Fast indexing and updating method for moving objects on road networks

Kyoung-Sook Kim, Si-Wan Kim, Tae-Wan Kim and Ki-Joune Li
Dept. Computer Science and Engineering
Pusan National University
Pusan 609-735, South Korea
kskim,swkim,twkim@stem.cs.pnu.edu, lik@pnu.edu

Abstract

Fast updates in spatial databases become a crucial issue in several application domains, such as LBS (Location Based Service). In particular, moving objects with frequent updates on their positions require fast update mechanisms in spatiotemporal database systems. In this paper, we propose a new method for the indexing of the current positions of moving objects on road network. Our method significantly improves not only the update cost but also the performance of query processing. The only constraint of our method is that the moving objects should be on roads, but we believe that most applications dealing moving objects imply this constraint. We provide a performance analysis on our model and by this we suggest further improvements on performance of our method. By experimental studies, we show that our method takes about 40% less in disk accesses compared to TPR-tree for updating and at maximum 4 times faster than TPR-tree for processing timestamp range queries.

1. Introduction

Moving objects are being widely treated in mobile applications, such as Location Based Service(LBS). There are however some serious problems in storing and handling them in spatiotemporal database system(ST-DBMS for short). Among them, we restrict our concerns on performance. Many moving objects frequently change their positions and submit update requests to ST-DBMS. This results in a considerable overhead on the system performance. If we do not provide a fast update mechanism, the application domains will remain restricted as clearly mentioned by Jensen and Saltenis [1].

Once an update request is submitted, several modules of ST-DBMS are involved in processing it. They include 1) storage system, 2) indexing mechanism and 3) transaction management. In this paper, we focus on how to reduce the update cost of indexing. We can significantly improve the update performance by providing an efficient novel indexing method. While many indexing methods have been proposed for moving objects, their aims are to improve the performance of indexing and query processing rather than to reduce update cost. For example, suppose that we have millions of moving objects and each of them issues an update request on its position at every minute. Furthermore each update must be handled within 0.06 msec. These requirements are common in LBS and seem to be a difficult task to satisfy.

In this paper, we propose an indexing method to handle current positions of moving objects and it significantly reduces the update cost. Our approach is motivated by two observations. First, any index structure consists of two parts; static one, which is almost invariant regardless of updates, and dynamic one, which is directly related with the update overhead. We should therefore maximize the static part and minimize the dynamic part to reduce the update overhead. Second, most moving objects are on road networks in real word, which is often termed one and half dimensional space in the literature [1] [5]. Our indexing method, *Indexing Moving Objects on road sectors* (IMORS for short), is based on a realistic assumption that all moving objects are on predefined road networks. We exploit this assumption to maximize the static part of index.

We present related works and motivations of our work in the next section. Section 3 describes our data structures and associated algorithms for moving objects. An analytical studies are given in section 4. By analysis, we can suggest further improvement on our method. Section 5 reports on performance experiments and comparison with other indexing methods. We conclude our paper in section 6.

2. Related Works and Motivations

Several methods have been proposed for indexing moving objects. They are classified into two categories; one of them is to handle current and future positions of moving objects [2][3][6][7] [8], and the other is to handle past positions or trajectories [4][5]. Each of them requires quite different functions and approaches. In this paper, we focus on the former one.

Most of indexing methods for current positions treat point moving objects and are based on extensions of existing spatial indexing methods, such as R*-tree[9], Kd-tree, Quad-tree, and so on. Time-Parameterized R-tree(*TPR*-tree) [2] is an extension of R*-tree with some additional considerations on the movement and anticipated future position of moving objects. It employs *time-parameterized bounding rectangles* to reflect near-future positions of moving objects instead of simple MBRs in R*-tree. An improvement of TPR-tree, called R^{EXP}-tree [3], has been proposed to compute better time-parameterized bounding rectangles as it calculate expired bounding boxes earlier than TPR-tree.

Data transformation techniques are introduced to use existing spatial indices, such as Kd-tree [6], multi-level partition tree [7], and R*-tree [8]. They exploit the duality between a line in the original space and a point in higher dimensional space. For example, Agarwal et. al. proposed an indexing method for moving objects based on the dual data transformation from a line segment with two ending points to a point in 4-dimensional space[7], and Kollios et. al. employ the dual transformation from a line $x = x_{ref} + v(t - t_{ref})$ to a point (x_{ref}, v)[6]. These indexing methods focus on improving the performance of query processing. But, the performance of systems which treat current positions of moving objects depend on not only query processing but also updating cost. Even though most of them use a function of time to reduce update cost, they may render the update cost expensive to maintain their sophisticated data structures. An analysis on the update cost is done by Wolfson et. al.[13][14]

Our work aims to provide a fast updating mechanism and an indexing method. Our approach is based on two observations. First dynamic index structures should carry out operations like split and merge for updates and these complex operations lead to the reorganization of a whole data structure. We want therefore to reduce dynamic overhead of an index structure to solve this problem. The second observation is that moving objects almost stay in road networks[10]. For example, vehicles are on streets except for some special cases. This fact allows us not only to reduce the uncertainty of the positional data but also to greatly simplify a lot of operations in indexing.

