On Bulk Loading TPR-Tree *

Bin Lin *and* Jianwen Su
Department of Computer Science
University of California at Santa Barbara
{linbin, su}@cs.ucsb.edu

Abstract

TPR-tree is a practical index structure for moving object databases. Due to the uniform distribution assumption, TPR-tree's bulk loading algorithm (TPR) is relatively inefficient in dealing with non-uniform datasets. In this paper we present a histogram-based bottom up algorithm (HBU) along with a modified top-down greedy split algorithm (TGS) for TPR-tree. HBU uses histograms to refine tree structures for different distributions. Empirical studies show that HBU outperforms both TPR and TGS for all kinds of non-uniform datasets, is relatively stable over varying degree of skewness and better for large datasets and large query windows.

1. Introduction

With the rapid development of wireless technologies, moving object applications (e.g. mobile computing, traffic monitoring, digital battle fields, etc.) have become more realistic and attractive. Managing the information of continuous movements by objects becomes a new challenge in database research [28]. Key research issues in moving object databases include: modelling moving objects [8], efficient data management [21, 22], and improving application performance by using efficient index structures [2, 11, 20, 25, 10, 15, 30]. In this paper we focus on indexing of moving objects.

Performance of index structure is critical for moving object databases. Bulk loading of an index structure plays an important role in improving index performance. Bulk loading algorithms can be roughly classified into three categories [27]: sort-based, buffer-based, and sample-based methods. Sort-based methods include many bulk loading algorithms for R-tree such as [17, 9, 12, 6]. The first bulk loading algorithm for R-tree was introduced in [17]. [9] proposed a bulk loading algorithm based on Hilbert Sort. [12] introduced the STR (Sort-Tile-Recursive) algorithm, which is based on sorting and partitioning on each dimension. The latter two algorithms use the bottom-up technique to build the trees: starting from the leaf level, and then proceeding upward until the root is produced. On the other hand, [6] developed a greedy top-down algorithm that builds the tree from top to bottom. These algorithms mentioned above focus on improving the query performance of the tree. The buffer based bulk loading algorithm of [1] and sample-based bulk loading algorithms of [5, 27] focus on improving the bulk loading time.

Among existing index structures for moving objects, TPR-tree (Time Parameterized R-tree) [20] is a practical solution for predictive queries. However, the bulk loading algorithm described in [20] is based on a relatively strong uniform distribution assumption. This assumption prevents the algorithm from accommodating non-uniform datasets. In moving object applications, not only the distribution of moving objects' locations can be very skewed, so do their velocities. Consider a digital battle field application which involves locations of soldiers, vehicles, fighter planes, etc, and perhaps information obtained from satellites. Clearly the velocities of these moving objects are very different and not uniform. The TPR-tree's bulk loading algorithm, though capable of loading a tree with non-uniform data, can be further improved in dealing with situation such as the digital battle field application. In this paper, we present a histogram based bottom up (HBU) bulk loading algorithm to deal with non-uniform datasets and compare it with the original TPR-tree bulk loading algorithm and a modified top-down greedy split (TGS) bulk loading algorithm. Empirical studies show that HBU outperforms the other two algorithms for all kinds of non-uniform datasets. Moreover, HBU is less sensitive to different distribution than the other two algorithms. Interestingly, when the data size and the skewness increase, so does the improvement by HBU.

The remainder of the paper is organized as follows. Section 2 reviews TPR-tree and discusses in some detail its bulk loading problem. Section 3 presents a modified TGS algorithm for TPR-tree. Section 4 describes a new histogram

* Supported in part by NSF grants IIS-0101134 and IIS-9817432.

based bottom up algorithm for TPR-tree. Section 5 reports the performance evaluations. Section 6 concludes this paper.

2. Moving Objects, TPR-tree, and Bulk Loading

Moving object databases have received much interest in the database research community recently due to their wide range of applications. An immediate issue is to represent the information about object location changes which can be queried about [13, 29]. A prevailing approach is to use linear functions of time to represent moving objects' trajectories [29]. In this case, query evaluation can be done through solving linear constraints.

Index structures play a significant role in query evaluation. Due to continuous coordinate changes, indexing moving objects becomes a challenging problem [2, 11, 20, 25, 10, 15, 30].

Existing index structures for trajectories can be roughly divided into two categories: one focusing on current and anticipated future locations of moving objects [20, 25, 15], and the other on historical trajectories of moving objects [2, 11, 30]. We call the former location-based index structures and the latter trajectory-based ones.

In general, index performance for moving objects is affected by three factors: the initial tree structure, updates, and movements. While the impact of the first two factors is obvious, the impact of movements is unique to moving objects and new. For example, a bounding rectangle may expand when points inside it move, and this happens even without any explicit update operations. Clearly, performance of query evaluation may deteriorate over time. To overcome this problem, bulk loading and reshaping are two major techniques. In TPR-tree, reshaping is done through updates [20] which occur when objects change their movements (moving direction and/or speed). Fortunately, high update frequency in moving object databases provides many opportunities to restructure the tree to make it more efficient. However initial tree structure remains an important factor. In this paper we study bulk loading of index structures for moving objects. In particular, we focus on location-based index structures. We note that most trajectory-based index structures are based on R-tree family, whose bulk-loading has been relatively better understood [17, 9, 12, 6].

For location-based index structures, [25] indexes moving points as lines in $(d+1)$-dimensional space, where d is the number of physical dimensions. But the tree needs to be rebuilt periodically [20]. [15] maps trajectories to points using a dual space transformation. Queries are also transformed to the new space and evaluated. The major drawback is that the dual space transformation cannot be extended to moving points in two or more physical (excluding time) dimensions. [16] proposes the STAR-tree for moving objects. STAR-tree groups moving points according to their current locations. So some bounding rectangles may expand very fast. To overcome this problem, some special events should be scheduled to regroup the points. STAR-tree is most suitable for workloads with infrequent updates. Our study on bulk loading is based on TPR-tree developed by Saltenis and Jensen [20]. Parametric R-tree [4] is based on similar ideas of TPR-tree. The main difference is that Parametric R-tree indexes past trajectories of moving objects and TPR-tree focuses on current and future locations of moving objects. Saltenis and Jensen also introduced a TPR-tree similar index — $\mathrm{R}^{\mathrm{EXP}}$-tree [18] — to handle moving objects with expiration time. A recent paper proposes a query-parameterized cost model and new insertion/deletion strategies which are used to improve TPR-tree performance [24]. The results reported in this paper can be applied to the new version to achieve further improvements.

