Indexing Mobile Objects Using Dual Transformations

George Kollios*
Boston University
gkollios@cs.bu.edu

Dimitris Papadopoulos
UC Riverside
tsotras@cs.ucr.edu

Dimitrios Gunopulos†
UC Riverside
dg@cs.ucr.edu

Vassilis J. Tsotras‡
UC Riverside
tsotras@cs.ucr.edu

Abstract

With the recent advances in wireless networks, embedded systems and GPS technology, databases that manage the location of moving objects have received increased interest. In this paper, we present indexing techniques for moving object databases. In particular, we propose methods to index moving objects in order to efficiently answer range queries about their current and future positions. This problem appears in real-life applications, such as predicting future congestion areas in a highway system, or allocating more bandwidth for areas where high concentration of mobile phones is imminent. We address the problem in external memory and present dynamic solutions, both for the one-dimensional, as well as the two-dimensional cases. Our approach transforms the problem into a dual-space that is easier to index. Important in this dynamic environment is not only query performance but also the update processing, given the large number of moving objects that issue updates. We compare the dual transformation approach with the TPR-tree, an efficient method for indexing moving objects that is based on time-parameterized index nodes. An experimental evaluation shows that the dual transformation approach provides comparable query performance but has much faster update processing. Moreover, the dual method does not require establishing a predefined query horizon.

Keywords: Spatiotemporal Databases – Access Methods – Mobile Objects

*Supported by NSF CAREER Award 0133825.
†Supported by NSF CAREER Award 9984729, NSF IIS-9907477, and the DoD.
‡Supported by NSF IIS-9907477, NSF EIA-9983445 and the DoD.

1 Introduction

A spatiotemporal database system manages data whose geometry changes over time. There are many applications that create such data, including global change (as in climate or land cover changes), transportation (traffic surveillance data, intelligent transportation systems), social (demographic, health, etc.), and multimedia (animated movies) applications. In general, one could consider two spatial attributes of spatiotemporal objects which are time dependent, namely: position (i.e., the object's location inside some reference space) and extent (i.e., the area or volume the object occupies in the reference space)[19]. Depending on the application, one or both spatial attributes may change over time. Examples include: an airplane flying around the globe, a car traveling on a highway, the land covered by a forest as it grows/shrinks over time, or an object that concurrently moves and changes its size in an animated movie. For the purposes of this paper we concentrate on applications with objects which change position over time but whose extent remains unchanged. Hence for our purposes we represent such objects as points moving in some reference space ("mobile points").

The usual assumption in traditional database management systems is that data stored in the database remains constant until explicitly changed by an update. For example, if a price field is 5, it remains 5 until explicitly updated. This model is appropriate when data changes in discrete steps, but it is inefficient for applications with continuously changing data [39]. Consider for example a database keeping the position of mobile objects (like automobiles). The primary goal of this database is to correctly represent reality as objects move. On the one hand, updating the database about each object's position at each unit of time is clearly an inefficient and infeasible solution due to the prohibitively large update overhead. On the other hand, updating the database only at few, representative time instants limits query accuracy.

A better approach is to abstract each object's location as a function of time $f(t)$, and update the database only when the parameters of f change (for example when the speed or the direction of a car changes). Using $f(t)$ the "motion" database can compute the location of the mobile object at any time in the future. While this approach minimizes the update overhead, it introduces a variety of novel problems (such as the need for appropriate data models, query languages and query processing and optimization techniques) since the database is not directly storing data values but functions to compute these values. Motion database problems have recently attracted the interest of the research community: ([39, 46, 47]) present the Moving Objects Spatio-Temporal (MOST) model and a language (FTL) for querying the current and future locations of mobile objects; ([19]) proposes a model that tracks and queries the history (past routes) of mobile objects, based on new spatio-temporal data types. Another spatiotemporal model appears in [11]. Spatio-temporal queries about mobile objects have important applications in traffic monitoring, intelligent navigation and mobile communications domains. For example, if we use a database to track cars in a highway system, it would be useful to be able to detect future congestion areas efficiently. In mobile communication systems, we could allocate more bandwidth in areas where high concentration of mobile phones is approaching.