By combining these observations, we propose a hybrid indexing method for moving objects. It consists of two parts; dynamic part and static part. Then we construct road networks as the static part and moving objects as dynamic part of indexing. We can improve the performance of updating because road networks rarely change, which are defined beforehand.

A similar method has been proposed by Song and Roussopoulos [12], which employs a hashing technique as its data structures. An update does not occur until an object moves into a new grid, therefore the update cost is greatly reduced. But the drawback of hashing method is that a hash function depends on the distribution of data. Since they have made a hash function based on the initial status and the objects move, initial status changes as time goes by and thus a hash function must be reconstructed. Consequently, the performance of indexing receives the influence at the dynamic property of moving objects. We should consider the indexing method to reduce dynamic parts.

3 IMORS: Indexing Moving Objects on Road Sectors

In this paper, we assume that *all moving objects are on road sectors*. Most of moving objects, such as vehicles, trains or even pedestrians walking along a street, belong to this category. And even though a moving object is not exactly on a road, we extend this assumption to a moving objects which are not too far from a road.

As mentioned in the previous section, we should reduce the part of data structure dynamically changing upon update requests. And the above assumption provides a very important clue to reduce the dynamic part or highly utilize the static part of index. Since the information about the geometry and connectivity of road networks rarely changes, IMORS incorporates it as a static part. In this section, we present the basic data structure and algorithms of IMORS.

3.1 Index Data Structure

Figure 1 illustrates the overall data structures of IMORS, which consists of two parts as mentioned in the previous section. First the static part contains only R*-tree built on road sectors. Note that a set of road sectors are contained in each leaf node of R*-tree. Second the dynamic part has two principal blocks structures; *Road Sector Blocks*($B_{RoadSect}$ in short), and *Data Blocks for Moving Objects* (B_{data}). Road sector means a non-intersecting segment of a road bounded by two intersecting points. Obviously, the road sector consists of several line segments and thus is polylines.

The R*-tree used in IMORS is similar to original R*-trees except that each of entry in a leaf node consist of road sectors rather than their MBRs. Unlike points or polygons, polylines may result in a considerable dead space within their MBRs and a performance degradation. Deciding the length of road sectors affects the performance of R*-tree. For example, we may divide a road sector into several smaller ones to reduce the dead space, but such a

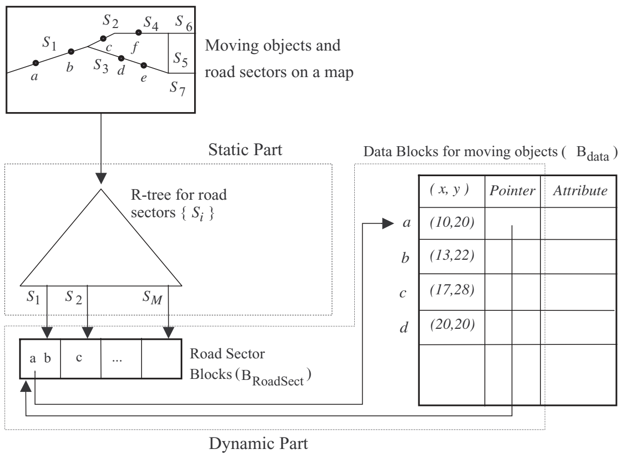
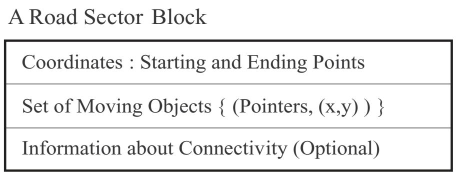

Moving objects and road sectors on a map
S_1 S_2 S_3 S_4 S_5 S_6 S_7
a b c d e f
Static Part
R-tree for road sectors { S_i }
S_1 S_2 S_M
Road Sector Blocks ($B_{RoadSect}$)
Data Blocks for moving objects (B_{data})
(x, y) Pointer Attribute
a (10,20)
b (13,22)
c (17,28)
d (20,20)
Dynamic Part

Figure 1. Overall data structure of IMORS

division leads to an increase of entries, and thus increases search paths. A detailed analysis will be given in section 4. But for simplicity, we assume that the maximum length of road sector is bounded by two intersecting points.

