Indexing Fast Moving Objects for KNN Queries Based on Nearest Landmarks

Dan Lin[1] Rui Zhang[1] Aoying Zhou[2]

[1]Department of Computer Science
The National University of Singapore, Singapore
{*lindan, zhangru1*}*@comp.nus.edu.sg*

[2]Department of Computer Science and Engineering
Fudan University, China
ayzhou@fudan.edu.cn

Abstract

With the rapid advancements in positioning technologies such as the Global Positioning System (GPS) and wireless communications, the tracking of continuously moving objects has become more convenient. However, this development poses new challenges to database technology since maintaining up-to-date information regarding the location of moving objects incurs an enormous amount of updates. Existing indexes can no longer keep up with the high update rate while providing speedy retrieval at the same time. This study aims to improve k nearest neighbor (kNN) query performance while reducing update costs. Our approach is based on an important observation that queries usually occur around certain places or spatial landmarks of interest, called reference points. We propose the Reference-Point-based tree (RP-tree), which is a two-layer index structure that indexes moving objects according to reference

points. Experimental results show that the RP-tree achieves significant improvement over the TPR-tree.

Key Words: Moving object,Index, Nearest neighbor, Query.

1 Introduction

Rapid advancements in positioning technologies, such as the Global Positioning System (GPS) and wireless communications, have enabled the tracking of continuously moving objects. Many applications can benefit from the proliferation of such tracking technologies. For example, in a transportation system, via tracking vehicles, congestion can be alleviated by diverting some vehicles to alternative routes, and taxis can be dispatched quickly to passengers. In order to keep track and manage a large number of moving objects, the locations of moving objects are maintained in databases; and the efficient processing of queries on databases of moving objects has become an important problem. To avoid the problem of having to constantly update the location of objects as they move, moving objects are typically modelled as functions of time, which are updated only when there are significant changes to the parameters of the functions.

An important query type in moving objects databases is the *k nearest neighbor* (kNN) query, which is to find the k objects (from among the moving objects in an input dataset) that are nearest to a given query location at a given future time t. More formally, the kNN problem can be defined as follows. Consider a set of moving objects $O = \{o_1, o_2, \cdots, o_n\}$. Given a query location q, a future time t (which is equal to or greater than the current time), and a positive integer k, the kNN query computes the subset of k objects $O' \subseteq O$ that are nearest to location q at time t. Note that the kNN query in moving objects databases is slightly different from the conventional kNN query for static data due to the consideration of the time parameter.

Of the few structures that have been proposed for moving objects so far, the TPR-tree [13] is the only practical spatial-temporal index for the predictive query described above. However, we observe that the TPR-tree is still inefficient in real applications, where moving objects are not uniformly distributed, but are instead usually assembled around locations of interest, such as cities, commercial centers, traffic junctions, etc. An example based on the Singapore map [1] is shown in Figure 1, where the internal solid lines are the motorways, RPs are the locations of interest selected from real landmarks, and shaded regions denote

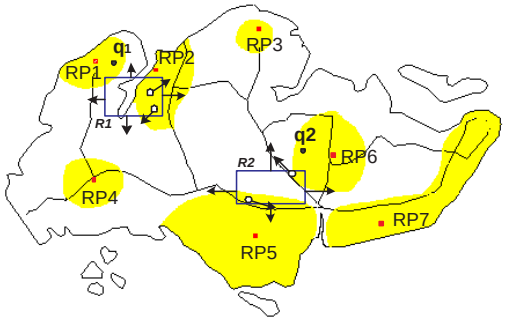
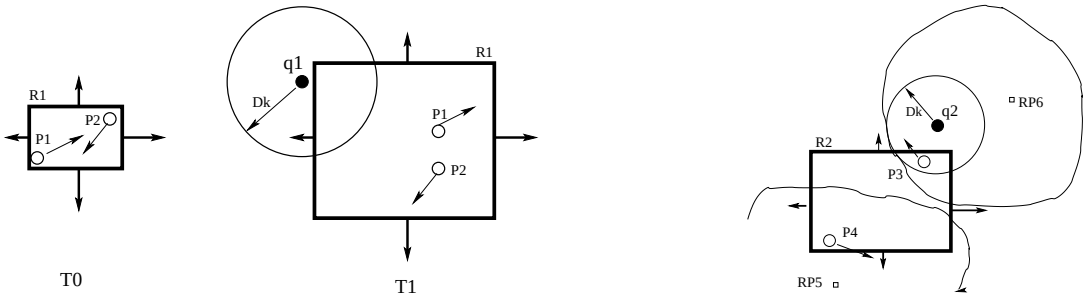


Figure 1. An example

high vehicle density. For convenience, we shall refer to a location of interest as a *reference point*.



(a) Query q1 (b) Query q2

Figure 2. Unnecessary search in the TPR-tree

Figure 1 depicts the distribution of vehicles. If we use the TPR-tree to index vehicles in this map, some unnecessary search may be executed due to the minimum bounding rectangles (MBRs) enlarged over time or the MBRs across neighborhoods of different reference points. Let us look at two examples by zooming in on two enlarged MBRs, R_1 and R_2 at timestamp T_1 in Figure 1. The detailed pictures are shown in Figure 2. At creation time T_0, MBR R_1 should tightly bound objects p_1 and p_2; at a later timestamp T_1, R_1 grows as shown in Figure 2(a). For the query q_1 at T_1, R_1 has to be searched due to the overlap with the query circle of q_1 (Dk is the distance between the query point and its k'th nearest neighbor) although there is no object in the overlapping region. Then, consider R_2 (as shown in Figure 2(b)), which bounds objects p_3 and p_4 in the neighborhood of RP_5 and RP_6 respectively. For the query q_2 in the neighborhood of RP_6, whose k nearest neighbors usually belong to the same neighborhood due to the high density of vehicles near the reference point, p_4 is almost unlikely to be the answer for q_2 but still has to be checked by the TPR-tree. In this case, if we index p_3 and p_4 separately according to the reference points RP_6 and RP_5,

we can save some unnecessary search. Therefore, our motivation is to avoid such wasteful operations so as to speed up query processing. We propose a new index structure: the Reference-Point-based tree or RP-tree, in which we partition the whole space into subspaces, and attach moving objects to their nearest reference points.

Besides the proposed RP-tree, we also consider the concurrency control scheme in our algorithms as multi-user environments are popular nowadays and good ability to parallelize has become an important requirement of database applications. Furthermore, we demonstrate the effectiveness of our new indexing method via a comprehensive performance study which shows that the RP-tree outperforms the TPR-tree considerably for both search and update.