TPR-tree is an extension of R^*-tree [3] for moving points by allowing the bounding rectangles to expand with time. In this model, a moving point's location is given as a linear function of time, and so are the boundaries of bounding rectangles. Specifically, the velocities of the boundaries of a bounding rectangle are determined by the maximum and minimum velocities of the enclosed moving points or other expanding rectangles.

The bulk loading algorithm of TPR-tree developed in [19], denoted as TPR in the remainder of the paper, attempts to minimize the area aggregations for some initial period of time of length *horizon* (a parameter of TPR-tree). The TPR algorithm assumes uniform data distributions in both points' initial locations and velocities. Thus, a d dimensional moving point is represented as a point in the $2d$ dimensional space: d location dimensions and d velocity dimensions. So packing moving points into tree nodes means to partition $2d$ dimensional space.

Figure 1 illustrates the location and velocity dimensions for 1 dimensional moving points. Part (a) partitions location and velocity dimensions evenly, while Part (b) divides slabs only on location dimensions. In both approaches, the entire space is partitioned into 9 rectangles, and each rectangle is corresponding to one node in the tree, but the impacts of different partitions are different. Since this is one dimensional moving points, at each time instant, each node in a tree has a bounding interval along the location dimension. The initial extent of nodes in (b) is smaller than that in (a), but it increases faster than (a). It is easy to see that after some time the extents of nodes in (b) will become greater than that of (a).

One key assumption in the TPR algorithm is that location and velocity distributions are all uniform. While this assumption simplifies the analysis and the bulk loading process, real datasets are not always uniform. For skewed data,

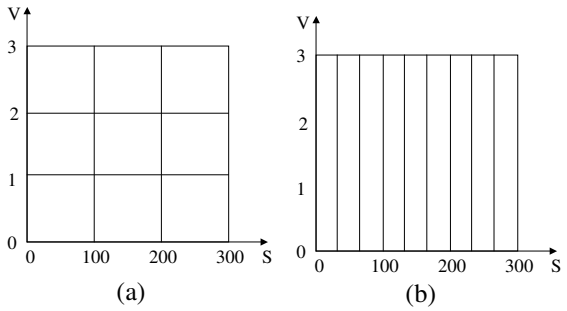


Figure 1. Examples of partitioning the location and velocity space

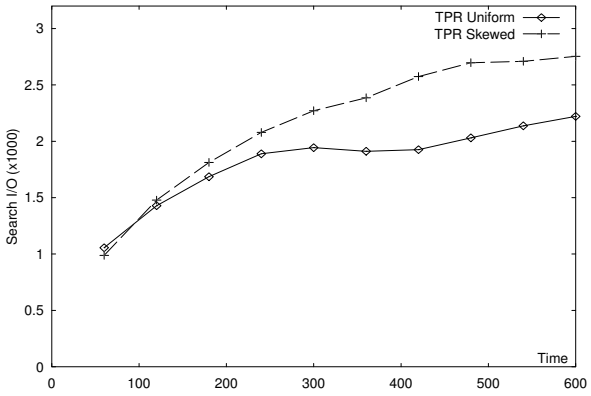


Figure 2. TPR-tree's query performance on uniform and skewed datasets

the query performance of TPR-tree built using the TPR bulk-loading algorithm deteriorates rapidly. Figure 2 compares the query performance of TPR-tree (in terms of I/O accesses) for uniform and skewed datasets. Both datasets contain 1 million two dimensional moving points. Page size is 1K bytes. Buffer size is 1,000 pages and LRU buffer management policy is applied. For the uniform dataset, all dimensions are uniform; for the skewed dataset, two location dimensions are uniform, and only two velocity dimensions are skewed. Figure 2 illustrates the evolutions of both tree's performances over time. It is clear that there is a significant gap between uniform and skewed data. The aim of this paper is to improve the bulk loading process for cases of skewed datasets in order to obtain more efficient TPR-trees.

3. A Top-down Greedy Split Algorithm

In this section we will present a top-down greedy algorithm for TPR-tree. Our algorithm is modified from the TGS algorithm [6] for R-tree, which tries to build the tree top down greedily through split operations. The approach considers the entire dataset during bulk loading; thus the algorithm can adapt to variation of distributions of input datasets.

We first briefly review the original TGS algorithm [6] and then discuss our modifications for TPR-tree. The core of the TGS algorithm is to construct tree nodes in the top-down manner. Let f be the fan-out factor and $M_i = f^i$ be the expected number of objects in a subtree of height i. Assume that we have a set of n objects which belong to a subtree of height $i+1$, where $M_i < n \leq M_{i+1}$. TGS repeatedly executes a split operation until we have all subsets with size M_i except for one which has size $\leq M_i$. For each split operation on a subset of $m \leq n$ objects, TGS examines, for each value of k where $1 \leq k \leq \lceil \frac{m}{M_i} \rceil$, a partition with a cut orthogonal to an axis into two subsets of size kM_i and $m - kM_i$ that minizes the sum of costs of the two resulting sets. TGS chooses the the partition with the least cost.

TPR-tree is different from R-tree in the following two ways. First, In R-tree, all the objects are static, while in TPR-tree, all the objects are moving, and the bounding rectangles are expanding. So the cost function to be minimized for the split operations is different in the case for TPR-tree. Second, TPR-tree has velocity dimensions in addition to location dimensions.

To modify TGS for TPR-tree, we made two major modifications. One is to add velocity dimensions. So for d dimensional datasets, as the analysis in TPR-tree, we actually consider $2d$ dimensions — d location dimensions and d velocity dimensions. The other is a new cost model that is discussed below.

Similar to the bulk loading algorithm TPR in [20], we want to minimize the integral of the cost function (e.g. areas of the bounding rectangles) over $[t_l, t_l + h]$, where t_l is the loading time and h is the horizon parameter of TPR-tree. More precisely, if the two resulting sets are r_1, r_2 and $Area(r, t)$ denotes area of a set of moving objects r at time t. Without loss of generality, we let $t_l = 0$. The cost function for TPR-tree used in our algorithm is the following formula:

$$F(r_1, r_2) = \int_0^h (Area(r_1, t) + Area(r_2, t))dt$$

Figure 3 shows the basic split step of TGS for TPR-tree. This step takes a set of moving objects as input, goes through each candidate orthogonal split, and then chooses the best one according to the cost function. The two resulting new sets are returned as the output. Lines 2 and 7 of the algorithm reflect the two changes for TPR-tree discussed above, respectively.