Each road sector entry on leaf nodes of R*-tree points to a block of $B_{RoadSect}$, which contains the identifiers of the moving objects on this road sector. This pointer allows to retrieve the moving objects on a road sector via R*-tree. Figure 2 depicts the data structure of a road sector block. The data blocks for moving objects (B_{data}) store the velocities and other related attributes of each moving object. Note that there are bi-directional pointers between each record in the data block and a moving object of road sector blocks. This bidirectional pointers improve the performance of the update and query processing, which we will discuss in the next subsection.



A Road Sector Block

Coordinates : Starting and Ending Points
Set of Moving Objects { (Pointers, (x,y)) }
Information about Connectivity (Optional)

Figure 2. Data structure of road sector bucket

Once an update on positional data of a moving object is submitted, the coordinates data in $B_{RoadSect}$ and B_{data} must be updated and the membership relation between the moving object and $B_{RoadSect}$ is to be changed if necessary. However no change occurs in the static part, R*-tree, and this is the key idea of IMORS. It means that we limit the scope of updates by separating dynamic and static parts of index data structure.

The updates on the data blocks B_{data} are inevitable in all cases. Consequently the additional update cost of IMORS lies on only the modification cost of bidirectional pointers between $B_{RoadSect}$ and B_{data}. This modification may result in overhead cost, but it is necessary only if the membership of a given moving object on a road sector is changed. The probability that the membership changes depends on the velocity of moving object and the length of road sector. We will discuss it in detail in section 4.

```
Insertion Algorithm($m$, $R$)
Input $m$:  new moving object, $R$:  R*-tree
Begin Algorithm
    $m$.oid ← Register_MO_on_DB($m$);
    $b_{RS}$ ← Search_R*-tree($R$, $m$.pos);
    $b_{ptr}$ ← Register_on_RSB($b_{RS}$, $m$.oid, $m$.pos);
    Register_Block_Pointer_on_DB($m$.oid, $b_{ptr}$);
End Insertion Algorithm
```

Figure 3. Insertion algorithm of IMORS

3.2 Insertion and Deletion

Inserting a new moving object into IMORS is carried out by three steps. First, a new object register its oid in B_{data} with its coordinates and other attributes. For the second step, we search R*-tree to retrieve a road sector where a new object is contained. And finally, a road sector block pointed by above road sector is now associated with oid of a new object in B_{data}. For association, we assign pointers to both of $B_{RoadSect}$ and B_{data}. The detailed algorithm is given in figure 3.

Note that the new moving object is to be registered in an overflow block, if there is no room on the road sector block $B_{RoadSect}$. This may happen when the road sector is congested by moving objects. Deletions in IMORS are performed by removing a target moving object from B_{RS} and B_{data}. We can merge B_{data} if necessary.

3.3 Update

Figure 4 shows the overall procedure of an update operation which is rather complex compare to insertion and deletion.

Suppose that a new position of a moving object with its object identifer is reported to the system. We retrieve the record from the data block by its oid and update its coordinates. Since this record has the pointer to the road sector block containing this object, we can directly access the block. If the object is still on the road sector, no more operation is needed. Otherwise we should place it on another

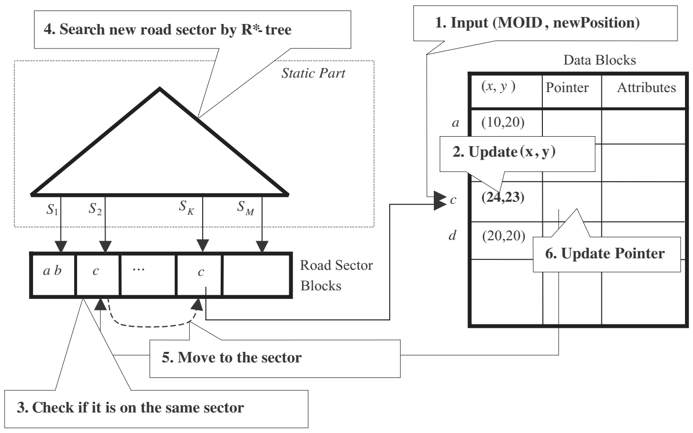


Figure 4. Update procedure of IMORS

```
Update Algorithm
Input  $m$:  moving object with new position
       $R$:  R*-tree
Begin Algorithm
  Update_Position($m$.oid, $m$.pos);
  $b_{RS} \leftarrow$ Get_RoadSectorBlock($m$.oid);
  If ( Is_on_RS($b_{RS}$.seg,$m$.pos) == NO ) {
    Remove_from_RSB($b_{RS}$, $m$.oid);
    $b_{RS} \leftarrow$ Search_R*-tree($R$,$m$.pos);
    $b_{ptr} \leftarrow$ Register_on_RSB($b_{RS}$,$m$.oid,$m$.pos);
    Register_Block_Pointer_on_DB($m$.oid,$b_{ptr}$);
  }
End Update Algorithm
```