The rest of the paper is organized as follows: in Section 2, we review the related work; in Section 3, we describe the structure of the RP-tree and our algorithms; Section 4 presents a cost model to estimate the disk page accesses of the kNN search. Section 5 reports the performance experiments. Finally, Section 6 concludes with directions for future work.

2 Related Work

Traditional indexes [6] for multi-dimensional databases, such as the R-tree [7] and its variants: R^+-trees [14], R*-trees [3], SS-trees [20] etc., have been designed mainly to speed up retrieval in applications where queries are relatively much more frequent than updates. However, this is invalid in applications for managing moving objects, where the workload is characterized by a heavy load of index updates and frequent queries. For indexing moving objects, some new index structures have been proposed recently. A detailed survey can be found in [11]. These index structures can be classified into two main categories: those indexing the historical locations of spatial objects and those indexing the current locations of spatial objects. Our approach falls into the latter one.

In the first category, the historical movements of objects are represented by their trajectories. Pfoser et al. [12] proposed the Spatio-Temporal R-tree (STR-tree) and Trajectory-Bundle tree (TB-tree). The STR-tree organizes line segments not only according to spatial properties, but also by attempting to group the segments according to the trajectories they belong to. The TB-tree aims only for trajectory preservation and leaves other spatial properties aside, thus performing more efficiently than the STR-tree. Another example of this category is the Multi-version 3D R-tree (MV3R-tree) [15], which is a combination of

multi-version B-trees and 3D R-trees. However, such a database would be very large as it has to capture the great deal of information generated by objects that are moving all the time. Hence, the critical problem is to decide what are good historical data and how to store them efficiently.

The structures in the second category either index the locations of moving objects or use functions to approximate movement. The Lazy Update R-tree (LUR-tree)[10] reduces update cost by ignoring deletions of objects that do not move out from the current minimum bounding rectangle (MBR). [18] introduced the issue of indexing moving objects to query their present and future positions by using the time function, and proposed the PMR-Quadtree. A hybrid tree structure called Q+Rtree [21] differentiates fast moving objects from quasi-static objects and stores them in the Quad-tree and the R*-tree respectively. A notable index for moving objects is the Time-Parameterized R-tree (TPR-tree) [13]. Due to its importance to our proposed RP-tree, we shall give more details of the TPR-tree in the next paragraph. Algorithms for both nearest neighbor queries and reverse nearest neighbor queries on moving objects have been proposed based on the TPR-tree [4]. Nearest neighbor queries are also studied in [2, 8]. The TPR*-tree [16] improved the TPR-tree by employing a new set of insertion and deletion algorithms.

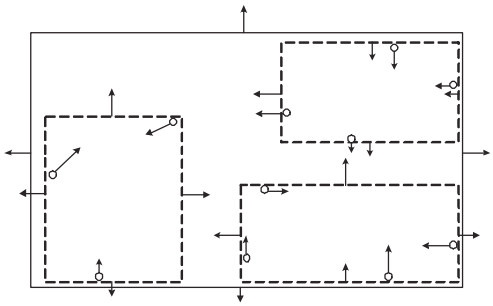


Figure 3. An example of time-parametric MBRs in a TPR-tree

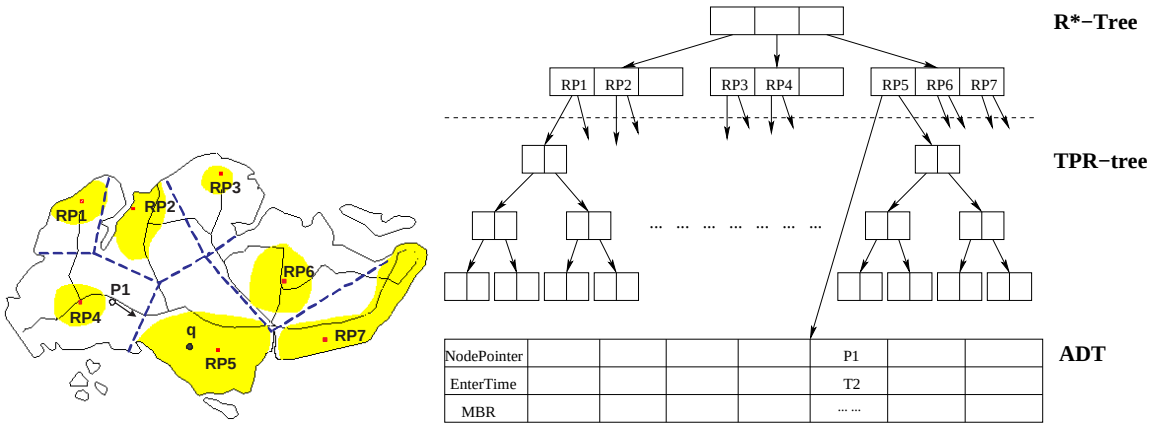
The TPR-tree is a variant of the R*-tree, in which the velocity of the moving objects are stored as well as its position at certain time. The current location of a moving point is calculated according to that position, velocity and the time elapsed. The coordinates of the bounding rectangles are also functions of time, which means that in each dimension, the lower bound of an MBR is set to move with the minimum velocity of the enclosed objects, while the upper bound is set to move with their maximum velocity (as shown in Figure 3). This ensures that the bounding rectangles are indeed bounding at all time considered. But conservative bounding rectangles never shrink, and are minimum only at some point of time. As time passes, the enlarged MBRs will overlap and adversely affect query performance. Therefore, there is a need

to adjust the bounding rectangles (the so-called “tightening” of the rectangles), and it is desirable to make the adjustment every time any of the moving objects or rectangles that they bound are updated. While the tightening of bounding rectangles increases query performance, it negatively affects update performance, which is a also very important issue.

3 The RP-tree

In this section, we present our proposed index structure: the Reference-Point-based tree (RP-tree), which facilitates the kNN query of moving objects and also reduce the update cost, in both single-user and multi-user environments. Then we give the update and query algorithms.

3.1 The Structure of the RP-tree



(a) Data space (b) The RP-tree

Figure 4. Overall representation

Our approach is motivated by the observation that moving objects are not uniformly distributed in the real world, but assemble around certain places (like shopping malls, commercial centers, etc.) that serve as sources and destinations of moving objects. Queries often follow the distribution of moving objects as people tend to be more interested in places where activities are frequently carried out. We refer to these locations of interest as *reference points* (*RP* for short).