In this paper we focus on the problem of indexing mobile objects. In particular we examine how to efficiently address range queries over the object locations into the future. An example of such a spatio-temporal query is: "Report all the objects that will be inside a query region P after 10 minutes from now". Note that the answer to these queries is tentative in the sense that it is computed based on the current knowledge stored in the database about the mobile objects' location functions. In the near future this knowledge may change, which implies that the same query could have a different answer.

As the number of mobile objects in the applications we consider (traffic monitoring, mobile communications, etc.) can be rather large, we are interested in external memory solutions. Furthermore, since we deal with highly dynamic data, we pay special attention to the *updatability* of our methods. Note that, although

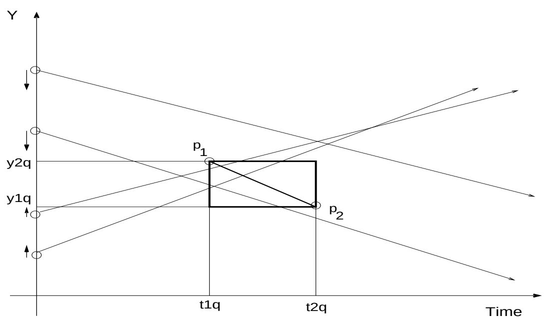

Figure 1: Trajectories and query in (t, y) plane.

using functions of time to represent the location of moving objects will decrease the update overhead, still many objects may change their functions at each time instant. In many applications the number of updates is expected to be order of magnitudes larger than the number of queries. Therefore, we consider the update overhead to be an important measure of the quality and applicability of the proposed methods.

We present methods for indexing moving objects that have good worst case performance. Also, we present more practical methods that are evaluated with an extensive experimental study. Our methods are based on the dual transformation [24, 47], where the initial location of the moving objects along with their trajectories are mapped to points in a multidimensional space. By mapping the moving objects into a dual space, we are able to design more efficient algorithms that achieve a good trade-off between query and update overhead.

The rest of the paper is organized as follows: Section 2 provides a formal problem description and describes the dual transformation, which is the core of our approach. Section 3 presents the related work, while the one-dimensional case is addressed in Section 4. The technique for indexing objects that move freely in two dimensions is described in section 5. Experimental results, along with discussion pointing out the advantages and drawbacks of the methods that employ indexing techniques in the primal space and the dual space, follows in Section 6. Finally, Section 7 concludes the paper.

2 Preliminaries

In this section we formally define the problem of indexing two dimensional moving objects. Next, we present a geometric duality transform that is used as the basis of our solutions.

2.1 Problem Definition

We consider a database that records the position of mobile objects in one and two dimensions. Following [47, 35, 24], we assume that an object's movement can be represented (or approximated) with a linear function of time. For each object we store an initial location, a starting time instant and a velocity vector (speed and direction). Therefore, we can calculate the future position of the object, provided that the characteristics of its motion remain the same. Objects update their motion information, when their speed or direction changes. We assume that the objects can move inside a finite domain (a line segment in one dimension or a rectangle in two). Furthermore, the system is dynamic, i.e. objects may be deleted or new objects may be inserted.

Let $P(t_0) = [x_0, y_0]$ be the initial position of an object at time t_0. Then, the object starts moving and at time $t > t_0$ its position will be $P(t) = [x(t), y(t)] = [x_0 + v_x(t - t_0), y_0 + v_y(t - t_0)]$, where $V = [v_x, v_y]$ is its