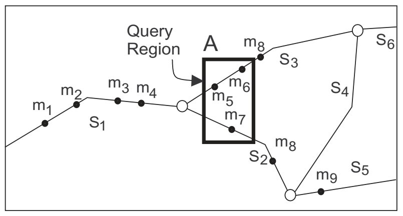
Figure 5. Update algorithm of IMORS

road sector block. In order to determine the new road sector block, we lookup the R–tree and get the road sector where the new location of the object is placed. Finally, we change the bidirectional pointers of the new road sector block and data block. This procedure is briefly described in figure 5.

3.4 Query Processing

Several types of queries on the current positions of moving objects can be processed by IMORS. In this paper, we present an algorithm for region query. The algorithm consists of two steps; finding road sectors and finding moving objects. Suppose that a query is given as "*find moving objects in region A*" as shown by figure 6. We can search the road sectors intersecting or contained within the query region via R*-tree. And each moving object in the road sectors is examined if it is contained within the query region. Figure 6 illustrates this procedure and a algorithm is given in figure 7. The performance of this algorithm is determined by two factors; the R*-tree on road sectors and the number of selected road sector blocks. And they mostly depend on the number and the length of road sectors. In section 4, we will discuss it in detail.

Several indexing methods for moving objects [2] [6] process the queries on the near-future positions. They compute the positions of moving object in the near future by linear interpolation with velocity. We do not treat the query on the anticipated future positions of moving objects in this paper. However we can compute the future positions more accurately than a simple linear interpolation, since we have the information on the road network where moving objects will stay. And by maintaining the road network, we can estimate the future positions even more accurately. It will be interesting to implement this approach in IMORS for future work.



R*-tree Search : { S_2, S_1 }
Refinement of moving objects : { m_5, m_6, m_7 }

Figure 6. Example of query processing

```
Containment Region Query Processing
Input   $W$:  Query Region
        $R$:  R*-tree
Output  $M$:  Set of Moving Objects
Begin Algorithm
  $B_{RS} \leftarrow$ Range_Search_R*-tree($R$,$W$);
    // $B_{RS}$ is a set of road sector blocks
    // $b_{RS}$ is a road sector block
  $M \leftarrow \{\}$;
  For each $b_{RS}$ in $B_{RS}$
    For each $m$ in $b_{RS}$.M
      /* $b_{RS}$.M is the set of moving objects
      contained by $b_{RS}$ */
      If $m$ is contained by $W$,
        Then $M \leftarrow M \cup \{m\}$;
  return $M$
End Containment Region Query Processing
```

Figure 7. Query processing algorithm of IMORS

Table 1. Symbol table

Symbol	Description
C_Q	the cost for query processing
C_U	the cost for updating
C_{RT}	the cost for R*-tree access
C_{RS}	the cost for road sector access
C_{DB}	the cost for data block access
C_{NRS}	the cost for new road sector access
l_s	the average length of a road sector
n_s	the number of road sectors
n_{opt}	the number of road sector to be optimal
$s_{x,i}$	the average x-axis length of i-th road sector s_i
$s_{y,i}$	the average y-axis length of i-th road sector s_i
q_x	the x-axis lengths of query
q_y	the y-axis lengths of query
Bf	the blocking factor of a data block
L	the total length of all road sectors
L_x	the sums of length of all road sectors along x
L_y	the sums of length of all road sectors along y
f	the update frequency
v	the average velocity of moving objects
ρ	probability of changing the road sector

4 Analysis of Performance

In this section, we present an analysis on performance of IMORS for updating and query processing. Analytic cost models will be given to tune the performance as well.

For a fast updating and query processing, main memory database systems may be used. But it is impossible to load the whole database concerning moving objects due to its huge size except some special cases. A large part of the database must still reside on disk space and we can reduce the page fault ratio, by providing more available main memory space. In our analysis, the number of disk accesses is of interest for this reason and used as the measure of performance.

4.1 Performance of Query Processing

We discuss on the performance of the region query processing that we have seen in section 3.4. The query processing cost of IMORS (C_Q) can be represented as follows,

$$C_Q = C_{RT} + C_{RS} + C_{DB}. \qquad (1)$$

where C_{RT}, C_{RS}, and C_{DB} represent the costs for R*-tree access, road sector block access and data block access, respectively. Since C_{DB} depends only on the number of moving objects, it is not of our interest and we analyze two costs, C_{RT} and C_{RS}.