In our algorithm, we treat reference points as pre-known data, which can be obtained by either clustering objects in the datasets or selecting places of interest in the real world. Given these reference points, we

use the Voronoi diagram to partition the entire space into disjoint subspaces (as shown in Figure 4(a), broken lines). Each subspace is called a *control area* and it has the following important property: an object falling into the *control area* of a reference point is closer to this reference point than to any other reference points. We now construct the RP-tree, which is a general tree structure comprising two layers. The upper layer indexes the locations of all reference points with their control areas as attributes. Note that such information is static. Each reference point exploits a second layer index to manipulate all moving objects inside its control area, and an *auxiliary data table* (ADT) to maintain the information of objects from other control areas that may enter this control area at some future time. In this paper, we choose the R*-tree as the upper layer index and the TPR-tree (which has been described in the previous section) as the lower layer index (see Figure 4(b)). Other structures can also be used if necessary.

The ADT is a table of records, which are in the form $\langle node\_pointer, enter\_time, MBR \rangle$. *node_pointer* points to a disk page (or node) that contains an object which may enter this control area at some future time *enter_time*. *MBR* is the time-parameterized minimum bounding rectangle of the node that the $node\_pointer$ points to. Figure 4(a) gives an example. At time T_0, a car p_1 in the control area of RP_4 travels towards RP_5 along a highway. If the car kept this velocity, it would enter the control area of RP_5 at time T_2. If T_2 is before the next update time, a pointer to the leaf node containing p_1, the entering time T_2, and the MBR of the leaf node is stored in the ADT of RP_5. When a query q in the control area of RP_5 is issued at time T_1 for kNN at the future time T_3 ($T_3 > T_2 > T_1 > T_0$), we check the ADT of RP_5 to see which objects from other control areas may come in. ADTs is organized as sequential files. We initially allocate certain space for each RP. The size of the ADT is enlarged when necessary. Typically, the ADT is one to two pages large as observed from our experiments.

There are several advantages of the RP-tree. First, different from the TPR-tree, which uses a single hierarchical structure to organize the entire set of moving objects and thus requires searching a large index tree to process kNN queries, the RP-tree requires searching only a few (often, only one) smaller "local" TPR-trees. This is because queries often concentrate on the reference points, and the answers to queries most likely exist in the control area of the reference points. Second, indexing objects separately in each control area may save unnecessary search occurring along the borders of the control area. The partition largely avoids the inefficient cases in the TPR-tree as described in Figure 2. Moreover, local TPR-trees are relatively independent of each other. This allows more freedom for concurrent operations.

3.2 Index Creation

Algorithm Insertion: (o)
/* Input: *o* is the object to be inserted */
1. Identify the nearest reference point RP_i of *o* in the upper level of the RP-tree
2. Insert *o* to the TPR-tree of the RP_i
3. If there is a split in the leafnode
4. For each object o_i in the two new leafnodes
5. Invoke Adjust_ADT(o_i)
6. Else
7. Invoke Adjust_ADT(*o*)

End Insertion

Algorithm Adjust_ADT: (*o*)
1. For all RPs whose control area that *o* may pass by according to its current location and velocity within the maximum update interval
2. $new\_entertime$ is the time when *o* will enter the control area of RP_i
3. If there exists a pointer in one of the tuples of ADT of RP_i pointing to the leaf node containing *o*
4. Compare the $new\_entertime$ with the $enter\_time$ in that tuple and modify $enter\_time$ to the earlier one
5. Update the MBR in that tuple
6. Else
7. Create a new tuple $\langle new\_nodepointer, new\_entertime, new\_mbr \rangle$ /*$new\_nodepointer$ points to the leaf node containing *o*, new_mbr is the corresponding MBR of the leaf node */
8. Add the new tuple to the ADT of RP_i

End Adjust_ADT

Figure 5. Algorithm of insertion

Figure 5 presents the essence of the insertion algorithm. Given a new object o, we first find the nearest RP (reference point) of o in the upper level of the RP-tree and insert object o to the corresponding TPR-tree. Then, we adjust the ADTs of RPs whose control area o may pass by before its next update. The *enter_time* in the ADT should be maintained as the earliest time when the leaf node overlaps with the control area. When a leaf node splits, the affected ADTs are the original ones and those introduced by the newly inserted object. We would then need to recalculate their *enter_time* and modify the pointers.

It is worth mentioning that we generally predict the future trajectory of one object within the maximum update interval for the ADT. That means an object that must have been updated before the query time (not query starting time) will not be treated as an answer. For example, an object o is inserted at time 0 and the maximum update interval is 120. Suppose a query is issued at time 90 and has the query time 140, and the

object o has not been updated yet. We do not need to retrieve object o since it will be definitely updated before 120, which means we can not use its current moving function to predict its position after time 120. We believe that this rule is rational because if we already know that an object has been updated, it would be meaningless to use its outdated data for predictions. In the experiments, for the fairness of comparison, this constraint is also applied to the TPR-tree.

To remove an object *o*, there are two steps. First, we remove *o* from the TPR-tree of its nearest RP. Second, we check the ADTs containing the pointers to the leaf node *o*, and modify the *enter_time* in the corresponding tuple or delete the tuple from the ADT. Note that when a leaf node underflows, we first load in the related ADTs and delete the tuples belonging to this leaf node, then reinsert entries in the leaf node to the TPR-tree, adjust these ADTs in the mean time, and finally write in the modified ADTs.

3.3 KNN Query

Figure 6 presents the kNN search algorithm. Given a query point *q*, we first identify its nearest RP by searching the upper level of the RP-tree. Then we search the TPR-tree of this RP to get *k* candidate nearest neighbors by the search algorithm proposed in [5]. Next, we search the ADT of this RP and modify the Dk (the distance between query point q and its k'th nearest neighbor). If the query circle (centered at q, with radius equal to Dk) is wholly inside the control area of this RP, the algorithm terminates. Otherwise, we perform an enlarged search: (i) find the RPs whose control areas overlap with the query circle from the upper level of the RP-tree; (ii) search their ADTs (also using *mindist* as a pruning metric) and local TPR-trees, adjust the radius of the query circle (i.e., Dk); and (iii) repeat (i) (ii) until the query circle is bounded in the sum of control areas of the accessed RPs. Note that during the enlarged search, some leaf nodes which are stored in both ADTs and local TPR-trees are only checked once.