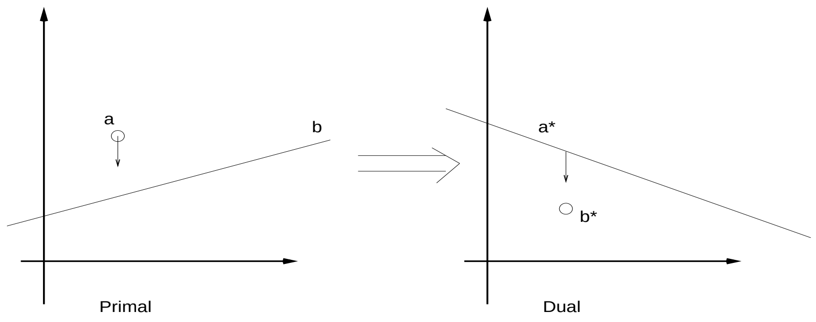

Figure 2: Hough-X dual transformation: primal plane (left), dual plane (right)

velocity vector. An example for the one-dimensional case is shown in Figure 1.

We would like to answer queries of the form: "Report the objects located inside the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$ at the time instants between t_{1q} and t_{2q} (where $t_{now} \leq t_{1q} \leq t_{2q}$), given the current motion information of all objects" (i.e. the *two dimensional MOR query* [24]).

We use the standard external memory model of computation [4] to study the theoretical aspects of the problem. In this model each disk access (an I/O) transmits in a single operation B units of data, i.e. B is the page capacity. We measure the efficiency of an algorithm in terms of the number of I/O's needed to perform an operation. If N is the number of the mobile objects and K is the number of objects reported by the MOR query, then the number of pages required to store the database is at least $n = \lceil \frac{N}{B} \rceil$ and the number of I/O's to report the answer is at least $k = \lceil \frac{K}{B} \rceil$. We say that an algorithm uses linear space, if it uses $O(n)$ disk pages, and that it uses logarithmic time to answer a query if it needs to perform $O(\log_B n + k)$ I/O's. Note that $\log_B n$ is for the external memory model different than $\log_2 n$, since B is not a problem constant but a problem parameter.

2.2 The dual space-time representation

In this section we present the dual transformation that we use later to index moving objects. In general, the dual transformation is a method that maps a hyper-plane h from R^d to a point in R^d and vice-versa. In this section we briefly describe how we can address the problem at hand in a more intuitive way, by using the dual transform for the one-dimensional case.

Specifically, a line from the primal plane (t, y) is mapped to a point in the dual plane. A class of transforms with similar properties may be used for the mapping. The problem setting parameters determine which one is more useful.

One dual transform for mapping the line with equation $y(t) = vt + a$ to a point in R^2 is to consider the dual plane where one axis represents the slope of an object's trajectory (i.e. velocity) and the other axis its intercept (Figure 2) Thus we get the dual point (v, a) (this is called Hough-X transform in [21]). Similarly, a point $p = (x, y)$ in the primal space is mapped to line $a(v) = -xv + y$ in the dual space. An important property of the duality transform is that it preserves the above-below relationship. As it is shown in Figure 2, the dual line of point a is above the dual point b^* of the line b.

Based on the above property, it is easy to show that the 1-d query $[(y_{1q}, y_{2q}), (t_{1q}, t_{2q})]$ becomes a polygon in the dual space. Consider a point moving with positive velocity. Then, the trajectory of this point intersects the query if and only if it intersects the segment defined by the points $p_1 = (t_{1q}, y_{2q})$ and $p_2 = (t_{2q}, y_{1q})$

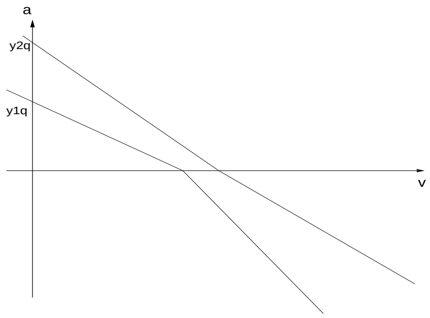

Figure 3: Query on the Hough-X dual plane.