Fig. 4 shows a one-dimensional example for the algorithm above. In this example, $h = 10, M = 2$. So lines A, B, C, D represents 4 possible orthogonal splits. In case A, $r_1 = a, b, r_2 = c, d, e, f$. $Area(r_1, t) = 10 +$

Algorithm BasicSplitStep(*n, M, F, N*)

Input: n – number of objects in the input node;
M – maximum number of objects per subtree;
F – the cost function in the form of $F(r_1, r_2)$;
S – input set.

Output: S_1, S_2 – two resulting sets.

1. If $n \le M$ stop (don't need to split)
2.* For each dimension d (including velocity dimensions)
3. Sort the input objects along this dimension
4. For i from 1 to $\lceil n/M \rceil$ - 1
5. Let $B_0 \leftarrow$ MBR of first $i \times M$ objects
6. Let $B_1 \leftarrow$ MBR of the other objects
7.* Remember i and the dimension d for minimal $F(B_0, B_1)$
8. Order the input set and split them into S_1, S_2 according to the final i and d

Figure 3. The algorithm of the basic split step

$2t, Area(r_2, t) = 20 + 3t$, here function $Area(r, t)$ actually computes the extents of r since there is only one dimension. So the cost of split A $F_A(r_1, r_2) = 550$. In the same way, we can get $F_B(r_1, r_2) = 650$, $F_C(r_1, r_2) = 750$, and $F_D(r_1, r_2) = 700$. So split A will be chosen.

4. HBU: A Histogram-based Bottom-Up Algorithm

The TGS algorithm presented in the previous section minimizes the cost function (e.g., the total area or area integral) at every split step. Such a strategy results in better trees in many cases since some properties of the dataset are reflected in the top down process. However, in other cases especially when the data are severely skewed, the top-down split process is incapable of adapting to the skewed distributions. In order to generate better trees in skewed data cases, we introduce a histogram-based bottom-up (HBU) bulk loading algorithm in this section. During the bulk loading process, this algorithm does global optimization by deciding the number of slabs on each dimension according to the distribution information. In the next section, we will show that HBU outperforms both TPR and TGS algorithms.

The algorithm HBU uses the same cost model as TPR, that is, aiming at minimizing the area integrals of all resulting time-parameterized bounding rectangles across time interval $[0, h]$, where h is the horizon parameter of TPR-tree. With the help of histograms, HBU does not need to make any distribution assumptions, such as the uniform assumption for the TPR algorithm. To compensate for skewed data, HBU calculates the expected extents of the resulting bounding rectangles for each candidate partition, and then decides the number of slabs on each dimension according to the cost calculated from the extents.

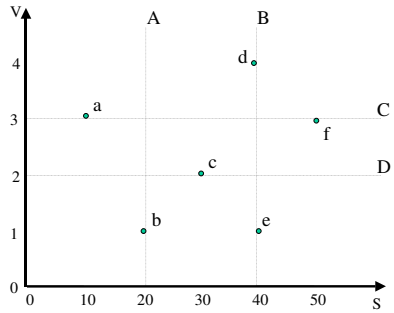


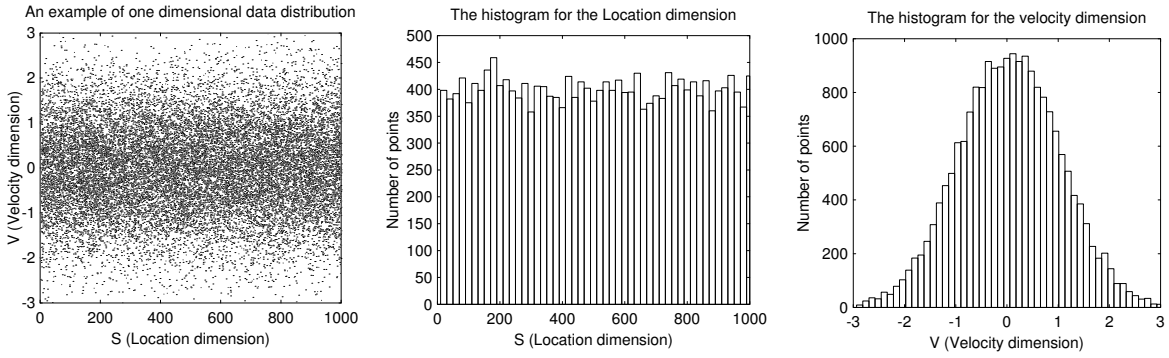
Figure 4. An example for split algorithm ($h = 10, M = 2$)

HBU builds the tree in the bottom-up fashion. It constructs the leaf level first, then takes the leaf level nodes as input to produce the last second level. This process is repeated until the root is produced. For all levels, the construction procedures are similar and consist of the following two steps: (1) the number of slabs on each (location or velocity) dimension is determined based on the histograms and number of expected resulting rectangles of this level, and (2) input objects are packed into rectangles according to these numbers.

4.1. Histograms

Histograms are widely used to describe distributions of datasets [7, 23]. The use of histograms avoids the need of making explicit assumptions on data distributions for the bulk loading process. In our algorithm, a histogram can describe the distribution of a location dimension or a velocity dimension. In general, for a d physical dimensional dataset (i.e., points moving in d dimensions), there are $2d$ histograms in total: d histograms for location dimensions and d histograms for velocity dimensions.

A *histogram* for a location or velocity dimension is composed of an array of buckets. Each bucket represents an interval on the location or velocity dimension, respectively, and records the number of points covered by the interval. In our algorithm, the intervals of all buckets in a fixed dimension are of the same size. Figure 5(a) shows a dataset containing the initial location and velocity of moving points in one dimensional space. Figure 5(b) is the histogram of the dataset on the location dimension, and (c) is the histogram on the velocity dimension. For this dataset, the distribution of the location dimension is uniform, while that of the velocity dimension is Gaussian.



(a) (b) (c)

Figure 5. An example of histograms for one dimensional moving points

In the following discussion, we focus on the algorithm for determining the number of slabs on each dimension. First the algorithm for one dimensional moving points is introduced, then it is extended to two dimensions.