3.4 Concurrency Control Scheme

In order to support multi-user concurrent access to the database via the indexes, the issue of concurrency control scheme must be addressed. A naive solution to this problem is to have the whole tree locked when one operation is executed, which is very inefficient. An efficient algorithm is the R-link tree proposed in [9], which introduces few lock-coupling and guarantees that insertion can be performed without blocking search processes. R-link is a right-link, a pointer from every node to its right sibling at the same level.

Algorithm KnnQuery: (*q*, *k*, *t*)
/* Input: To find *k* nearest neighbors of *q* at future time *t* */
1. Identify the nearest reference point RP_i of the *q* in the upper level of the RP-tree
2. Search in the TPR-tree of RP_i
3. Store the *k* candidate nearest neighbors in a list *L*
4. D_{qc} is the distance between *q* and its nearest boundary of the control area of RP_i
5. D_{qk} is the distance between *q* and the *k*-th candidate nearest neighbor
6. Invoke Search_ADT(q, k, t, RP_i, D_{qk}, L)
7. If ($D_{qc} < D_{qk}$)
8. Invoke Enlarged_Search(q, k, t, RP_i, D_{qk}, L)
9. Return *L*

End KnnQuery

Algorithm Search_ADT: (q, k, t, RP_i, D_{qk}, L**)**
1. *adt* = the pointer to the first tuple in the ADT of RP_i
2. While (*adt* is not null)
3. $mindist$ is the minimum distance between *q* and $adt.MBR$
4. If ($adt.enter\_time < t$ and $mindist < D_{qk}$)
5. Search the leaf node that $adt \rightarrow NodePointer$ points to
6. Modify *L* that stores knn candidates and adjust D_{qk}
7. Return (D_{qk}, L)

End Search_ADT

Algorithm Enlarged_Search: (q, k, t, RP_i, D_{qk}, L**)**
1. Sort RPs according to the minimum distance between *q* and boundaries of control area of RP in an ascending order
2. While (D_{qk} exceeds the sum of control area of searched RPs)
3. Pick the RP_i according to the sorted order
4. Invoke Search_ADT(q, k, t, RP_i, L)
5. Search in the TPR-tree of RP_i
6. Adjust L and D_{qk}
7. Return (D_{qk}, L)

End Enlarged_Search

Figure 6. Algorithm of kNN query

Besides right links, a unique logical sequence number (LSN) is assigned to each node. Each entry in a node contains the LSN that it expects the child node to have. If a node has to be split, the new right sibling is assigned the old node's LSN and the old node receives a new LSN. A process traversing the tree can detect the split even if it has not been installed in the parent by comparing the expected LSN, as taken from the entry in the parent node, with the actual LSN. If the later is higher than the former, there was a split and the process moves right. When the process finally meets a node with expected LSN, it knows that this is the rightmost node split off the old node.

We employed this algorithm to the TPR-trees in the RP-tree. For ADT, we simply lock the tuple in the ADT when its tuple is being used. And the R*-tree in the upper level does not need any lock since it is a static structure.

4 Cost Model

In this section, we present a cost model to estimate the disk access cost of the kNN search of the RP-tree. [17] has proposed a method to estimate the kNN search cost for static databases, which uses a rectangle of equal volume to replace a query circle. Our cost model is based on the formulas in [17], but we need to modify these formulas to capture the features of a moving object database. We mainly study moving objects in two-dimensional space (the data space has unit length in each dimension), therefore the cost analysis is also confined to two-dimensional space. Generalization to higher dimensions is straightforward. Notations used in the analysis are summarized in Table 1.

Notation	Description
b	maximum capacity of a node
D_k	distance between the query point and its k'th nearest neighbor
f	average fan-out of a node
h	the height of a tree
IO	query cost measured as number of disk page accesses
k	number of nearest neighbors required
L	side length of a square
n_i	number of nodes at level i
N	number of objects in the dataset
N_r	number of reference points
P	probability
s_i	average extent of a level-i node of a hierarchical structure
T_q	predictive time length of a query
Vol	volume
$\Theta(q, r)$	the query circle with center q and radius r
$\Xi(R, d)$	the Minkowski region of a rectangle R with distance d

Table 1. Notations used in equations

4.1 The k'th Nearest Neighbor Distance

In [17], the k'th Nearest Neighbor Distance D_k is estimated by the following equation.

$$D_k = \frac{2}{C_v}\left[1 - \sqrt{1 - (\frac{k}{N})^{\frac{1}{d}}}\right] \tag{1}$$

where

$$C_v = \frac{\sqrt{\pi}}{[\Gamma(d/2+1)]^{\frac{1}{d}}},\ \Gamma(x+1) = x \cdot \Gamma(x), \Gamma(1) = 1, \Gamma(1/2) = \pi^{1/2}$$

and the d is dimensionality of the data space.

Here, we let $d = 2$ and get the following equation.

$$D_k = \frac{2}{\sqrt{\pi}}\left[1 - \sqrt{1 - (\frac{k}{N})^{\frac{1}{2}}}\right] \tag{2}$$

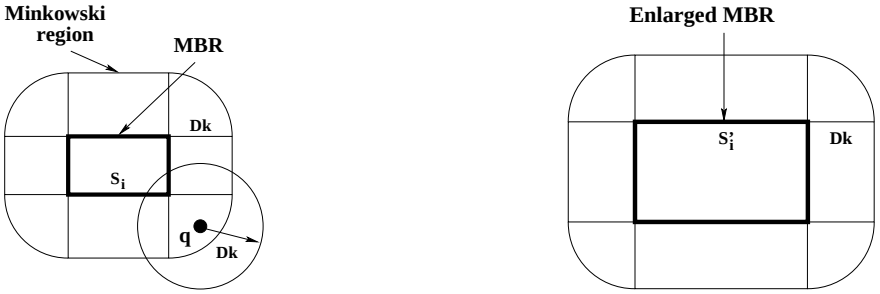
4.2 Disk Page Accesses with the R*-tree and the TPR-tree

The average disk page accesses of general queries on hierarchical structures are given by equation 3 [19]. Note that a disk page is a node, so page and node are used interchangeably in the sequel.

$$IO(N) = \sum_{i=0}^{h-1}(n_i \cdot P_{IO_i}) \tag{3}$$

h is the height of the tree (leaf nodes are at level 0). P_{IO_i} is the probability that a node at level i of the tree is accessed. n_i is the total number of nodes at level i of the tree. In turn, h and n_i can be estimated as $h = 1 + \lceil \log_f N/b \rceil$ and $n_i = N/f^{i+1}$ respectively, where b is the maximum capacity of a node, and f is the average node fanout (typically, $f = 69\% \cdot b$).