(Figure 1). Thus, the dual point of the trajectory, must be above the dual line p_2^* and below p_1^*. The same idea is used for the negative velocities. Therefore, using a linear constraint query [17], the query Q in the dual Hough-X plane (Figure 3) is expressed in the following way:

- If $v > 0$, then $Q = C_1 \wedge C_2$, where:
  - $C_1 = a + t_{2q}v \geq y_{1q}$ and $C_2 = a + t_{1q}v \leq y_{2q}$
- If $v < 0$, then $Q = D_1 \wedge D_2$, where:
  - $D_1 = a + t_{1q}v \geq y_{1q}$ and $D_2 = a + t_{2q}v \leq y_{2q}$

By rewriting the equation $y = vt + a$ as $t = \frac{1}{v}y - \frac{a}{v}$, we can arrive to a different dual representation. Now the point in the dual plane has coordinates (b, n), where $b = -\frac{a}{v}$ and $n = \frac{1}{v}$ (Hough-Y in [21]). Coordinate b is the point where the line intersects the line $y = 0$ in the primal space. By using this transform, horizontal lines cannot be represented. Similarly, the Hough-X transform cannot represent vertical lines. Therefore, for static objects, we can use only the Hough-X transform.

3 Related work

The straightforward approach of representing an object moving on an 1-dimensional line is by plotting the trajectories as lines in the time-location (t, y) plane (same for (t, x) plane). The equation describing each line is $y(t) = vt + a$ where v is the slope (velocity in this case) and a is the intercept, which is computed using the motion information (Figure 1). In this setting, the query is expressed as the 2-dimensional interval $[(y_{1q}, y_{2q}), (t_{1q}, t_{2q})]$, and it reports the objects that correspond to the lines intersecting the query rectangle.

The space-time approach provides an intuitive representation. Nevertheless, it is problematic, since the trajectories correspond to long lines. Using traditional indexing techniques in this setting tends to show many drawbacks. Consider for example using a Spatial Access Method, such an R-tree [20] or an R*-tree [8]. In this setting each line is approximated by a minimum bounding rectangle (MBR). Obviously, the MBR approximation has much larger area than the line itself. Furthermore, since the trajectory of an object is valid until an update is issued, it has a starting point but no end. Thus all trajectories expand till "infinity", i.e. they share an ending point on the time dimension.

Another approach is to partition the space into disjoint cells and store in each cell those lines that intersect it [45, 12]. This could be accomplished by using an index such an R+-tree [38], a cell-tree [18], and the

PMR-quadtree [37]. The shortcoming of these methods is that they introduce replication, since each line is copied into the cells that intersect it. Given that lines are typically long, the situation becomes even worse. Moreover, using space partitioning would also result in high update overhead, since when an object changes its motion information, it has to be removed from all cells that store its trajectory.

Agarwal et al. [1] proposed the use of multi-level partition trees [1] to index moving objects using the duality transform, in order to answer range queries at a specific time instant (i.e. snapshot queries, where $t_{1q} = t_{2q}$). They decompose the motion of the objects on the plane, by taking the projections on the (t, x) and (t, y) planes. They construct a primary partition tree T^x to keep the dual points corresponding to the motion projected on the (t, x) plane. Then at every node v of T^x they attach a secondary partition T^y_v for the points S^y_v with respect to the (t, y) projection , where S_v is the set of points stored in the primary subtree rooted at v. The total space used by the index is $O(n \log_B n)$, where N is the number of objects, B is the page capacity and $n = N/B$. The query is answered by decomposing it into two sub-queries, one on each of the two projections, and taking the dual of them, σ^x and σ^y, respectively. The search begins by searching the primary partition T^x for the dual points, with respect to the (t, x) projection, that satisfy the query σ^x. If it finds a triangle associated with a node v of the partition tree T^x that lies completely inside σ^x, then it continues searching in the secondary tree T^y_v and reports all dual points, with respect to (t, y) projection, that satisfy the query σy. The query is satisfied, if and only if the query in both projections is satisfied. This is true for snapshot range queries. In [1] it is shown that the query takes $O(n^{\frac{1}{2}+\epsilon} + K/B)$ I/Os (here K is the size of the query result) and that the size of the index can be reduced to $O(n)$ without affecting the asymptotic query time. Furthermore, by using multiple multilevel partition trees, is is also shown that the same bounds hold for the window range query.