4.2. One dimensional moving points

A dataset for one dimensional moving points actually has two dimensions — one for the initial locations, the other for velocity. For this two dimensional space, packing these points into tree nodes corresponds to partitioning this space into bounding rectangles. If the number of rectangles after partitioning is K, and we can cut S slabs on the location dimension, and V slabs on the velocity dimension, such that $K = SV$ holds. The key in partitioning is to determine the values of S and V.

In our algorithm, the ratio between S and V is determined according to the histograms. The tree is constructed in the bottom up fashion. Let level 0 be the bottom level (leaves). Our cost model is defined as follows.

$$cost = \sum_{m=1}^{S} \sum_{n=1}^{V} \int_0^h (L_s[m] + L_v[n]t)dt \qquad (1)$$

where h is the horizon parameter of TPR-tree, $L_s[]$ and $L_v[]$ are the expected slab sizes (called *extents*) for the location and velocity dimensions, respectively. $L_s[]$ and $L_v[]$ are computed from the respective histograms. Intuitively, the formula calculates the sum of area integrals of all resulting time parameterized bounding rectangles over time interval $[0, h]$,

Since the number of slabs on each dimension must be a whole number, the algorithm HBU tries all possible combinations of S and V where $V = \lceil K/S \rceil$. More precisely, for each combination, HBU computes the expected extents of slabs on each dimension with the distribution information from histograms. Based on these extents information, HBU uses Equation (1) to calculate the cost for the given S and V combination. Then the combination with smallest cost is chosen as the solution.

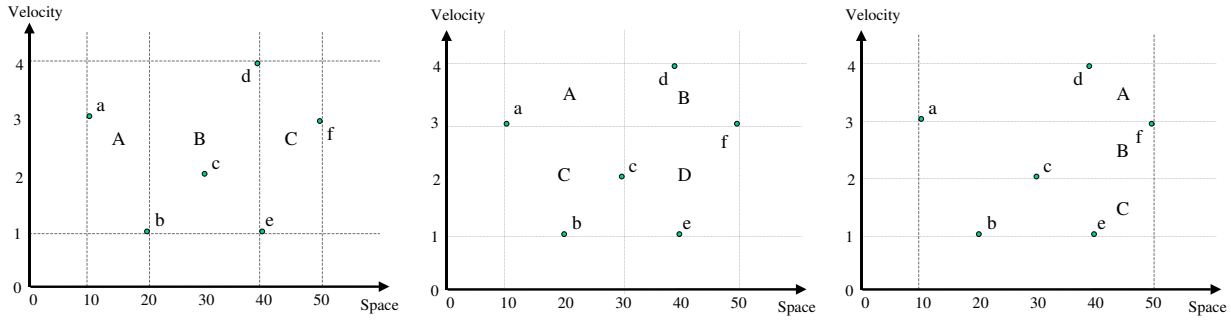
Figure 6 illustrates the computation of HBU. In this example $K = 3, h = 10$, so there are three combinations for S and V: (a) $S = 3, V = 1$, (b) $S = 2, V = 2$, and (c) $S = 1, V = 3$. The three cases are shown in Figure 6(a-c), respectively. In case (a), $L_s[1] = 10, L_s[2] = 20, L_s[3] = 10$, and $L_v[1] = 3$. Therefore,

$$cost_a = \sum_{m=1}^{S} \sum_{n=1}^{V} \int_0^h (L_s[m] + L_v[n]t)dt = 850.$$

Similarly, we get $cost_b = 1100$ and $cost_c = 1350$. Clearly, combination (a) has the least cost and will be chosen. This example illustrates the aspect of the algorithm in determining the optimal combination for S and V and assumes that the values of L_s and L_v are already computed. The algorithm on the other hand computes the L_s and L_v based on the provided histograms.

Figure 7 is the algorithm DetermineSV to determine S and V for one level of the tree. During the bulk loading process, the algorithm DetermineSV is called once for each level to determine the S and V values. Line 1 does an exhaustive search over the all possible integer combinations of S and V which satisfy $V = \lceil K/S \rceil$. For each candidate combination, lines 2 and 3 use the function CalculateExtent() to determine the expected extents for resulting location slabs (L_s) and velocity slabs (L_v), respectively. Line 4 calculates the corresponding cost according to the expected extents.

Figure 8 shows the algorithm of CalculateExtent. Line 1 calculates the expected number of points per slab. Lines 3 and 4 are responsible to estimate the extents — first the points are counted into slabs. Whenever a slab is full, the expected extent of it is calculated according the extent information of the input histogram.



(a) $S = 3, V = 1$ **(b)** $S = 2, V = 2$ **(c)** $S = 1, V = 3$

Figure 6. An example of HBU for one dimensional moving points ($K = 3, h = 10$)

Algorithm DetermineSV(*H_s, H_v, K*)
Input: H_s – the histogram for space dimension;
H_v – is the histogram for velocity dimension;
K – expected number of resulting bounding rectangles.
Output: S – number of slabs on space dimension;
V – number of slabs on velocity dimension.
1. for any integer combination (S, V) satisfying $V = \left\lceil \frac{K}{S} \right\rceil$ do
2. $L_s \leftarrow$ CalculateExtent(S, H_s)
3. $L_v \leftarrow$ CalculateExtent(V, H_v)
// Calculate the sum of all area integrals of time
// parameterized rectangles
4. $cost \leftarrow \sum_{m=1}^{S}\sum_{n=1}^{V}\int_0^h (L_s[m] + L_v[n])dt$
5. if *cost* is better valued, record current S, V values
6. return S and V

Figure 7. The algorithm for deciding S and V

Algorithm CalculateExtent(*NumOfSlabs, H*)
Input: *NumofSlabs* – number of slabs on this dimension;
H – the histogram of this dimension.
Output: L – the array of extents of resulting slabs.
1. *NumPerSlabs* $\leftarrow \lceil N / NumOfSlabs \rceil$; $j \leftarrow 0$
2. for i from 0 to *NumOfSlabs* do
3. starting from the j^{th} bucket,
count the *H[j].num* into the slab until
their sum = *NumPerSlabs*
4. *L[i].extent* $\leftarrow$ sum of the *H[j].extent*
5. Adjust j to corresponding value for next
6. return L

Figure 8. Calculating expected extents of the resulting slabs

4.3. Two dimensional moving points

Now we turn to datasets for two dimensional moving points. The algorithm for one-dimensional moving points can be naturally extended to higher dimension by trying all possible combinations of slab numbers on different dimensions exhaustively. But the time complexity will increase exponentially. One idea for improvement is to first determine the product of the location slab number and the velocity slab number on each physical dimension of movements, then apply the one-dimension algorithm on each such physical dimension. More precisely, let S_1 and S_2 denote the number of slabs on first and second location dimensions, respectively, and V_1 and V_2 denote the number of slabs on first and second velocity dimensions, respectively. We first determine the products S_1V_1 and S_2V_2, and then calculates the exact values of S_1, V_1, S_2, V_2 by applying the one dimensional algorithm on each physical dimension. The following analysis is based on this idea.