The number of node accesses is equal to the number of nodes whose MBRs intersect the query circle $\Theta(q, D_k)$. An MBR R intersects $\Theta(q, D_k)$, if and only if its Minkowski region $\Xi(R, D_k)$ contains q (see Figure 7(a)). In the unit data space U, the intersection probability equals the volume of $\Xi(R, D_k) \bigcap U$. To calculate this probability, we first need the average extent of a node (i.e., the extent of an MBR R) at level i. In the R*-tree, the average extent of a node is given by equation 4 [17]. In the TPR-tree, MBRs are time-parameterized, thus we consider the enlargement of the MBRs after a period of time by its maximum



(a) (b)

Figure 7. Minkowski region

and minimum velocity (see Figure 7(b) and equation 5).

$$s_i = \frac{1}{2^{[\log_2(n_i)]/2}} = \frac{1}{(\frac{N}{f^{i+1}})^{\frac{1}{2}}} \qquad (0 \le i \le h-1) \tag{4}$$

$$s_i^{'} = s_i + \Delta v \cdot t, \qquad \Delta v = v_{max} - v_{min} \tag{5}$$

In two-dimensional space, the Minkowski sum of the rectangle R with distance D_k is given by equation 6.

$$Vol(\Xi(R, D_k)) = s_R^2 + 4D_k \cdot s_R + \pi D_k^2 \tag{6}$$

When the Minkowski region is approximated by a square of the same volume [17], the side length of the square L_R is given by equation 7.

$$L_R = \sqrt{Vol(\Xi(R, D_k))} \tag{7}$$

Then, the probability of a node at level i of the tree being accessed is

$$P_{IO_i} = \begin{cases} \left(\frac{L_i - (L_i/2 + s_i/2)^2}{1 - s_i}\right)^2 & if \;\; L_i + s_i < 2 \;\; (0 \le i < h) \\ 1 & \text{otherwise.} \end{cases} \tag{8}$$

Combining equations 1-8, we obtain equation 9 to estimate the kNN search cost of the R*-tree. By replacing s_i and L_i with s_i' and L_i' respectively in equation 9, we obtain equation 10 to estimate the kNN search cost of the TPR-tree.

$$IO_{r*}(N,k) = \sum_{i=0}^{\log_f \frac{N}{b}} \left[\frac{N}{f^{i+1}} \cdot \left(\frac{L_i - (L_i/2 + s_i/2)^2}{1 - s_i} \right)^2 \right] \tag{9}$$

$$IO_{tpr}(N,k) = \int_0^{T_q} \sum_{i=0}^{\log_f \frac{N}{b}} \left[\frac{N}{f^{i+1}} \cdot (\frac{L_i^{'} - (L_i^{'}/2 + s_i^{'}/2)^2}{1 - s_i^{'}})^2 \right] dt \tag{10}$$

4.3 Disk Page Accesses with the RP-tree

In the RP-tree kNN search algorithm, we first locate the control area of the reference point nearest to the query point q. Second, we search the local TPR-tree of the reference point and the corresponding ADT. If the k'th candidate is outside the control area of the reference point, the enlarged search is executed.

The disk accesses to the RP-tree consist of two parts. One part comes from accesses to the R*-tree to search the nearest referent point of a given query point. The other part is caused by the accesses to MRBs referenced by local TPR-trees or ADTs. For each reference point whose control area overlaps with $\Theta(q, D_k)$, its local TRP-tree and ADT will be searched. However, it is difficult to estimate how many control areas will overlap, because the distribution of moving objects in reality is not uniform and hard to be described with a function. Therefore, we use the sampling strategy to estimate the number of control area that overlaps with $\Theta(q, D_k)$, *i.e.,* we test a number of queries that follow the same distribution as the data and obtain an average number of control areas being affected.

Finally, we summarize the number of disk page accesses for kNN queries in the RP-tree as follows.

$$IO_{rp}(N,k) \quad = \quad IO_{r*}(N_r, 1) + \sum_{accessed} [IO(local\ TPR-tree) + IO(leaf\ nodes\ in\ ADT)] \tag{11}$$

5 Performance Evaluation

In this section, we examine the efficiency of the RP-tree and evaluate the accuracy of the cost model. We compare the kNN query and update performance of the TPR-tree and RP-tree through extensive experiments.

All the experiments were conducted on a computer with 1.6G PentiumIV CPU and 256M memory. The disk page size was 4KB, and the resulting leaf node capacity was 204 entries for both the TPR-tree and the RP-tree. We used two kinds of datasets. One is based on a synthetic network. The other is based on real road network in Singapore (see Figure 1). The generations consist of three steps as follows:

First, we need to decide the reference points. For the synthetic dataset, some number of reference points were generated randomly, simulating locations of interest. For the real-network-based dataset, commercial centers are picked up as the reference points. Second, some moving objects were generated around the reference points, simulating vehicles moving within areas close to the locations of interest, with a relatively low speed varying from 0 to 6 (the extent of each dimension of the space was set as 5,000). These objects are called the *intra-objects*. Finally, some moving objects were generated in a network of two-way routes which connected the reference points. These objects simulated vehicles travelling on the motorways, with a relatively high speed varying from 0 to 10, and they are called the *inter-objects*.

In the experiments, we varied the number of objects in the datasets from 100K to 1M and the ratio between the number of inter-objects and the total number of objects from 0 to 1. The average interval between two successive updates of an object was set as 60 time units and the maximum interval was set as 120 time units. The distribution of the query points followed the same distribution of the moving objects, with predictive time ranging from 0 to 60 time units after the starting time.

Unless otherwise noted, the experiment results we present in the following are of datasets with 100K objects and the ratio between the number of inter-objects and total number of objects was 0.5; the number of reference points was 100. We shall discuss the effects of the number of reference points in Section 5.3. We created the RP-tree on a dataset at timestamp 0. After that, 500 20-NN queries were performed. The average number of disk page accesses was used as the metric of performance of kNN search and updates, while throughput and response time were measured in evaluating concurrency control issues. The parameters used are summarized in Table 2, where values in bold denote default values used.