First, the R*-tree access cost, C_{RT} depends on several factors, such as the distribution of data, the size of road sectors, query size, etc. But except for road sector size, they are given by application and they cannot be controlled to improve the performance. For this reason, we should find a cost model by the function of road sector size.

The R*-tree in IMORS contains a set of polylines unlike other R*-trees. In analyzing the performance of the R*-tree, we should take into account this particular property. As shown in figure 8, a polyline object may cause a large dead space, which results in a degradation of the performance. In order to reduce the dead space, we could divide a polyline into several pieces. However, the division increases the number of objects and consequently the number of selected road sectors. But since each road sector has at least one block in IMORS, the increase of selected road sectors results in an increase of accesses to road sector blocks. By properly controlling the road sector size in IMORS, we improve the performance of R*-tree.

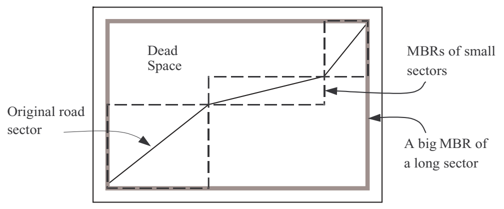


Figure 8. Dead space and length of road sector

Pagel et. al. [18] and Kamel et. al. [17] have proposed a simple and accurate cost model of R-tree describing the relationship between object size and the cost of leaf node access,

$$C_{RT} = \sum_{i=1}^{n_s} (s_{x,i} + q_x)(s_{y,i} + q_y) \qquad (2)$$

where $s_{x,i}$, $s_{y,i}$, q_x, and q_y are the length of i-th road sector s_i and the side lengths of query q along x and y-axis, respectively and n_s is the number of road sectors.

In fact, eq.(2) indicates the number of the selected objects by a query at leaf nodes. A more accurate cost model proposed by Theodoridis and Sellis [11] could be employed to estimate the cost of internal node accesses and the height of R*-tree. But it is complicated and we consider only the cost of leaf node accesses to simplify our cost model. This equation allows us to find approximately the optimal size of road sectors. Let the sums of length of all road sectors along x and y-axis be L_x and L_y respectively. Then $L_x = \sum_{i=1}^{n_s} s_{x,i}$ and $L_x = \sum_{i=1}^{n_s} s_{x,i}$. If s_x and s_y are the average extents of road sectors along x and y-axis, $s_x = \frac{L_x}{n_{sect}}$ and $s_y = \frac{L_y}{n_s}$. For the simplicity, we assume that $s_{x,i} = s_x$ and $s_{y,i} = s_y$ for $i = 1, ..., n_s$. Therefore, eq.(2) becomes

$$C_{RT}(n_s) = \sum_{i=1}^{n_s}(s_{x,i}+q_x)(s_{y,i}+q_y)$$
$$= n_s(\frac{L_x}{n_s}+q_x)(\frac{L_y}{n_s}+q_y)$$
$$= \frac{L_xL_y}{n_s} + n_sq_xq_y + L_xq_y + L_yq_x \qquad (3)$$

We can obtain the minimum cost of $C_{RT}(n_s)$, when $n_s = n_{opt}$. From $C'_{RT}(n_{opt}) = 0$, we see that

$$n_{opt} = \sqrt{\frac{L_xL_y}{q_xq_y}} \qquad (4)$$

Second, let us discuss on the cost for road sector block accesses (C_{RS}) in eq.(1). The cost is determined by two factors; the number of road sectors and the number of overflow blocks. But from several experiments we observed that overflow blocks are rarely found, and in most cases, a block of e.g. 16 K bytes is sufficiently large to store all moving objects on a road sector. For this reason we ignore the cost for overflow block. Then the cost depends on only the number of road sectors, which we have analyzed in the previous paragraph. It means that C_{RS} is minimum when $n_s = \sqrt{\frac{L_xL_y}{q_xq_y}}$ holds as eq.(4).

From these two observations, we conclude that the cost of query processing of IMORS is minimum when $n_{sect} = \sqrt{\frac{L_xL_y}{q_xq_y}}$. It gives a very useful hint to define the optimal length of road sector and to tune the performance. Let L is the total length of road sectors, then the length of each road sector l_s is determined by

$$l_s = \frac{L}{n_s} \qquad (5)$$

It means that we can tune the performance of query processing by dividing a road sector into several small ones, so that the length of each small sector should be near to $\frac{L}{n_s}$. A delicate problem lies on deciding the size of query (q_x, and q_y), since we may have no idea a priori. But we can compute the size, by archiving the statistics on query sizes.