Since the distribution is not uniform, our idea is to focus on the "bulk" part of data. For each dimension, if μ is the mean value of the dataset, and σ is the standard deviation, then we focus on the data within the interval $[\mu - \sigma, \mu + \sigma]$, which accounts for roughly 68% of data in the Gaussian distribution. In the following analysis, we begin with the assumption that the data distribution inside this interval is uniform. (Note that the assumption is dropped when calculating the number of slabs for individual dimensions using the algorithm DetermineSV (Figure 7).

First we focus on the two location dimensions to find the relationship between S_1 and S_2. The goal is to minimize the total area of all the rectangles. Let the total number of rectangles formed on the two location dimension be K_s. Then $K_s = S_1S_2$. So the sum of the area of rectangles Sum_a is $4\sigma_{s,1}\sigma_{s,2}$, where $\sigma_{s,1}$ and $\sigma_{s,2}$ are the standard deviations of the first and the second location dimensions, respectively. Obviously Sum_a is unaffected by the ratio be-

tween S_1 and S_2. We can then try to minimize the sum of perimeters of all rectangles Sum_p.

$$Sum_p = 4K_s(\frac{\sigma_{s,1}}{S_1}+\frac{\sigma_{s,2}}{S_2}) = 4K_s(\frac{\sigma_{s,1}}{S_1}+\frac{\sigma_{s,2}}{K_s/S_1}) \quad (2)$$

Setting the partial derivative of Equation (2) be 0, we get

$$\begin{aligned} \frac{\partial Sum_p}{\partial S_1} &= 0 \\ S_1 &= \sqrt{\frac{K_s\sigma_{s,1}}{\sigma_{s,2}}} \\ \frac{S_1}{S_2} &= \frac{\sigma_{s,1}}{\sigma_{s,2}} \end{aligned} \quad (3)$$

Now we bring the two velocity dimensions into consideration. Here the rectangles are time parameterized, in particular, they are expanding over time. So we want to minimize the area of these time parameterized rectangles. Suppose $e_{s,1}$ and $e_{s,2}$ are the overall extents of first and second location dimensions, $e_{v,1}$ and $e_{v,2}$ the overall extents of first and second velocity dimensions, respectively. Let K be the number of resulting time parameterized rectangles. Then the sum of the area is

$$\begin{aligned} Sum &= K(e_{s,1} + e_{v,1}t)(e_{s,2} + e_{v,2}t) \\ &= K(e_{v,1}e_{v,2}t^2 + (e_{s,1}e_{v,2} + e_{s,2}e_{v,1})t \\ &\quad +e_{s,1}e_{s,2}) \end{aligned} \quad (4)$$

Due to the uniform distribution, $e_{v,1}e_{v,2}$ and $e_{s,1}e_{s,2}$ are unchanged, so we only need to minimize $(e_{s,1}e_{v,2} + e_{s,2}e_{v,1})t$. In addition, we have

$$\begin{aligned} e_{v,1} &= \sigma_{v,1}/V_1 \quad (5) \\ e_{v,2} &= \sigma_{v,2}/V_2 \quad (6) \\ e_{s,1} &= \sigma_{s,1}/S_1 \quad (7) \\ e_{s,2} &= \sigma_{s,2}/S_2 \quad (8) \\ K_v &= V_1V_2 \end{aligned}$$

where K_v is the number of slabs on velocity dimensions. From Equations (5) and (6), we obtain

$$e_{v,2} = \frac{A}{e_{v,1}}$$

where $A = \frac{\sigma_{v,1}\sigma_{v,2}}{K_v}$. To minimize

$$\begin{aligned} f(e_{v,1}) &= (e_{s,1}e_{v,2} + e_{s,2}e_{v,1})t \\ &= (e_{s,1}\frac{A}{e_{v,1}} + e_{s,2}e_{v,1})t \end{aligned}$$

let the derivative of $f(e_{v,1})$ be 0, we have

$$\begin{aligned} e_{v,1} &= \sqrt{\frac{Ae_{s,1}}{e_{s,2}}} \\ e_{v,2} &= \frac{A}{e_{v,1}} = \sqrt{\frac{Ae_{s,2}}{e_{s,1}}} \end{aligned}$$

By dividing $e_{v,1}$ with $e_{v,2}$, and then replacing $e_{v,1}$, $e_{v,2}$, $e_{s,1}$, and $e_{s,2}$ with Equations (5), (6), (7), and (8), we get

$$\frac{S_1V_1}{S_2V_2} = \frac{\sigma_{s,1}\sigma_{v,1}}{\sigma_{s,2}\sigma_{v,2}}$$

Now we let $B = \frac{S_1V_1}{S_2V_2} = \frac{\sigma_{s,1}\sigma_{v,1}}{\sigma_{s,2}\sigma_{v,2}}$. Then

$$S_1V_1 = \sqrt{BK} \quad \text{and} \quad S_2V_2 = \sqrt{K/B}.$$

This gives us the product of location and velocity slab numbers on each physical dimension. Based on this, we apply the one-dimensional algorithm on each dimension to get S_1, V_1, and S_2, V_2, which are used to build the tree.

5. Performance Evaluation

In this section we evaluate the HBU algorithm by comparing it with the TPR algorithm of [20] and the modified TGS (top-down greedy split-based) algorithm presented in Section 3. First we describe the experimental setting in particular the datasets and parameters for the experiments (Section 5.1). We then give overall comparison results that HBU outperforms both TPR and TGS in all kinds of non-uniform distribution (Section 5.2). Experiments also show that HBU is less sensitive to skewed data than TPR and TGS (Section 5.3). Finally, we study scalability (Section 5.4), impact of query window size (Section 5.5), and horizon values (Section 5.6) for the three algorithms.

