location of moving objects, enabling efficient search even when a leaf node in Base R-tree manages a large number of moving objects. The **Secondary Index** is an auxiliary index for high-speed update using the ID of moving objects as the key, pointing the nodes of Base R-tree to which moving objects are inserted.

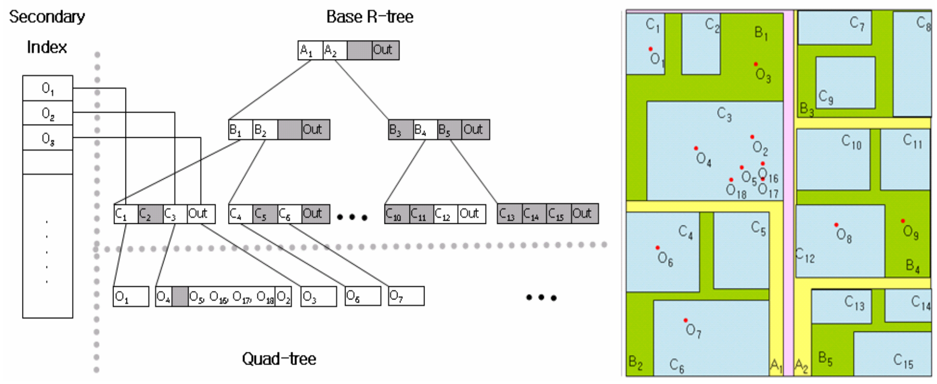
Secondary Index
Base R-tree
Quad-tree
Out
A1 A2
B1 B2 B3 B4 B5
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12 C13 C14 C15
O1 O2 O3 O4 O5 O6 O7 O8 O9 O16 O17 O18

Fig. 4. Structure of MBR-tree

In Base R-tree in Figure 4, moving objects are divided into two types: an **In-Object** moving inside a static object, and an **Out-Object** moving outside static objects. The In-Object is managed in BRANCH of the leaf node containing the object in Base R-tree, and the Out-Object is managed in the node with the smallest MBR size containing the object among the nodes of Base R-tree.

Figure 5 shows the data structure of MBR-tree. In Figure 5, **RNODE** is the data structure of Base R-tree nodes. RNODE has *ParentPt* a pointer to the parent node, *Level* information on node level, *Branch* information on child nodes, *Count* the number of child nodes, and *MObjCount* the number of moving objects inserted into the child nodes. It also has *OutMObjQuad* a pointer to a Quad-tree node to store Out-Objects.

BRANCH has *ChildPt* a pointer to the child node of RNODE, *mbr* to store the MBR of child nodes, and *InMObjQuad* a pointer to a Quad-tree node to store In-Objects. **QUADNODE**, which is the data structure of Quad-tree nodes, has *Count* the number of moving objects stored, *MObj* a pointer to the first moving object, and *ChildPt* a pointer to the child node of Quad-tree. **MOBJECT**, which is the data structure to store information on moving objects, has *Oid* the ID of the object, *Loc* location information, and *NextPt* a pointer to the next moving object.

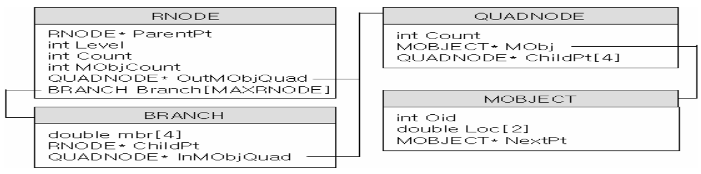

Fig. 5. Data Structure of MBR-tree

3.3 Algorithms

This section examines the insert, update, delete, and search algorithms of MBR-tree in detail.

