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,tkim}@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 a fast update mechanism of spatial database system. Conventional indexing methods have difficulties in providing efficient updating mechanism. 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 mechanism but also the performance of query processing. The only constraint of our method is that the moving objects should be on roads, but we expect that most applications of moving objects satisfy this condition. We provide a performance analysis to improve indexing and update performance of our method. From experiments, we observe that it shows a very good indexing and query processing performance as well as fast updates.

1. Introduction

Moving objects are being widely treated in several application domains, such as LBS (Location Based Service). There are however some serious problems in storing and handling them in a spatiotemporal database system. One of the problems is related with a performance issue. Many moving objects frequently change their positions and submit update requests to database system. This results in a considerable overhead on the system. 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 spatial database systems are involved in processing it. They include 1) storage system, 2) indexing mechanism and 3) transaction management for concurrency control and recovery. In this paper, we will focus on how to reduce the update cost of index. We can significantly improve the update performance by providing an efficient indexing method.

While a number of indexing methods have been proposed for moving objects, they aim to improve the performance of indexing and query processing rather than to reduce update cost. For example, suppose that we have a million moving objects and each of them gives an update request on their position for each minute. Then each update must be handled within 0.06 msec, which is in fact a very difficult requirement to satisfy.

In this paper, we will propose an indexing method for current positions of moving objects that significantly reduces the update cost. Our approach is motivated by two observations. First, any index data 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. The second observation is that most moving objects are on road networks in real world, which is called 1.5 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 work and the motivation of our work in the next section. Section 3 describes our indexing data structure and algorithms for moving objects. An analysis on the performance of our method is given in section 4. We can improve the performance by this analytic cost model. Section 5 reports on performance experiments and comparison with other indexing method. We conclude our paper in section 6.

2. Related Work and Motivations

Several methods have been proposed for indexing moving objects. They are classified into two categories; indexing methods for the current and future positions of moving objects [2][3][6][7] [8][14], and methods for the past positions or trajectories [4][5]. Each of them requires quite
different functions and approaches. In this paper, we focus on the former category.

Most of indexing methods for current positions treat point moving objects and are based on extensions of existing spatial indexing methods. For example, **TPR-tree** (Time-Parameterized R-tree) [2] is an extension of R*-tree with some additional considerations on the movement and anticipated future position of moving objects. It employs \( t \)-**parameterized bounding rectangles** to reflect on the near-future positions of moving objects instead of simple MBR in R*-tree. An improvement of TPR-tree, called \( R_{\text{EXP}} \)-tree [3], has been proposed to compute better \( t \)-parameterized bounding rectangles.

Data transformation techniques are introduced to allow the use existing spatial index, 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, and [7], and Kollios et. al. employ the dual transformation from a line \( x = x_{\text{ref}} + v(t - t_{\text{ref}}) \) to a point \((x_{\text{ref}}, v)\) [6].

These indexing methods support various types of queries moving objects. In particular, they provide the query processing on anticipated positions, which does not exist in spatial databases [16]. For example, a nearest neighbor query processing technique is proposed for moving objects with TPR-tree using \( t \)-parameterized bounding rectangles [13]. And Song et. al. [14] proposed several efficient algorithms to process nearest neighbor query for moving objects as well.

While these indexing methods focus on improving the performance of query processing, they do not provide an efficient updating mechanism. Even though we can reduce the amount of updates by profiting from the dynamic property of moving objects [2], the explicit updates of positions are still an important burden of systems in most applications. An analysis on the update cost is conducted by Wolfson et. al. [17]

Our work aims to provide a fast updating mechanism and an indexing method. Our approach is based on two observations. First a sophisticated indexing mechanism may render the update cost expensive and a clear separation between the static and dynamic parts in index simplifies the update mechanism. In this paper, we will propose an indexing method of this approach. A similar method has been proposed by Song and Roussopoulos [15], which employs a hashing method as its data structure. An update does not take place until an object moves into a new bucket, and therefore the update cost is greatly reduced. We will briefly compare it with our method in section 5.