5.1. Datasets and parameters of experiments

We use a data generator similar to GSTD [26]. Since datasets generated by GSTD contain only discrete positions of moving objects, they cannot be used to create a TPR-tree directly [14]. So we extract the distribution functions from the GSTD package and combine them with current TPR-tree's generator to get datasets with different distributions.

We evaluate the effect of the three bulk loading algorithms using three categories of synthetic datasets for two dimensional moving points with uniform, Gaussian, and skewed distributions of the initial locations and velocities, respectively. In GSTD, standard deviation is used to control the skewness for Gaussian distribution. Smaller standard deviation means more skewness. Skewed distribution is produced by a Zipfian distribution generator borrowed from GSTD. It is also controlled by a parameter α. Greater α means more skewness.

Moving points are generated in a workspace of 1,000 square kilometers. The number of moving points is $N = 1,000,000$ for most experiments. After bulk loading, a workload composed of both queries and updates is executed over the tree. For most experiments, we evaluate the tree's

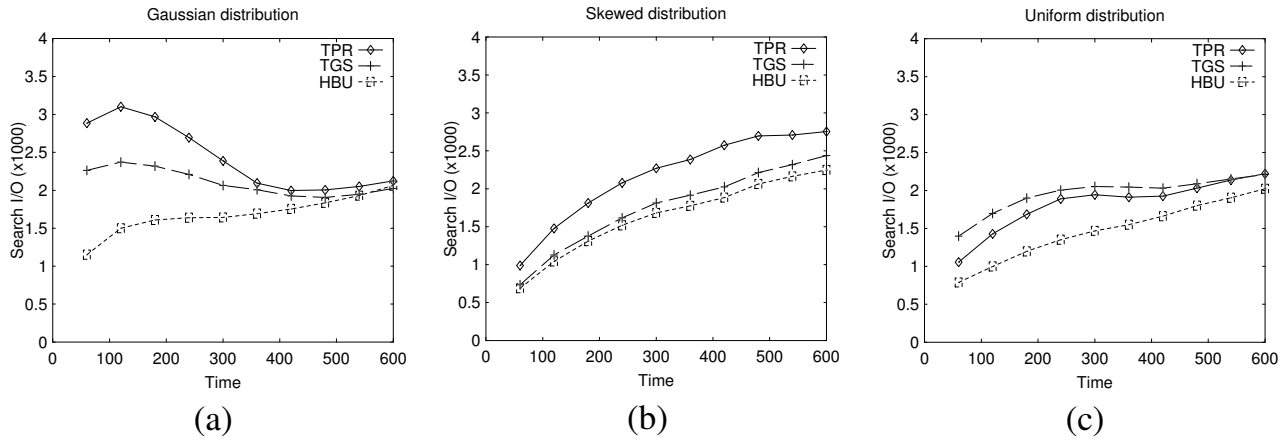


Figure 9. Average search I/O ($\times 10^3$) for Gaussian (v2), skewed (v2), and uniform datasets

performance for the first 600 time units (minutes) after the bulk loading. A parameter called Update Interval (UI) is used to control the frequency of updates [19]. It represents the approximate time interval between two consecutive updates for the same object. For most experiments $UI = 600$, and the horizon parameter h of TPR-tree is set to $UI/2$. We will give the reason for this h value in later experiments. The maximum speed of moving points is 10km/min (600km/h).

For all experiments, the disk page size is set to 1k bytes so that trees constructed can have reasonably large size without increasing the sizes of datasets. This shortens the time needed to complete the experiments without sacrificing the quality of experimental results, since (1) the running time of experiments depends on the dataset size, and (2) the experimental results (i.e., disk accesses) depend basically on the size of the tree. So the maximum number of entries is 51 for leave nodes and 28 for intermediate nodes. Buffer size is $1,000$ pages for most experiments and LRU buffer management policy is applied.

TPR-tree supports three types of queries: time instant range queries, time interval range queries, and moving range queries. The first two categories are straightforward. A moving range query is also a time interval query, but the query window may change (e.g., move, expand, or shrink) over time. A parameter *QWS* is used to described the size (area) of query windows in terms of a fraction of the total workspace. For most experiments we set $QWS = 1\%$. The compositions of the three types of queries are 60%, 20%, and 20%, the same as the composition used in [20]. For time interval queries (the latter two types), the query time interval (*QTI*) is randomly picked from $[0, 20]$. For time instant/interval range queries, query windows are randomly scattered over the workspace; for moving range queries, the centers of query windows are always some moving points in the tree.

5.2. Overall comparisons

In our first experiment, we compare the three bulk loading algorithms over all three categories of datasets. In particular, for non-uniform (i.e., Gaussian and skewed) datasets we explored all possible combinations of non-uniformity on the initial location and velocity dimensions. We use the notation "sivj" ($0 \leq i, j \leq 2$) to denote i location dimensions and j velocity dimensions are non-uniform, while other dimensions are uniform. When $i = 0$ ($j = 0$), we drop "si" (respectively "vj") in the notation; when $i = j = 1$, we use a suffix "s" to mean the non-uniform dimensions are the initial locations and velocity of the *same* physical dimension, and "d" to mean that they belong to *different* physical dimensions. For both Gaussian and skewed distributions, we considered all possible combinations (Figure 10).