3.3.1 Insert Algorithm

The insert algorithm of MBR-tree is as in Figure 6. It is executed with input *oid* the ID of a moving object and *loc* the location of the object.

```
Algorithm : Insert (oid, loc)
    begin
1:      [RNodePt, BranchID] ← Find_InNode (loc)
2:      if (RNodePt is not NULL) //Moving Object is In-Object
3:          Insert [oid, loc] into RNodePt.Branch[BranchID].InMObjQuad
4:          Insert [oid, loc, RNodePt, BranchID] into Secondary Index
        else //Moving Object is Out-Object
5:          RNodePt ← Find_OutNode (loc)
6:          Insert [oid, loc] into RNodePt.OutMObjQuad
7:          Insert [oid, loc, RNodePt, BranchID] into Secondary Index
        end if
8:      while (RNodePt.ParentPt is not NULL)
9:          RNodePt.MObjCount ++
10:         RNodePt ← RNodePt.ParentPt
        end while
    end
```

Fig. 6. Insert Algorithm

```
Algorithm : Update (oid, new_loc)
    begin
1:      Find [RNodePt, BranchID] with oid in Secondary Index
2:      if (BranchID is not NULL) // Moving Object is In-Object
3:          if(RNodePt.Branch[BranchID].mbr Contained new_loc)
4:              Update new_loc in RNodePt.Branch[BranchID].InMObjQuad
5:              Update new_loc in Secondary Index
            else
6:              Delete (oid)
7:              Insert (oid, new_loc)
            end if
        else // Moving Object is Out-Object
8:          Delete (oid)
9:          Insert (oid, new_loc)
        end if
    end
```

Fig. 7. Update Algorithm

The insert algorithm inserts a moving object, distinguishing it between In-Object and Out-Object. As in Figure 6, if the input moving object is an In-Object, the algorithm finds the leaf node in Base R-tree containing *loc* using Find_InNode(loc) function and inserts the moving object into the InMObjQuad of the corresponding branch. If the moving object is an Out-Object, it finds the node with the smallest MBR among Base R-tree nodes containing *loc* using Find_OutNode(loc) function and inserts the moving object into OutMObjQuad. After insertion, it increases MObjCount by 1 from the root node to the node to which the object has been inserted.

3.3.2 Update Algorithm

The update algorithm of MBR-tree is as in Figure 7. It is executed with input *oid* the ID of a moving object and *new_loc* the new location. The update algorithm does not reinsert the moving object to be updated within the same static object into MBR-tree.

As in Figure 7, the update algorithm finds the pointer to the corresponding node and Branch ID in the secondary index using *oid*. If the input moving object is an In-Object and the updated location does not deviate from the MBR of the corresponding branch, the object is not reinserted into InMObjQuad but only the location information of the moving object is updated. However, if it deviates from the MBR, the moving object is reinserted. If the input object is an Out-Object, it is reinserted into MBR-tree.

3.3.3 Delete Algorithm

The delete algorithm of MBR-tree is as in Figure 8. It is executed with input *oid* the ID of a moving object.

```
Algorithm : Delete (oid)
   begin
1:   Find [RNodePt, BranchID] with oid in Secondary Index
2:   if (BranchID is not NULL)  // Moving Object is In-Object
3:      Delete In-Object with oid in RNodePt.Branch[BranchID].InMObjQuad
     else // Moving Object is Out-Object
4:      Delete Out-Object with oid in RNodePt.OutMObjQuad
     end if
6:   Delete Moving Object with oid in Secondary Index
7:   while (RNodePt.ParentPt is not NULL)
8:      RNodePt.MObjCount--
9:      RNodePt←RNodePt.ParentPt
     end while
   end
```

Fig. 8. Delete Algorithm

```
Algorithm : Search_Window (RNodePt, Rectangle)
   begin
1:   if (RNodePt.OutMObjQuad is not Null)
2:     Find Moving Object that inside Rectangle in RNodePt.OutMObjQuad
     end if
3:   if (RNodePt.Level is ZERO) // Leaf Node of RNODE
4:     for each RNodePt.Branch in RNodePt
5:       if (RNodePt.Branch.mbr intersects Rectangle)
6:         Find Moving Object that inside Rectangle in RNodePt.Branch.InMObjQuad
         end if
       end for
     else // Internal Node of RNODE
7:     for each RNodePt.Branch in RNodePt
8:       if (RNodePt.Branch.mbr intersects Rectangle &
                                      RNodePt.MObjCount is not ZERO)
9:         Search_Window (RNodePt.Branch.ChildPt, Rectangle)
         end if
       end for
     end if
   end
```

Fig. 9. Search Algorithm

The delete algorithm accesses the corresponding node using the secondary index and deletes the moving object. As in Figure 8, the delete algorithm finds the pointer to the corresponding node and Branch ID in the secondary index using *oid*. If the moving object is an In-Object, it is deleted from InMObjQuad of the corresponding branch, and if it is an Out-Object, it is deleted from OutMObjQuad of the corresponding node. After deletion, it decreases MObjCount by 1 from the node from which the object has been deleted to the root node.