Parameter	Setting
Page size	4K
Node capacity	204
Space	5000×5000
Dataset	**Synthetic**, Real
Dataset size	**100K**, ... , 1M
Maximum velocity of inter-objects	10
Maximum velocity of intro-objects	6
Ratio between inter-objects and the total number of objects	0, 0.1, ... , **0.5**, ... , 1
Maximum update interval	120
Number of reference points	1, 25, 50, 75, **100**, 125,150
Number of queries	500
Number of nearest neighbors	1, **20**, 40, 80, 160, 320
Maximum predictive interval	60

Table 2. Parameters and their settings

5.1 Query Performance

In this set of experiments, we compared the query performance of the TPR-tree and the RP-tree by varying the size of the dataset, the ratio of inter-objects and total objects, the number of updates, and the number of required nearest neighbors. Finally, we investigate the performance of both indexes by using the real-network-based dataset.

5.1.1 Effect of dataset size

In the first experiment, we fixed the number of reference points to 100, and compared the search performance (queries were issued at timestamp 0) when the number of objects in the dataset ranged from 100K to 1M. As shown in Figure 8, the RP-tree scales up better than the TPR-tree. More than 50% I/O cost is reduced. This indicates that the search algorithm of the RP-tree, which restricts the search area often in one subspace, is more effective than that of the TPR-tree.

We also examine the size of ADT that may affect the search performance of the RP-tree. Figure 9 shows both the average and maximum size of ADT of all the subspaces. We can observe that the average size of ADT is small and increases slowly with the dataset size. Whereas, the maximum size of ADT is much bigger. The possible reason is that central subspaces may have objects coming into it from every directions which results in the big ADT size. This may not degrade the search performance, because one query needs to search only a few nodes that the ADT refers to. Recall that, the ADT stores the MBR information of the node, which avoids unnecessary node accesses.

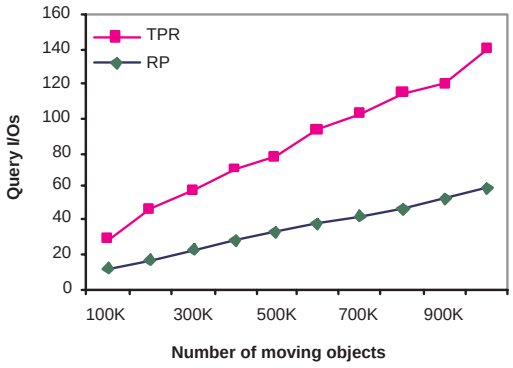


Figure 8. Query performance for varying number of moving objects

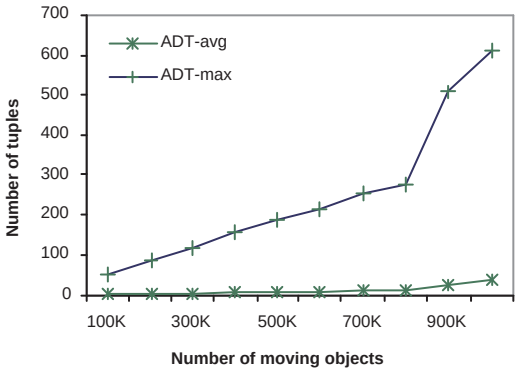


Figure 9. The size of ADT for varying number of moving objects

5.1.2 Effect of ratio between inter-objects and total objects

In this experiment, we varied the ratio between inter-objects and total objects from 0 to 1. As shown in Figure 10, the query costs of both trees are lower when the ratio is equal to 0 or 1, and reach the maximum value when the ratio equals 0.5. This behavior could be explained as follows. When the ratio equals 0, it means all objects are intra-objects with low speed. Accordingly, the MBRs containing such objects also expand at lower speed. When the ratio equals 1, it means all objects are inter-objects that move in the route of the network interconnecting the reference points. In this case, the MBRs may contain objects moving towards the same direction and move with these objects. Finally, when the ratio is between 0 and 1, the MBRs containing both slow-moving objects and fast-moving objects may enlarge seriously because slow moving objects usually move around reference points while fast moving objects travel among reference points.

Again, we study the average and maximum size of the ADT (see Figure 11). Observe that the average size of the ADT is almost the same for the different ratios. The maximum size of the ADT first increases and then shrinks, which shows the similar trend of the query performance of the RP-tree. This is because the ADT stores pointers to the nodes that contain objects may enter the current control area. When all the objects are slow or fast, objects that may enter the control area of other reference points are usually stored in the same nodes, and hence the ADT will contain less tuples.

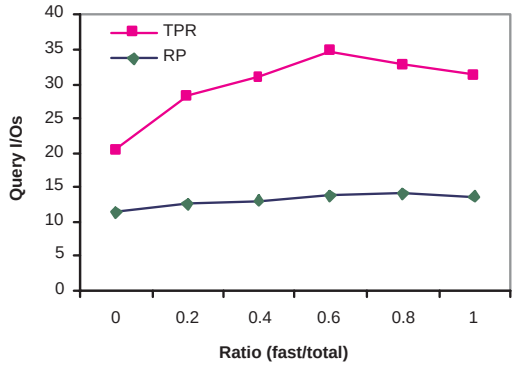


Figure 10. Query performance for varying ratio (inter-objects/total objects)

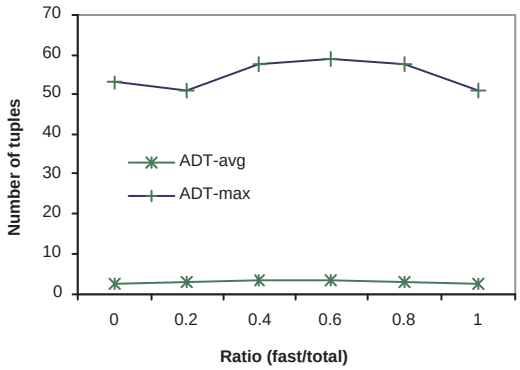


Figure 11. The size of ADT for varying ratio (inter-objects/total objects)

5.1.3 Effect of the number of updates

To study the character of the indexes with time, we measured the performance of the TPR-tree and the RP-tree using the same query workload after every 10K updates. Figure 12 shows the query cost as a function of the number of updates. We observe that the deterioration of the TPR-tree is considerably faster, whereas the RP-tree keeps almost constant performance. In particular, after 100k updates, the query cost of the TPR-tree is up to four times higher than the RP-tree. This is because, as mentioned in Section 3.1, each local TPR-tree of the RP-tree is relatively smaller than a single TPR-tree indexing the entire dataset, and small TPR-trees would be more efficient in search operation and less influenced by the increase in the overlap of the MBRs as time passes.