Figure 9(a) and (b) compare the three bulk loading algorithms for non-uniform datasets of type "v2" with Gaussian (a) and skewed (b) distributions. The overall performance of HBU is always better than that of TPR and TGS. We note that for the Gaussian distributions, the differences are initially large and become smaller as the time progresses, while for the skewed case, differences stay relatively steady. This is because TPR and TGS partition too much on the location dimensions in the Gaussian distribution case. So they perform at the beginning much worse than HBU. However, the expanding speeds of the bounding rectangles are relatively small, so after a while their performances become similar to HBU. Figure 9(c) compares the three algorithms for the uniform dataset. In this experiment TGS performs a little worse than TPR. This is easy to understand since TPR's analysis is based on uniform distribution assumption. But surprisingly HBU performs even better than TPR for the uniform case. This may indicate that HBU's histogram

Gaussian Distribution

type	TPR	TGS	TGS/TPR	HBU	HBU/TPR
s1	2.10	2.15	102.4%	1.87	89.0%
v1	1.80	1.98	110.0%	1.78	98.9%
s2	2.15	2.23	103.7%	1.87	87.0%
v2	2.43	2.10	86.4%	1.68	69.1%
s1v1s	1.97	1.94	98.5%	1.61	81.7%
s1v1d	2.24	2.00	89.3%	1.72	76.8%
s1v2	2.31	2.16	93.5%	1.82	78.8%
s2v1	2.09	2.12	101.4%	1.90	90.9%
s2v2	2.13	1.85	86.9%	1.68	78.9%

Skewed Distribution

type	TPR	TGS	TGS/TPR	HBU	HBU/TPR
s1	1.86	1.77	95.2%	1.61	86.6%
v1	2.27	2.01	88.5%	1.66	73.1%
s2	2.11	2.16	102.4%	1.92	91.0%
v2	2.18	1.76	80.7%	1.64	75.2%
s1v1s	1.86	3.23	173.7%	1.72	92.5%
s1v1d	1.96	1.78	90.8%	1.66	84.7%
s1v2	1.71	1.50	87.7%	1.48	86.5%
s2v1	2.12	2.19	103.3%	2.11	99.5%
s2v2	3.53	3.38	95.8%	3.19	90.4%

Figure 10. Average search I/O ($\times 10^3$) for Gaussian and skewed datasets

based estimation of the real query performance (i.e., cost) is slightly more accurate than TPR's cost model.

We also examined different scenarios of non-uniformity with Gaussian and skewed datasets. The "TGS/TPR" and "HBU/TPR" columns in Figure 10 show the ratios of TGS and HBU over TPR, respectively. A ratio less than 100% means TGS or HBU performs better than TPR.

From the tables in Figure 10, we can see that HBU outperforms consistently both TPR and TGS on all datasets with all different combinations of non-uniformities having Gaussian distribution and skewed distribution, respectively. The improvement is as high as up to 30%. The average improvement gained for Gaussian datasets is higher than that for skewed datasets. This may be explained by two factors: (1) some of our analysis is based on Gaussian distribution, and (2) initial performance gain for Gaussian datasets (Figure 9) is quite significant. Overall, TGS is also slightly better than TPR, with the difference of performance is less than that between HBU and TPR. It is easy to see that HBU performs best among the three, thanks mainly to the distribution information from histograms.

One conclusion from results in Figures 9 and 10 is that HBU's performance is relatively more stable for different distributions than the other two algorithms. This is not surprising since the histograms allow HBU to refine tree structures to better accommodate variations in data distributions, and as a result to yield better performance.

For the remainder of this section, we will focus on non-uniform distribution datasets. Due to the limitation of space, we only display the experiment results for "v2" type (two velocity dimensions are non-uniform and two location dimensions are uniform).

5.3. Sensitivity to skewness

Figure 11 shows how the degree of non-uniformity affects the three algorithms. We use $\beta = \frac{\max - \min}{\sigma}$ as the non-uniformity factor for Gaussian distribution and α (skewness factor, introduced in Section 5.1) for skewed distribution, where $\max$, $\min$, and σ are the maximum value, minimum value and standard deviation, respectively. The higher β and α values are, the more skewness a dataset is.

Also, in the skewed distribution we make most points move with small velocities to prevent them from moving too far. So when the skewness increases, the expanding speeds of bounding rectangles become smaller, which makes cost (disk I/Os) of queries decreases.

From Figure 11, we can see that HBU outperforms both TPR and TGS for both Gaussian and skewed datasets and TGS performs worse than HBU, but still better than TPR especially when the non-uniformity factors are higher. In all cases, the I/O accesses of HBU are relatively stable when varying skewness degrees. TPR on the other hand deteriorates for higher skewness cases These confirms the ability of HBU and the inability of TPR in adapting to input datasets. Not surprisingly, TGS always behaves in-between TPR and HBU. For the Gaussian cases, the I/O cost of TPR and TGS increases for higher skewness which is expected. However, similar phenomena do not occur for skewed datasets. We suspect that it may due to the high skewness for skewed dataset makes the datasets appear closer to uniform for the bulk part. All algorithms perform better for when datasets

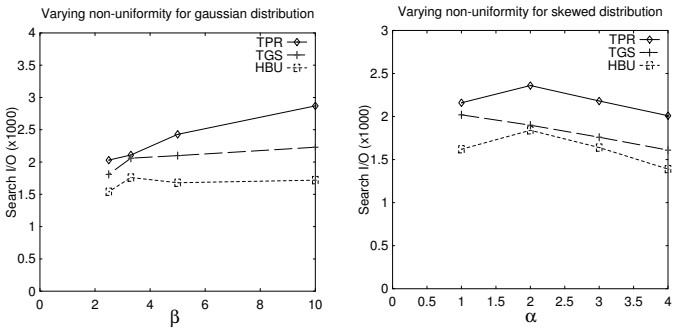


Figure 11. Average search I/O ($\times 10^3$) with varying skewness

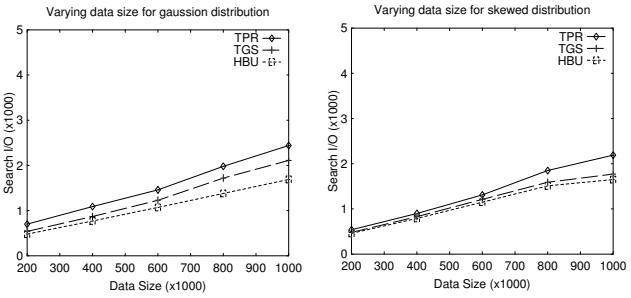


Figure 12. Average search I/O ($\times 10^3$) with varying dataset size

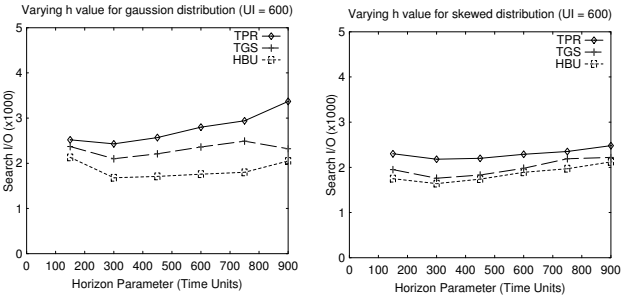


Figure 14. Average search I/O ($\times 10^3$) with $UI = 600$ and varying Horizon values

are more uniform (Figures 9 and 10).