3.3.4 Search Algorithm

The search algorithm of MBR-tree is as in Figure 9. It is executed with input *RNodePt* a node pointer in Base R-tree and *Rectangle* a window range. The search algorithm retrieves all moving objects included in *Rectangle* among nodes managing moving objects. As in Figure 9, the search algorithm retrieves Out-Object included in *Rectangle* if there are moving objects in OutMObjQuad of input *RNodePt*. Next, it checks if the current node is a leaf node and, if it is, the algorithm retrieves moving objects included in *Rectangle* among In-Objects stored in MObjQuad of the branches. If the current node is not a leaf node, it checks if there are moving objects in its child nodes, and the search algorithm is executed recursively for child nodes.

4 Performance Evaluation

This chapter compares MBR-tree with LUR-tree and hashing technique through evaluating their performance. Performance evaluation was made by comparing update performance, window query performance and semantic search performance in terms of time and memory usage.

4.1 Experiment Environment

Performance evaluation was made using a PC with Intel Pentium4 2.53GHz CPU and 1GB memory. Data used in performance evaluation was generated from City Simulator and GSTD. Figure 10(a) shows the map used in City Simulator and data generated from it, and Figure 10(b) shows data generated from GSTD.

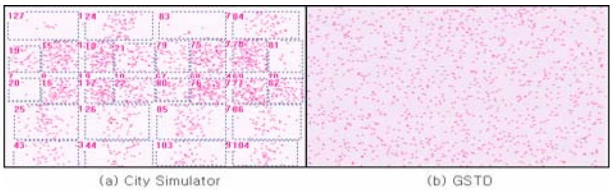

Fig. 10. Data Generated from City Simulator and GSTD

In performance evaluation, we used hashing with 144 buckets (12X12) and hashing with 324 buckets (18X18) to compare with MBR-tree. The number of buckets in hashing (12X12) is similar to the number of leaf nodes in MBR-tree.

4.2 Update Performance Evaluation

Update performance evaluation uses trajectory data in which 1000/3000/5000 moving objects move 300 times as generated from City Simulator and GSTD. Figure 11 shows graphs that compare the index update performance according to the number of moving objects.

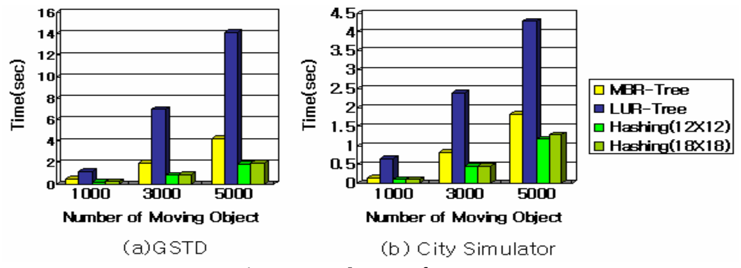

Fig. 11. Update Performance

The results of performance evaluation show that MBR-tree is much superior to LUR-tree in the update performance and not much inferior to hashing, which generally has high update performance.

4.3 Query Performance Evaluation

Query performance evaluation was made using the data of 50,000 moving objects generated from City Simulator and GSTD.

4.3.1 Window Query Performance Evaluation

Figure 12 shows graphs that compare the window query performance according to window size.

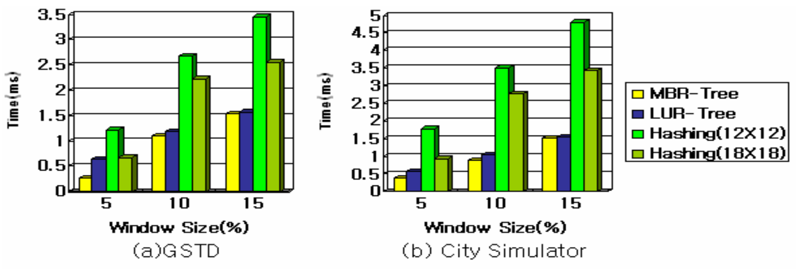

Fig. 12. Window Query Performance

The results of performance evaluation show that the window query performance of MBR-tree is 2.1 times higher than that of hashing on the average and higher than that of LUR-tree based on R-tree, which generally has high window query performance.