Elbassioni et al. [15] proposed a technique (MB-index) that partitions the objects along each dimension in the dual space, and uses B-trees in order to index each partition. Assuming a set of N objects moving in d-dimensional space, with uniformly distributed and independent velocities and initial positions, , they proposed a scheme for selecting the boundaries of the partitions and answering the query, that yields $O(n^{1-1/3d} * (\sigma \log_B n)^{1/3d} + k)$ average query time, using $O(n)$ space ($n = N/B$, $k = K/B$). The total number of B-trees used is $\sigma 3^d s^{2d-1}$, where $\sigma = \prod_{i=1}^{d} \ln(v_{i,max}/v_{i,min})$ and $s = (\frac{n}{\log_B n})^{\frac{1}{d}}$.

Saltenis et al. [35] presented another technique to index moving objects. They proposed the time-parameterized R-tree (TPR-tree), which extends the R*-tree. The coordinates of the bounding rectangles in the TPR-tree are functions of time and, intuitively, are capable of following the objects as they move. The position of a moving object is represented by its location at a particular time instant (reference position) and its velocity vector. The bounding intervals employed by the TPR-tree are not always minimum, since the storage cost would be excessive. Even though it would be the ideal case (if the bounding intervals were kept always minimum), doing so could deteriorate to enumerating all the enclosed moving points or rectangles. Instead, the TPR-tree uses ”conservative” bounding rectangles, which are minimum at some time point, but not at later times. The bounding rectangles may be calculated at load-time (i.e. when the objects are first inserted into the index), or when an update is issued. As pointed out in [34], the TPR-tree with load-time bounding rectangles is equivalent to the dual space-time representation. It performs best, only when update-time bounding rectangles are used.

The TPR-tree assumes a predefined time horizon H, from which all the time instances specified in the queries are drawn. This implies that the user has good knowledge of (or can efficiently estimate) H. The horizon is defined as $H = UI + W$, where UI is the average time interval between two updates, and W

[1] Partition trees group a set of points into disjoint subsets denoted by triangles. A point may lie into many triangles, but it belongs to only one subset.

is the querying window. The insertion algorithm of the R*-tree, which the TPR-tree extends to moving points, aims at minimizing objective functions such as the areas of the bounding rectangles, their margins (perimeters), and the overlap among the bounding rectangles. In the case of the TPR-tree, these functions are time dependent, and their evolution in $[t_l, t_l + H]$ is considered, where t_l is the time instance when the index is created. Thus, given an objective function $A(t)$, instead of minimizing the objective function, the integral $\int_{t_l}^{t_l+H} A(t)dt$ is minimized.

The STAR-tree, introduced by Procopiuc et al. [33], is also a time parameterized structure. It is based upon R-trees, but it does not use the notion of the horizon. Instead it employs kinetic events to update the index when the bounding boxes start overlapping a lot. If the bounding boxes of the children of a node v overlap considerably, it re-organizes the grand children of v among the children of v. Using geometric approximation techniques developed in [3], it maintains a time parameterized rectangle $A_v(t)$ which is a close approximation of $R_v(t)$, the actual minimum bounding rectangle of node v at any time instant t in to the future. It provides a trade-off between the quality of $A_v(t)$ and the complexity of the shape of $A_v(t)$. For linear motion, the trajectories of the vertices of $A_v(t)$ can be represented as polygonal chains. In order to guarantee that $A_v(t)$ is an ϵ-approximation of $R_v(t)$, trajectories of the corners of $A_v(t)$ need $O(1/\sqrt{\epsilon})$ vertices. An ϵ-approximation means that the projection of the $A_v(t)$ on (x, t) or (y, t) planes contains the corresponding projections of $R_v(t)$ but it is not larger than $1 + \epsilon$ than the extend on the $R_v(t)$ at a any time instant.