5.4. Scalability

We also examine the situations when the dataset sizes increase. Figure 12 displays the average I/O operations per query for the three algorithms when the number of moving points varies from 200,000 to 1 million. In this experiment, we set buffer size to 0 to prevent smaller data size from gaining more benefits from buffer. We can see that for all different sizes of datasets, HBU outperforms TGS and TPR. But for smaller size, the improvement by HBU is relatively small. This is expected because when number of the moving points is small, the number of slices to divide is also small, which reduces the advantages of using histograms.

5.5. Impact of query window size

Figure 13 explores the impact of the query window size on different bulk loading algorithms. The chart on left hand side is for Gaussian datasets and the chart on right hand side is for skewed datasets. When the size of the query window is small, so is the difference of these three algorithms. When the size increases, so does the improvement by HBU. This

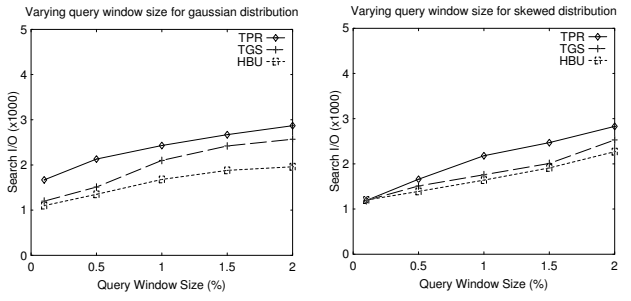


Figure 13. Average search I/O ($\times 10^3$) with varying query window size

can be explained again by the intuition that larger query windows have more opportunities to intersect with bounding rectangles.

5.6. Determining horizon values

In our last set of experiment, we tried to determine the best h (horizon parameter) value for the TPR-tree. [19] points out that the best values of h are between $UI/2 + QTI$ and $UI + QTI$ for their experiments, where UI is the update interval and QTI is the query time interval. In our experiment, we investigated the best h value for non-uniform datasets. Figure 14 shows that our experiment results comply with the observation made in [19]. Indeed, all algorithms prefer h value between $[UI/2, UI]$. More precisely, when $h = UI/2$, they get the best performance. So in the experiments above, we use $h = UI/2$. Another interesting phenomenon we observe is that HBU is less sensitive to different horizon values.

6. Conclusions

Motivated by emerging challenges for indexing moving object databases, and the importance of bulk loading strategy for index structures, this paper proposed a histogram-based bottom up (HBU) bulk loading algorithm for TPR-tree, a practical index structure for predictive queries of moving objects. HBU can easily accommodate datasets with different distributions. By using histograms, this algorithm refines the tree structure to adapt to different distributions. We also present a modified TGS bulk loading algorithm for TPR-tree. Empirical studies demonstrate that HBU outperforms both TGS and TPR bulk loading algorithms in terms of query I/Os for all kinds of non-uniform datasets and is scalable. Future work includes generalization of this histogram based algorithm to other R-tree variants and maintaining histograms for tree reshaping after bulk loading.

References

[1] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data structures*, 1995.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[3] 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. ACM SIGMOD Int. Conf. on Management of Data*, 1990.

[4] M. Cai and P. Z. Revesz. Parametric R-tree: An index structure for moving objects. In *Proc. 10th COMAD Int. Conf. Management of Data*, pages 57–64, 2000.

[5] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proc. 9th Australasian Database Conference (ADC'98)*, pages 15–26, Perth, Australia, 1998.

[6] Y. J. Garcia, M. A. Lopez, and S. T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *Proc. ACM Symp. on Advances in Geographic Information Systems*, pages 163–164, 1998.

[7] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 471–475, 2001.

[8] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Varirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1), 2000.

[9] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. Int. Conf. on Information and knowledge Management*, pages 490–499, 1993.

[10] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 261–272, 1999.

[11] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Trans. on Knowledge and Data Engineering*, 10(1):1–20, 1998.

[12] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *Proc. Int. Conf. on Data Engineering*, pages 497–506, 1997.

[13] J. Moreira, C. Ribeiro, and J. Saglio. Representation and manipulation of moving points: An extended data model for location estimation. *Cartography and Geographical Information Systems*, 26, 1999.

[14] M. A. Nascimento. Personal communications, 2003.

[15] D. Papadopoulos, G. Kollios, D. Gunopulos, and V. J. Tsotras. Indexing mobile objects on the plane. In *Proc. Int. Workshop on Database and Expert Systems Applications (DEXA)*, pages 693–697, 2002.

[16] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-tree: An efficient self-adjusting index for moving objects. In *Proc. Int. Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 178–193, 2002.

[17] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 17–31, 1985.

[18] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proc. Int. Conf. on Data Engineering*, pages 463–472, 2002.

[19] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *TimeCenter Technical Report*, 1999.

[20] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continously moving objects. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 331–342, 2000.

[21] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. Int. Conf. on Data Engineering*, 1997.

[22] Z. Song and N. Roussopulos. K-nearest neighbor search for moving query point. In Springer, editor, *Advances in Spatial and Temporal Databases*, pages 79–97, Redonodo Beach, CA, July 2001.

[23] C. Sun, D. Agrawal, and A. El Abbadi. Exploring spatial datasets with histograms. In *Proc. Int. Conf. on Data Engineering*, 2002.

[24] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. Int. Conf. on Very Large Data Bases*, 2003.

[25] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[26] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatio-temporal datasets. In *Proc. Symp. on Large Spatial Databases*, 1999.

[27] J. Van den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *Proc. Int. Conf. on Very Large Data Bases*, pages 461–470, 2001.

[28] O. Wolfson, S. Chamberlain, S. Dao, and L. Jiang. Location management in moving objects databases. In *Proc. the Second International Workshop on Satellite-Based Information Services (WOSBIS'97)*, Budapest, Hungary, Oct. 1997.

[29] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: issues and solutions. In *Proc. Int. Conf. on Statistical and Scientific Database Management*, pages 111–122, 1998.

[30] H. Zhu, J. Su, and O. Ibarra. Trajectory queries and octagons in moving object databases. In *Proc. ACM Conference on Information and Knowledge Management (CIKM)*, pages 413–421, Nov. 2002.