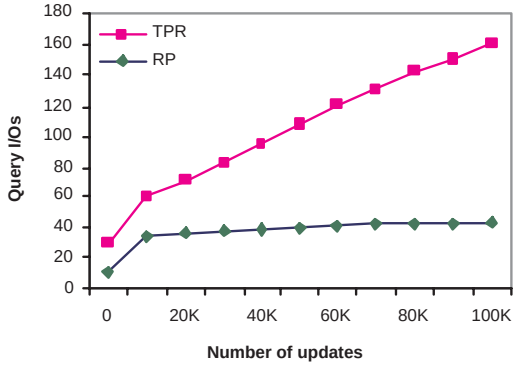


Figure 12. Query performance for varying number of updates

5.1.4 Effect of *k*

Figure 13 shows the query performance when the number of required neighbors (*k*) is varied. The search cost of both the RP-tree and the TPR-tree increase as *k* increases, since the search circle is larger for greater *k*. The RP-tree always outperforms the TPR-tree and the cost of the RP-tree increases much slower than that of the TPR-tree.

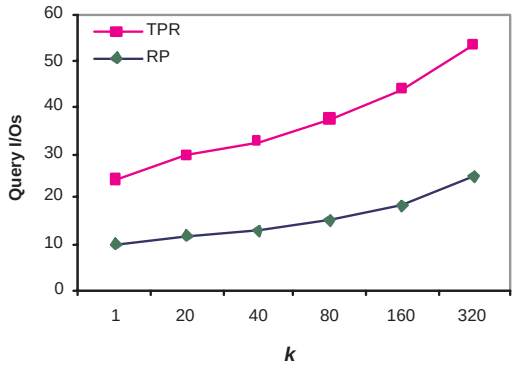


Figure 13. Query performance for varying *k*

5.1.5 Real data set

Finally, we compare the search performance of the TPR-tree and the RP-tree by using the real-network-based datasets. The number of reference points is fixed to 35. Figure 14 shows the results. As expected, the RP-tree always performs better than the TPR-tree. The reason is similar as that for Figure 8.

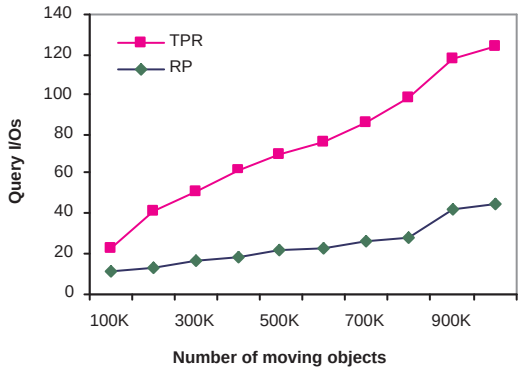


Figure 14. Query performance for real-network-based datasets

5.2 Update Performance

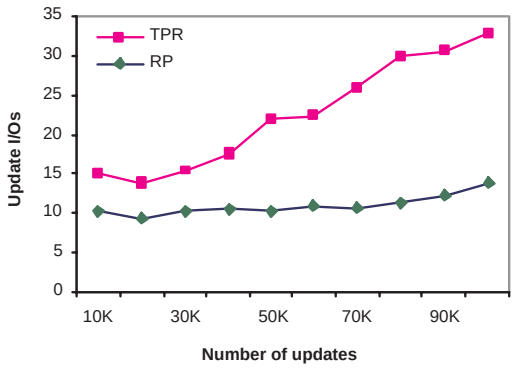


Figure 15. Update performance for varying number of updates

We now compare the average update cost (amortized over insertion and deletion) of the RP-tree against the TPR-tree. For 100K moving objects, we performed 100K updates and computed the average update cost of every 10K updates. Note that for each update, one deletion and one insertion were issued, which kept the size of the tree unchanged after updates. From Figure 15, we can see that the RP-tree maintains nearly constant update cost while the cost of the TPR-tree increases significantly over time. This is because each deletion must perform a query to retrieve the object to be removed, and the cost of this query increases with the number of updates. The RP-tree restricts the query to a few (and often one) local TPR-trees, and thus achieves better performance.

5.3 Effects of the Number of Reference Points

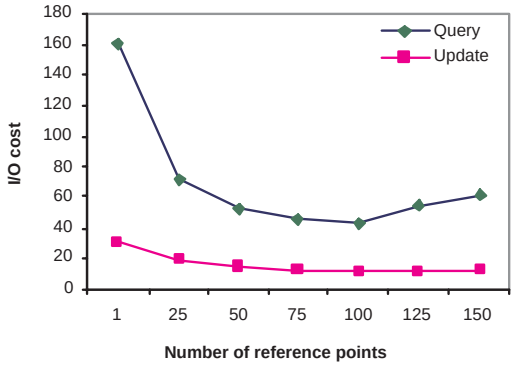


Figure 16. Performance for varying number of reference points

All the above experiments were conducted when the number of reference points was 100. To see how the number of reference points affects the performance of the RP-tree, we also carried out experiments

on datasets with different numbers of reference points as shown in Figure 16. Observe that the query and update costs both decrease quickly at first, because the TPR-tree in the RP-tree becomes smaller when the number of reference points increases. Then, the query cost increases slightly, because the control area of one reference point becomes too small and more neighbor TPR-trees need to be checked during one query process. The update cost has the same increasing trend because the size of the ADT grows when the number of reference points increases. Thus, the maintenance cost of the ADTs counteracts the savings from the smaller TPR-trees. According to these observations, we have use 100 as the default number of reference points in our experiments.