4.3.2 Semantic Search Performance Evaluation

Figure 13 is a graph that compares the semantic search performance.

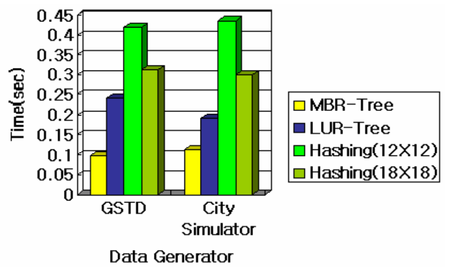

Fig. 13. Semantic Search Performance

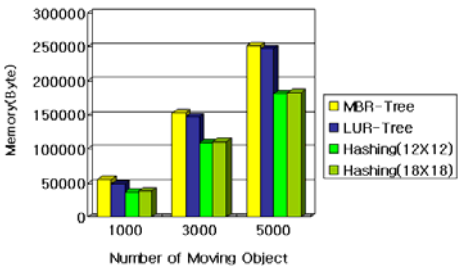

Fig. 14. Memory Usage

The results of performance evaluation show that the semantic search performance of MBR-tree is 2.1 times higher than that of LUR-tree and 3.4 times higher than that of hashing on the average.

4.4 Memory Usage

Because memory usage is closely related to the number of moving objects regardless of the type of dataset, we use data of 1000/3000/5000 moving objects generated from GSTD. Figure 14 is a graph that compares the size of memory usage according to the number of moving objects.

In the comparison of memory usage, MBR-tree shows a slight difference from LUR-tree in memory usage but uses 1.4 times larger memory space than hashing, which generally uses a small size of memory.

5 Conclusions

This paper proposed MBR-tree that can index location data of moving objects by forming R-tree nodes using the MBR of static objects on the map. MBR-tree reduced update cost and improved the semantic search performance by managing moving objects in the unit of static object.

In the results of performance experiment, hashing showed advantage in memory usage and update speed but was much inferior in the search performance, and LUR-tree showed advantage in the search performance but its update cost was high. MBR-tree reduced update cost effectively compared to LUR-tree while guaranteeing search speed higher than hashing. Particularly in semantic search, MBR-tree showed much higher performance than all the other methods. Accordingly, MBR-tree can have high utility in the environment where search transactions are as important as update transactions and where high semantic search performance is required.

Acknowledgements

This research was supported by the Seoul Metropolitan Government, Korea, under the Seoul R&BD Program supervised by the Seoul Development Institute.

References

1. Kwon, D.S., Lee, S.J., and Lee, S.H.,: Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. Proc. of the Third International Conference on Mobile Data Management, (2002) 113-120.
2. Inam, O., and Matin, A.,: A Survey of Indexing Techniques for Moving Object Trajectories. Technical Report, University of Waterloo, (2003).
3. Mokbel, M.F., Ghanem, T.M., and Aref W.G.,: Spatio-Temporal Access Methods. IEEE Data Eng. Bull., Vol.26, No.2 (2003) 40-49.
4. Roddick, J.F., Hoel, E., Egenhofer, M.J., and Papadias, D.,: Spatial, Temporal and Spatio-Temporal Databases: Hot Issues and Directions for PhD Research. ACM SIGMOD Record, Vol.33, No.2 (2004) 126-131.
5. Roth, J.,: Novel Architectures for Location-Based Services. Annual Meeting for Information Technology & Computer Science, (2004) 5-8.
6. Song, Z., and Roussopoulos, N.,: Hashing Moving Objects. Proc. of the 2nd International Conference on Mobile Data Management, (2001) 161-172.
7. Theodoridis, Y., Silva, J.R.O., and Nascimento, M.A.,: On the Generation of Spatiotemporal Datasets. Proc. of the 6th International Symposium on Advances in Spatial Databases, (1999) 147-164.