4.2 Update Cost

From figure 4 in section 3, we see that the update cost of IMORS consists of 1) the cost for accessing data block C_{DB}, 2) the cost for accessing of road sector block C_{RS}, 3) the cost for R*-tree searching C_{RT}, and 4) the cost for changing road sector block C_{NRS}. Note that the cost for updating bidirectional pointers in figure 4 is negligible since the blocks containing the pointers are already read at previous steps. Thus the cost for update C_U is described as follow,

$$C_U = C_{DB} + C_{RS} + \rho(C_{RT} + C_{NRS}) \qquad (6)$$

where ρ represents the probability that a moving object should change the road sector. Each access to data block, road sector block or new road sector block needs only one disk access ($C_{DB}, C_{RS}, C_{NRS} = 1$). The object type on R*-tree is of point as explained in section 3.3 and this query point, which is in fact a new position of a moving object, is always contained by a road sector at leaf node. This means that the cost for R*-tree searching, C_{RT} is determined by the height of the tree, given as $h = \log_{Bf}\frac{n_s}{Bf}$.

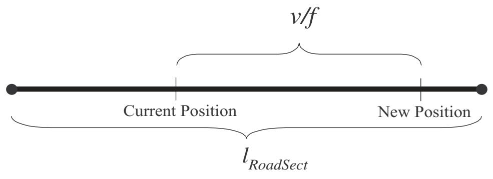


Figure 9. Length of road sector, velocity, frequency and and probability ρ

The most significant factor in eq.(6) is the probability ρ. In other words, we can improve the update performance by reducing the probability. It depends on three parameters, the length of road sector l_s, the velocity v, and the update frequency f. The relationship between ρ and these parameters are depicted by figure 9, and the probability is given as

$$\rho = \min(1, \frac{v}{l_sf}) \qquad (7)$$

We derive the following equation from eq.(6), eq.(7) and $l_s = \frac{L}{n_s}$,

$$C_U(n_{sect}) = C_{DB} + C_{RS} + \rho(C_{RT} + C_{NRS})$$
$$= 2 + \min(1, \frac{v}{l_sf})(\log_{Bf}\frac{n_s}{Bf} + 1)$$
$$= 2 + \min(1, \frac{n_sv}{fL})(\log_{Bf}\frac{n_s}{Bf} + 1) \qquad (8)$$

or

$$C_U(l_s) = 2 + \min(1, \frac{v}{l_sf})(\log_{Bf}\frac{L}{l_sBf} + 1) \qquad (9)$$

where Bf means the blocking factor. In order to reduce C_U, we should enlarge the length of road sector l_s or reduce the number of road sectors n_s, since the velocity and update frequency of moving objects are fixed by application. However, there is a tradeoff between the performances of query processing and updates. While we can improve the update performance by enlarging the length of road sector, it may result in a degradation of query processing performance. We should make a compromise between them by considering the type and workload of application. Note that the maximum length of road sector is limited by two intersection points.

5 Experiments and Discussions

We report on performance experiments with IMORS in this section. Several parameters are examined to figure out their influences on the performance and to compare IMORS with TPR-tree, which is one of the most important indexing methods for moving objects. We prepared a data set with the benchmark data generation tool developed by Brinkhoff [15] [16]. We generated a synthetic data set consisting of 100,000 moving objects on road networks, which are not evenly distributed on road networks but more or less skewed on some roads. Figure 10 illustrates the road networks at Oldenberg in Germany, used in the experiments, where the extent of the map is 23854×30851. They request updates on the positional data every time unit during the simulation. We configured the velocity parameter of this tool as medium.

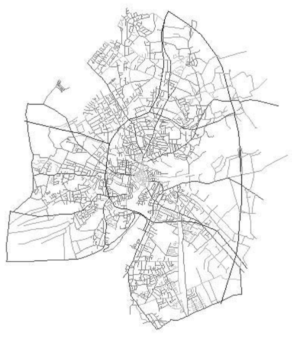


Figure 10. Road networks of test data (at Oldenburg in Germany)

Figure 11 shows the update performance of IMORS. Each curve of the graph corresponds to a block size (4K and 16K bytes respectively). As explained in section 4.2, the performance becomes better by increasing the maximum length of road sector. In any case, the cost of update must be more than two disk accesses, since it needs at least one disk access for data block and one for road sector block. The difference between the data and road sector block accesses and the total cost therefore come from the overhead for changing road sector. An exception is found when the block size is 4K bytes, due to overflow blocks. It is evident that the probability of overflows becomes more important,