5.4 Concurrency Control

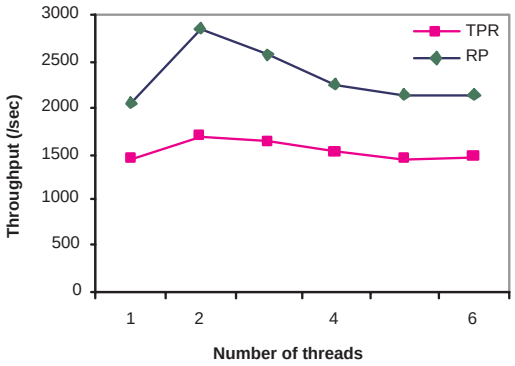


Figure 17. Throughput

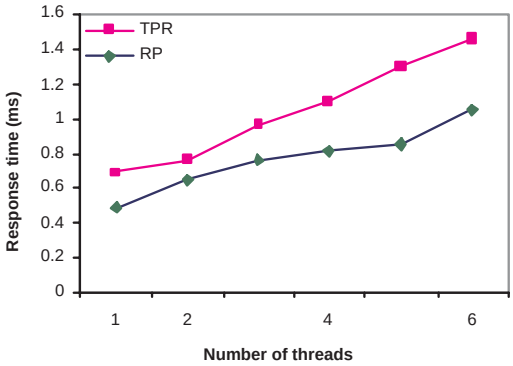


Figure 18. Response time

In this section, we compare the performance of the TPR-tree (also employing the R-link technique) and the RP-tree under multi-user environments. We used multi-thread programs to simulate multi-user environments. We number of threads varies from 1 to 6. The workload was defined as one searcher and a varying number of updaters (since update is more frequent than query in applications of moving objects). We investigated the throughput and response time of search and update operations on the index. Throughput is the rate at which operations could be served by the system. Response time is the time interval between issuing an operation and getting the response from the system when the task was successfully completed. Both the TPR-tree and the RP-tree reached their maximum throughput with two threads (one searcher and one updater). The results are shown in in Figure 17 and Figure 18. Throughput of the RP-tree is over 30% higher than that of the TPR-tree while the response time of the RP-tree is always less than that of the TPR-tree. The main reason for this behavior is that the local TPR-trees in the RP-tree are

independent of each other. In addition, the ADT search would only lock leaf nodes.

5.5 Cost Model Evaluation

Finally, we evaluate the accuracy of our cost model for the RP-tree. The data set used contains objects varies from 100K to 1M. We performed 500 20-NN queries on each data set and obtained the average I/Os. The result is shown in Figure 19. The maximum relative error between the estimated cost and the actual cost is 10%.

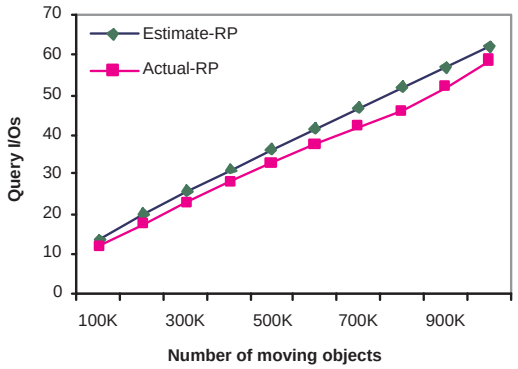


Figure 19. Cost model evaluation

6 Conclusion

With the wireless technique becoming increasingly popular, query efficiency is an important issue that challenges the systems that are based on it. In this paper, we have examined using the indexing technique to enhance performance for *k* nearest neighbor (kNN) queries.

Through the careful analysis of some existing index structures, we have discovered that traditional spatial index structures do not work well in moving object environments which are characterized by a large number of continuously moving objects and concurrent active queries over these objects. A new index structure called the RP-tree has therefore been proposed. Different from the existing approaches, the RP-tree indexes moving objects based on an important observation that queries usually occur around certain places, i.e., *reference points*. The RP-tree partitions the whole space into separate subspaces according to the reference points, and constructs individual TPR-trees in each subspace, with an auxiliary data structure, the ADT. We presented the kNN search and update algorithms for the RP-tree with concurrency control considerations. We also developed a cost model to estimate the disk page access cost of the RP-tree kNN

search. Our experimental study demonstrates that the RP-tree achieves considerable improvement over the TPR-tree in both single-user and multi-user environments and our cost model is accurate.

Currently, we are extending the RP-tree to integrate network information, that is, performing kNN queries with network route distance instead of Euclidean distance since the route distance would be more helpful in the real applications. The difficulty lies in two factors. First, it is usually costly to maintain and retrieve network information. Second, trajectory prediction of moving objects become complex, because we have taken the constraints imposed by roads into account.

References

[1] Digital chart of the world server. *http://www.maproom.psu.edu/dcw/*.

[2] C. C. Aggarwal and D. Agrawal. On nearest neighbor indexing of nonlinear trajectories. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 252–259, San Diego, CA, USA, Jun, 2003.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD Int. Conf. on Manag. of Data*, pages 322–331, Atlantic City, New Jersey, United States, May,1990.

[4] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proc. of Int. Database Engineering and Applications Symposium*, pages 44–53, Edmonton, Canada, July, 2002.

[5] K. L. Cheung and A. W c. Fu. Enhanced nearest neighbor search on the r-tree. *ACM SIGMOD Record*, 27(3):16-21, 1998.

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

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

[8] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Proc. of Int. Workshop on Spatio-Temporal Database Management*, pages 119–134, Hong Kong, China, Sep, 1999.

[9] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 134–145, Zurich, Switzerland, Sep, 1995.

[10] D. Kwon, Sa. Lee, and Su. Lee. Indexing the current positions of moving objects using the lazy update. In *Proc. of Int. Conf. on Mobile Data Management*, pages 113–120, Singapore, Jan, 2002.

[11] B. C. Ooi, K. L. Tan, and C. Yu. Fast update and efficient retrieval: an oxymoron on moving object indexes. In *Proc. of Int. Web GIS Workshop, Keynote*, Singapore, Dec,2002.

[12] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 395–406, Cairo, Egypt, Sep, 2000.

[13] S. Saltenis, C. S.Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. of ACM SIGMOD Int. Conf. on Mangement of Data*, pages 331–342, Dallas, Texas, USA, May, 2000.

[14] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 507–518, Brighton, England, Sep, 1987.

[15] Y. Tao and D. Papadias. Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 431–440, Roma, Italy, Sep, 2001.

[16] Y. Tao, D. Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. of Int. Conf. on Very Large Data Bases*, pages 790–801, Berlin, Germany, Sep, 2003.

[17] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *To appear in IEEE Transactions on Knowledge and Data Engineering*.

[18] J. Tayeb, O. Ulusoy, and O. Wolfoson. A quadtree-based dynamic attribute indexing method. *The computer Journal*, 1998.

[19] Y. Theodoridis and T. Sellis. Model for the prediction of r-tree performance. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–171, Montreal, Quebec, Canada, Jun, 1996.

[20] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *Proc. of Int. Conf. on Data Engineering*, pages 516–523, New Orleans, Louisiana, Feb, 1996.

[21] Y. Xia and S. Prabhakar. Q+rtree: Efficient indexing for moving object data bases. In *Proc. of Int. Conf. on Database Systems for Advanced Applications*, pages 175–182, Kyoto, Japan, Mar, 2003.