The R^{EXP}-tree, which extends the TPR-tree, was proposed to index moving objects with expiration time in [36]. The operations are similar to those of the TPR-tree. Special care is taken when an objective function has to be minimized in the insertion algorithms, since now the expiration time of the entries have to be taken into account. Also, an algorithm for maintaining the horizon dynamically is provided. Furthermore, regarding the removal of expired entries, a lazy strategy is employed. Only live entries are considered during search, insertion, and deletion operations, but expired entries are physically removed from a node only when the contents of the node is modified and the node is written to disk. In addition, when an expired entry in an internal node is discarded, either when writing the node to the disk or deallocating it, the whole subtree rooted at this entry has to be deallocated.

Algorithms to process nearest neighbor queries using the dual transformation are presented in [23]. Such queries (as well as range) are also examined in [31] where techniques using indexing in the primal space are presented. Song et al. [40] propose a sampling technique for moving point nearest neighbor queries. They incrementally compute the results at predefined positions, using previous results to avoid re-computation. This approach has limitations, since they deal with static objects. Also it inherits the usual limitations of sampling, i.e. if the sampling rate is low the results will be incorrect, otherwise there is a significant computational overhead. Furthermore, there is no accuracy guarantee since even a high sampling rate may miss some results.

Tao et al. [42] address the problem of time-parameterized queries in a moving objects environment. Time-parameterized queries retrieve the actual result at the time of the query is issued, the validity period of the result given the current motion of the query and the database objects, as well as the change that causes the expiration of the result. In that context, they propose techniques to answer window queries, k-nearest neighbor queries and spatial joins. Their techniques employ branch-and-bound algorithms on TPR-trees. Improved algorithms for nearest neighbor time parameterized queries are presented in [43]. Related is also work on dynamic queries over mobile objects [25]. Here queries are assigned to mobile observers and the result changes as the observer moves; query processing techniques that reuse previous stored results are presented. Recently, continuous range queries in the spatiotemporal environment have been addressed in [22].

Prabhakar et al. [32] proposed two techniques for answering continuous queries on moving objects, namely query indexing and velocity constrained indexing (VCI). Query indexing relies on reversing the role of queries and data, that is, instead of indexing the objects, an index on the queries is built, while the data reside in flat files. Also it involves incremental evaluation of queries, and exploits the relative locations of objects and queries. On the other hand, VCI takes into consideration the maximum possible speed of objects in order to delay the expensive operation of updating an index to reflect the movement of objects. [32] proposed a scheme that combines the two techniques, in order to facilitate processing of ongoing queries and fast updates.

Pfoser et al. [30] propose two R-tree based schemes for indexing the past trajectories of the moving objects and asking historical queries, assuming that their motion is piecewise linear. For each object o_i, let Γ_i denote the set of line segments of its trajectory, and let $\Gamma = \bigcup \Gamma_i$. The first index, called STR-tree, considers each segment of Γ independently and builds an R-tree on them. They introduce new heuristics to split a node, which take the trajectories of the objects into account while inserting a new segment into the tree. Since the segments of a trajectory are stored at different parts of the tree, updating a trajectory is expensive. In the second index, called the TB-tree, they alleviate this drawback by storing all line segments of the same trajectory at the same leaf of the index. [48] presents an approach to index trajectories that divides the trajectory predicates in topological and non-topological parts. Moreover, minimum bounding octagons are introduced as a better approximation to traditional MBRs.