The second observation is that moving objects almost stay in road networks. For example, vehicles are on streets except for very special cases. But this fact allows us not only to reduce the uncertainty of the positional data but also to greatly simplify a lot the indexing. We will consider road networks as the static part of indexing data structure for moving objects. Some efforts have been done for modelling [11] or generating benchmark data of moving objects on road networks. [19]

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 such a moving object unless it is too far from a road.

As mentioned in previous section, we should reduce the part of index data structure dynamically changing upon update requests. And the above assumption provides a very important solution to reduce the dynamic part or maximize the static part of index. Since the information about the geometry and connectivity of road networks rarely changes, our indexing method 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 structure of IMORS, which consists of two parts as mentioned in the previous section. First the static part contains only a R*-tree built on road sectors. Note that a set of road sectors are contained in each R*-tree leaf node as its entries. Second the dynamic part has two principal data structures: **Road Sector Blocks** \( B_{\text{RS}} \) in short), and **Data Blocks for Moving Objects** \( B_{\text{data}} \). Road sector means a non-intersecting piece of a road bounded by two intersections. The shape of a road sector is obviously of polygon.

The R*-tree used in IMORS is similar with other ordinary R*-trees except that its entries are of road sectors rather than points or polygons. Unlike points or polygons, polygons 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 small ones to reduce the dead space, but such a division leads to an increase of entries in R*-tree. A detail analysis will be given in section 4. But for the reason of simplicity, we assume that the maximum length of road sector is bounded by the length of non-intersected sector.
Each road sector entry on leaf nodes of R*-tree points to a block of \( B_{RS} \), 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.

### Figure 1. Overall Data Structure of IMORS

**Figure 2. Data Structure of road sector Bucket**

Once an update on positional data of a moving object is submitted, the coordinates data in \( B_{RS} \) and \( B_{data} \) must be updated and the membership relation between the moving object and \( B_{RS} \) 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 \( B_{RS} \) 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.

### 3.2 Insertion and Deletion

Inserting a new moving object into IMORS is carried out by three steps. First, this new object is registered in \( B_{data} \) with its coordinates and other attributes. For the second step, we search the road sector via R*-tree, where the moving object is placed and register it on the road sector block. And finally, the bidirectional pointers are written on road sector block and data block. The detail 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 \( B_{RS} \). This may happen when the road sector is congested by moving objects and degrade the update and indexing performance. Deletions in IMORS are performed in the same way by removing a moving object from \( B_{RS} \) and \( B_{data} \). We omit the detail algorithm in this paper.

### 3.3 Update

While insertion and deletion are relatively simple, updates of positional data should be handled with care. Figure 5 shows the overall procedure of an update operation.

Suppose that a new position of a moving object with its object identifier is reported to the system. We retrieve the record from the data block by its object identifier 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 road sector block. In order to determine the new road sector block, we lookup the R-tree and get the road sec-
Update Algorithm

Input \( m \): moving object with new position, \( R \): R*-tree

Begin Algorithm

\[
\text{Update\ Position}(m.\text{oid}, \ m.\text{position}); \\
b_{RS} \leftarrow \text{Get\ RoadSectorBlock}(m.\text{oid}); \\
\text{If} \ ( \text{Is\ on\ Road\ Sector}(b_{RS}.\text{Sector}, \ m.\text{position}) \ == \ \text{NO} ) \ {\{ \\
\text{Remove\ from\ Road\ Sector\ Block}(R, \ m.\text{oid}); \\
b_{RS} \leftarrow \text{Search\ R*-tree}(R, \ m.\text{position}); \\
b_{ptr} \leftarrow \text{Register\ on\ RoadSectorBlock}(b_{RS}, \ m.\text{oid}, \ m.\text{position}); \\
\text{Register\ Block\ Pointer\ on\ DataBlock}(m.\text{oid}, b_{ptr}); \\
\} }
\]

End Update Algorithm

Figure 4. Update Algorithm of IMORS

Figure 5. Update Procedure of IMORS

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*-tre. And each moving object in the road sectors is examined if it is contained within the query region. Figure 6 illustrates this procedure and its 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 or 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 of high priority, we can estimate the future positions even more accurately. It will be interesting to implement this approach in IMORS for future work.