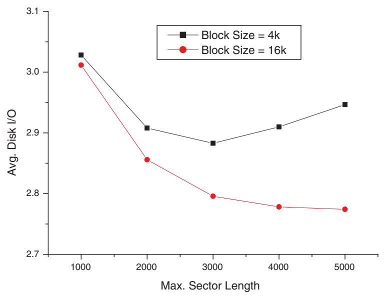


Figure 11. IMORS average disk I/O per update by max. road sector length

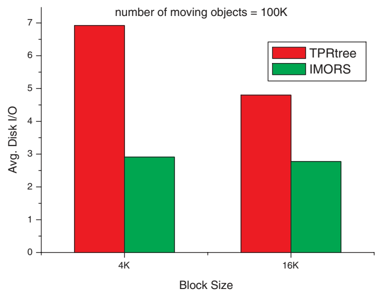
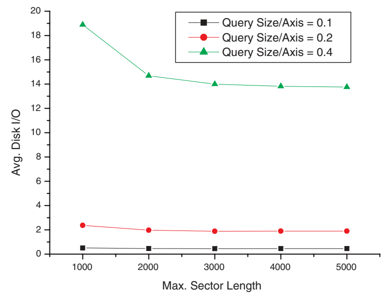


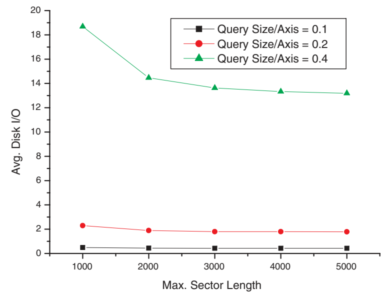
Figure 12. IMORS and TPR-tree average disk I/O per update by block size

as increases the length of sector size. But we simply avoid this problem by increasing the size of block to 16K bytes.

A comparison of update costs between IMORS with TPR-tree is given in figure 12. It shows that the IMORS performs the update operation about twice faster than TPR-tree. Figure 13 shows the average number of disk accesses per query when the sizes of block are given as 4K and 16K bytes and 1,000 randomly generated queries are submitted. The query size in this graph represents the ratio of the query side length over the extents of the map along x and y-axis. When the query length is 0.1, the average number of accesses is even less than 1. It is not surprising since no road sector is found in some query regions, which is consequently filtered out at the root level of R*-tree. An interesting point is that the query performance is very slightly improved by increasing the block size, while we considerably improve the performance in TPR-tree as shown in figure 14a. This comes from some properties of IMORS. First, the depth of R*-tree remains 2 even we increase the block size. Second, if the size of block is sufficiently large to hold all moving objects in one road sector block, the number of road sector block remain constant. These are the reasons

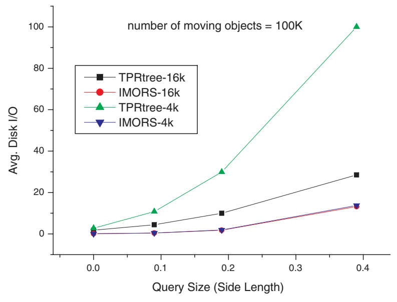


(a) Query performance of IMORS (block Size=4K bytes)

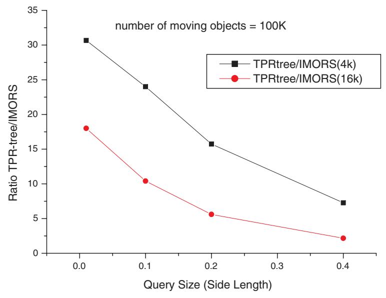


(b) Query performance of IMORS (block Size = 16K bytes)

Figure 13. IMORS avg. disk I/O per query by road sector length



(a) Query performance of IMORS and TPR-tree



(b) Query performance ratio of IMORS over TPR-tree

Figure 14. Query performance comparison between IMORS and TPR-tree ratio avg. disk I/O per query by query size

why no significant improvement of performance is found even though we increase the block size.

The query performance is significantly improved by by IMORS. Figure 14 shows a surprising comparison of query processing performance between IMORS and TPR. When the query size is small (e.g. 0.01), IMORS processes the query more than thirty times faster than TPR-tree. From other experiments, we observed that this ratio becomes more significant as increases the number of moving objects. In this experiment, we examine only queries on the current positions. For this reason, we minimize the time horizon H to reduce the unnecessary computation of time-parameterized bounding rectangle and maximize the performance of TPR-tree .

Even though we did not compare IMORS with the hashing technique for moving objects [12] in the experiments, it may be worthwhile to mention it. This method employs a similar approach to IMORS. It however employs hashing methods for its underlying static data structure as IMORS uses R*-tree. And IMORS restricts the searching space as does R-tree, while the hashing technique simply decomposes the space without any restriction of movement. We expect that IMORS outperforms this hashing methods for this reason.