Work regarding the selectivity estimation of queries on moving objects appear in [10] and [44]. In the first work, Choi et al. [10] address the problem in the context of dynamic point data and static queries (i.e. the query region remains fixed), and they begin from the one-dimensional case. Assuming that the locations, as well as the velocity, of the objects that move on a line segment follow a uniform distribution, they derive the probability that a point qualifies the query, hence the selectivity of the query. The multi-dimensional case is reduced to the one-dimensional case by projecting objects and queries onto individual dimensions. Having computed the selectivity for each one of the one-dimensional cases, the general probability that a point qualifies a query is given as the product of the individual 1D selectivities (i.e. the probability that the projection p_i of point p on the i-th dimension intersects the projection q_i of the query during the query time interval q_t). This approach in general may not be accurate, since a data point may still violate a query q, even if its projection intersects that of q on every dimension. It is not sufficient that only the spatial conditions should hold; the intersection time intervals on all dimensions must also overlap, i.e. the temporal condition should also hold.

Tao et al. [44] propose cost models for selectivity estimation of spatio-temporal window queries. They address the problem dealing both with points and rectangles, and they allow both the objects and the query to be dynamic with respect to time. Apart from assuming uniformity, they also extend their results to non-uniform datasets by employing spatio-temporal histograms, which in addition to the locations of the objects, also consider the velocity distributions during partitioning.

In [7] a main memory framework (kinetic data structure) was proposed that addresses the issue of mobility and maintenance of configuration functions among continuously moving objects. The main idea of this work is that even though the objects move continuously, the relevant combinatorial structure changes only at certain discrete time, for instance when points pass each other. Using this observation, future events are scheduled that update a data structure at these times so that necessary invariants of the structure hold. Application of this framework to external range trees [5] appears in [1], where a structure is presented that can answer snapshot range queries in $O(\log_B n + K/B)$ I/O's using slightly more than a linear number of disk blocks. This result holds only when queries arrive in chronological order; once a kinetic event has changed the data structure, no queries can refer to time points before the event. Non-chronological queries are addressed using partial persistence techniques. Furthermore, in that work it is shown how to combine kinetic range trees with partition trees to achieve a trade-off between the number of kinetic events and query performance.

Finally, frameworks for moving object databases, such as the Moving Objects Spatio-Temporal (MOST) model and a language (FTL) for querying the current and future locations of moving objects, are presented in [39, 46, 47]. In another recent work, Güting et al. [19] propose a DBMS data model and query language capable of handling time-dependent geometries that describe moving objects. They formally define the types and operations necessary for implementing a spatio-temporal DBMS extension. A query language for moving object environments, based on generalized distances is presented in [27]. Plane sweeping methods for evaluating queries in this language are also suggested.

4 Indexing in one dimension

In this section we illustrate techniques for the one-dimensional case, i.e., for objects moving on a line segment. There are various reasons for examining the 1-dimensional case. First, the problem is simpler and can give good intuition about the various solutions. It is also easier to prove lower bounds and approach optimal solutions for this case. Moreover, it can have practical uses as well. A large highway system can be approximated as a collection of smaller line segments (this is the 1.5 dimensional problem discussed in [24]), on each of which we can apply the 1-dimensional methods.

4.1 A Lower Bound

By using the dual space-time representation, the problem of indexing moving objects on a line is transformed into the problem of *simplex* range searching in two dimensions. In simplex range searching we are given a set S of 2-dimensional points, and we want to answer efficiently queries of the following form: given a set of linear constraints $ax \leq b$, find all points in S that satisfy all the constraints. Geometrically, the constraints form a polygon on the plane, and we want to find the points in the interior of the polygon.

The only known lower bound for simplex range searching, if we want to report all the points that fall in the query region rather than their number, is due to Chazelle and Rosenberg ([9]). They show that simplex reporting in d-dimensions with a query time of $O(N^{\delta} + K)$, where N is the number of points, K is the number of reported points and $0 < \delta \leq 1$, requires space $\Omega(N^{d(1-\delta)-\epsilon})$, for any fixed ϵ. This result is shown for the pointer machine model of computation. The bound holds for the static case, even if the query region is the intersection of just two hyper-planes. Since ϵ can be arbitrary small, any algorithm that uses linear space for d-dimensional range searching has worst case query time of $O(N^{(d-1)/d} + K)$.