Figure 6. Example of Query Processing

4 Analysis of Performance

In this section, we present an analysis on performance of IMORS for updating and query processing. Analytic cost
Containment Region Query Processing
Input \( W \): Query Region, \( R \): R*-tree
Output \( M \): Set of Moving Objects

Begin Algorithm
\[
B_{RS} \leftarrow \text{Search.Intersecting.Sectors.R*-tree}(R, W); \\
// \quad B_{RS} \text{ denotes a set of road sector blocks} \\
// \quad b_{RS} \text{ denotes a road sector block} \\
M \leftarrow \{\}; \\
\text{For each } b_{RS} \text{ in } B_{RS} \\
\quad \text{For each } m \text{ in } b_{RS}.M \\
\quad \quad // \quad b_{RS}.M \text{ is the set of moving objects contained by } b_{RS} \\
\quad \quad \text{If } m \text{ is contained by } W, \text{ then } M \leftarrow M \cup \{m\}; \\
\]
return \( M \)
End Containment Region Query Processing

Figure 7. Query Processing Algorithm of IMORS

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 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,
\[
C_Q = C_{RT} + C_{RS} + C_{DB}. \quad (1)
\]
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. [21] and Kamel et. al. [20] have proposed a simple and accurate cost model of R-tree describing the relationship between object size and the cost of leaf node access,
\[
C_{RT} = \sum_{i=1}^{n_s}(s_{x,i} + q_x)(s_{y,i} + q_y) \quad (2)
\]
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
Table 1. Symbol table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_Q)</td>
<td>query processing cost</td>
</tr>
<tr>
<td>(C_U)</td>
<td>update cost</td>
</tr>
<tr>
<td>(C_{RT})</td>
<td>R*-tree access cost</td>
</tr>
<tr>
<td>(C_{RS})</td>
<td>cost for road sector accesses</td>
</tr>
<tr>
<td>(C_{DB})</td>
<td>cost for data block accesses</td>
</tr>
<tr>
<td>(C_{NewRS})</td>
<td>cost for new road sector access</td>
</tr>
<tr>
<td>(l_s)</td>
<td>average length of a road sector</td>
</tr>
<tr>
<td>(n_s)</td>
<td>number of road sectors</td>
</tr>
<tr>
<td>(n_{opt})</td>
<td>optimal number of road sector</td>
</tr>
<tr>
<td>(s_{x,i})</td>
<td>x-axis length of i-th road sector</td>
</tr>
<tr>
<td>(s_{y,i})</td>
<td>y-axis length of i-th road sector</td>
</tr>
<tr>
<td>(q_x)</td>
<td>x-axis length of query</td>
</tr>
<tr>
<td>(q_y)</td>
<td>y-axis length of query</td>
</tr>
<tr>
<td>(Bf)</td>
<td>blocking factor of data block</td>
</tr>
<tr>
<td>(L)</td>
<td>total length of all road sectors</td>
</tr>
<tr>
<td>(L_x)</td>
<td>sum of lengths of all road sectors</td>
</tr>
<tr>
<td>(L_y)</td>
<td>sum of length of all road sectors</td>
</tr>
<tr>
<td>(f)</td>
<td>update frequency</td>
</tr>
<tr>
<td>(v)</td>
<td>average velocity of moving objects</td>
</tr>
<tr>
<td>(\rho)</td>
<td>probability of changing road sector</td>
</tr>
</tbody>
</table>