6 Conclusion

In many applications of moving objects, such as LBS, the fast update operation of their positions becomes one of critical requirements in spatiotemporal database management systems. Although several indexing methods have been proposed for moving objects, they often overlook the overheads of update operations which are quite frequent especially in LBS. In this paper, we focus on relieving update overheads and provide efficient mechanisms in both searching and updating.

The key idea of our method, named IMORS, is motivated by two observations. First we can improve the performance of updates by separating the data structure of IMORS into static and dynamic parts, and by reducing the operations concerning the dynamic parts during updates. Second, in reality most moving objects navigate on predefined road networks. This fact enables us to extract the static part from the index structure and render the update overheads light.

An analytic model is provided to tune the performance of our method. Even though IMORS is initially proposed to improve update performance, it also shows a significant improvement on range search performance. Experimental studies show that our method accomplishes above functionalities. From experimental results, we observe that the update cost is about 40% less in disk accesses than TPR-tree, and the timestamp range queries are much faster than TPR-tree; more than thirty times faster for small queries, and about four times faster for large queries.

We do not discuss on concurrency control problem in this paper, which may degrade the performance of updates as well. But we expect that IMORS could be incorporated in a concurrency control mechanism without great efforts, since our method simplifies the update mechanism by separating dynamic part of the index. In addition to concurrency control, it will be interesting to employ other indexing methods for road sectors other than R*-tree. Finally, the implementations of near-future query will be of our interest for next works.

7 Acknowledge

This work was supported by the Korea Science and Engineering Foundation(KOSEF) through the ”Indexing and query processing for moving with minimal update costs” project at the Advanced Information Technology Research Center(AITrc), by grant F01-2002-000-10014-0 from Korea Science and Engineering Foundation(KOSEF) and by the program for cultivating graduate students in regional strategic industry

References

[1] C. S. Jensen and S. Saltenis, Towards Increasingly Update Efficient Moving-Object Indexing, IEEE Data Engineering Bulletin 25(2), pp.35–40, 2002

[2] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, Indexing the Positions of Continuously Moving Objects, In *Proc. ACM SIGMOD*, pp.331–342, 2000.

[3] S. Saltenis and C. S. Jensen, Indexing of Moving Objects for Location-Based Services, In *Proc. ICDE*, 2002.

[4] D. Pfoser, C. S. Jensen, and Y. Theodoridis, Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. VLDB*, pp.395–406, 2000

[5] D. Pfoser, Indexing the Trajectories of Moving Objects. IEEE Data Engineering Bulletin 25(2) pp.3–9, 2002

[6] G. Kollios, D. Gunopulos, V. J. Tsotras, On Indexing Mobile Objects, In *Proc. ACM PODS*, pp.261–272, 1999.

[7] P. K. Agarwal, L. Arge, and J. Erickson, Indexing Moving Points, In *Proc. ACM PODS*, pp.175–186, 2000

[8] D. Papadopoulos, G. Kollios, D. Gunopulos, and V. J. Tsotras, Indexing Mobile Objects on the Plane. In *Proc. DEXA Workshops*, pp.693–697, 2002

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. SIGMOD Conference*, pp.322–331, 1990

[10] G. Trajcevski, O. Wolfson, B. Xu, and P. Nelson, Real-Time Traffic Updates in Moving Objects Databases, In *Proc. 5th International Workshop on Mobility in Databases and Distributed Systems*, pp.2–6, 2002

[11] Y. Theodoridis and T. Sellis, A Model for the Prediction of R-tree Performance, In *Proc. ACM PODS*, pp.161–171

[12] Z. Song and N. Roussopoulos, Hashing Moving Objects, In *Proc. Mobile Data Management Conf.* pp.161–172, 2001

[13] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, Moving Objects Databases: Issues and Solutions, In *Proc. SSDBM*, pp.111–122, 1998

[14] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha, Updating and Querying Databases that Track Mobile Units, Distributed and Parallel Database Journal 7(3), pp.257–283, 1999

[15] T. Brinkhoff, Generating Network-Based Moving Objects, In *Proc. SSDBM*, pp.253–255, 2000

[16] T. Brinkhoff, A Framework for Generating Network-Based Moving Objects, GeoInformatica 6(2) pp.153–180, 2002

[17] I. Kamel and C. Faloutsos, On Packing R-trees, In *Proc. ACM CIKM*, pp.490–499, 1993

[18] B.-U. Pagel, H. W. Six, H. Toben, and P. Widmayer, Towards an Analysis of Range Query Performance in Spatial Data Structures, In *Proc. ACM PODS*, pp.214–221, 1993