Proposed by Theodoridis and Sellis [12] 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 R*-tree. But it is complicated and we consider only the

\[ L = \sum_{i=1}^{n} s_{x,i} \]  

\[ L_y = \sum_{i=1}^{n} s_{y,i} \]  

If \(s_x\) and \(s_y\) are the average extents of road sectors along x-axis, y-axis, 

\[ s_{x,i} = s_x \]  

\[ s_{y,i} = s_y \]  

for \(i = 1, \ldots, n_s\). Therefore, eq.(2) becomes

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

\[ = n_s \left( \frac{L_x}{n_s} + q_x \right) \left( \frac{L_y}{n_s} + q_y \right) \]  

\[ = \frac{L_x}{n_s} \frac{L_y}{n_s} + n_s q_x q_y + L_x q_y + L_y q_x \]  

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

\[ n_{opt} = \sqrt{\frac{L_x}{q_x q_y} \frac{L_y}{n_s}} \]  

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

\[ n_s = \sqrt{\frac{L_x}{q_x q_y} \frac{L_y}{n_s}} \]  

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

\[ l_s = \frac{L}{n_s} \]  

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 5 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 \(C_{NewRS}\). Note that the cost for updating bidirectional pointers in figure 5 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,

\[ C_U = C_{DB} + C_{RS} + \rho(C_{RT} + C_{NewRS}) \]  

where \(\rho\) 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 \((C_{DB}, C_{RS}, C_{NewRS} = 1)\). 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 \(h = \log \frac{n_s}{Bf} \frac{L}{\frac{L_x}{n_s}}\).

The most significant factor in eq.(6) is the probability \(\rho\). 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 $\rho$ and these parameters are depicted by figure 9, and the probability is given as

$$\rho = \min(1, \frac{v}{l_s f})$$  \hspace{1cm} (7)

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

$$C_U(n_{sect}) = C_{DB} + C_{RS} + \rho(C_{RT} + C_{NewRS})$$
$$= 2 + \min(1, \frac{v}{l_s f})(\log_{Bf} \frac{n_s}{Bf} + 1)$$
$$= 2 + \min(1, \frac{n_s v}{L})(\log_{Bf} \frac{n_s}{Bf} + 1)$$ \hspace{1cm} (8)

or

$$C_U(l_s) = 2 + \min(1, \frac{v}{l_s f})(\log_{Bf} \frac{L}{l_s Bf} + 1)$$ \hspace{1cm} (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 [18] [19]. 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 Oldenborg in Germany, used in the experiments, where the extent of the map is $23854 \times 30851$. They request updates on the positional data every time unit during the simulation. We configured the velocity parameter of this tool as medium.

Figure 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, 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 fil-
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

Figure 13. IMORS Avg. disk I/O per query by Road Sector Length

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

The query performance is significantly improved 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 [15] 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.
Figure 14. Query Performance Comparison between IMORS and TPR-tree Ratio avg. disk I/O per query by Query Size

6 Conclusion

In many application domains of moving objects, such as LBS, the fast update of their positional data becomes a critical requirement of database management systems. Although several indexing methods have been proposed for moving objects, they focus on efficient query processing techniques rather than fast updates.

In this paper, we propose a new indexing method, which provides a fast update mechanism as well as efficient query processing. The key idea of our method, named IMORS, is motivated by two observations. First we can improve the performance of updates by separating the index structure into static and dynamic parts, and reducing the operations concerning the dynamic parts during updates. Second, most moving objects follow on predefined road networks. This enables us to extract the static part from the index structure and render the updates light.

An analytic model is provided to tune the performance of our method. Even though IMORS has initially focused on the improvement of update performance, it shows a significant improvement of query processing performance as well. Experiments show that our method gives fast performances for updating, and very good performances of query processing as well. From our experiments we observed that the update mechanism is about twice faster than TPR-tree, and the query processing is much faster than TPR-tree; more than thirty times faster for small queries, and about five times faster for large queries.

We did 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 a great effort, 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 implementation of anticipation query processing will be of our interest for future work.

References