Here we show that a similar bound holds for the input-output complexity of simplex searching. Following the approach in [41] we use the external memory pointer machine as our model of computation. This is a generalization of the pointer machine suitable for analyzing external memory algorithms. In this model, a data structure is modeled as a directed graph $G = (V, E)$, with a source w. Each node of the graph represents a disk block and is therefore allowed to have B data and pointer fields. The points are stored in the nodes of G. Given a query, the algorithm traverses G starting from w, examining the points at the nodes it visits. The algorithm can only visit nodes that are neighbors of already visited nodes (with the exception of the root) and, when it terminates the answer to the query must be contained in the set of visited nodes. The running time of the algorithm is the number of nodes it visits.

Theorem 1 *Simplex reporting in d-dimensions with a query time of* $O(n^{\delta} + k)$ *I/O's, requires* $\Omega(n^{d(1-\delta)-\epsilon})$

disk blocks, for any fixed ϵ; here N is the number of points, $n = N/B$, K is the number of reported points, $k = K/B$, and $0 < \delta \leq 1$.

Proof: To prove the lower bound we need to show that, given δ, there exists a set of N points, and a set of $\Omega(n^{d(1-\delta)-\delta-\epsilon})$ queries, such that each query has $\Theta(Bn^{\delta})$ points, and the intersection of any pair of query results is small. To answer a query with $\Theta(Bn^{\delta})$ points, the answering algorithm must visit $\Omega(n^{\delta})$ nodes. To answer this query in $O(n^{\delta})$ I/O's, at least a constant fraction of that many blocks have a constant fraction of their points in the answer of the query. But if the set of the queries has small intersection, it follows that in order to answer this set of queries in time $O(n^{\delta})$ at least $\Theta(n^{\delta}) \times \Omega(n^{d(1-\delta)-\delta-\epsilon}) = \Omega(n^{d(1-\delta)-\epsilon})$ nodes have to be visited. It remains to show that such a set of queries exist. To do so we simply modify the existing construction in [9] by replacing each point in the point set by B copies. ∎

A corollary of this lower bound is that in the worst case a data structure that uses linear space to answer the 2-dimensional simplex range query and thus the 1-dimensional MOR query, requires $O(\sqrt{n} + k)$ I/O's. Next we will present a dynamic, external-memory algorithm that achieves almost optimal query time with linear space. As we shall see, however, this algorithm is not practical. So we also consider faster algorithms to approximate the queries. Finally we give a worst case logarithmic query time algorithm for a restricted but practical version of the problem.

4.2 An (Almost) Optimal Solution

Matousek [26] gave an almost optimal algorithm for simplex range searching, given a static set of points. This main memory algorithm is based on the idea of simplicial partitions.

We briefly describe this approach here. For a set S of N points, a simplicial partition of S is a set $\{(S_1, \Delta_1), \ldots (S_r, \Delta_r)\}$ where $\{S_1, \ldots, S_r\}$ is a partitioning of S, and Δ_i is a triangle that contains all the points in S_i. If $max_i|S_i| < 2min_i|S_i|$, where $|S_i|$ is the cardinality of the set S_i, we say that the partition is balanced. Matousek [26] shows that, given a set S of N points, and a parameter s (where $0 < s < N/2$), we can construct in linear time, a balanced simplicial partition for S of size $O(s)$ such that any line crosses at most $O(\sqrt{s})$ triangles in the partition.

This construction can be used recursively to construct a partition tree for S. The root of the tree contains the whole set S, and a triangle that contains all the points. We find a balanced simplicial partition of S of size $\sqrt{|S|}$. Each of the children of the root are associated with a set S_i from the simplicial partition, and the triangle Δ_i that contains the points in S_i. For each of the S_i's we find simplicial partitions of size $\sqrt{|S_i|}$, and continue until each leaf contains a constant number of points. The construction time is $O(N \log_2 N)$.

To answer a simplex range query, we start at the root. We take each of the triangles in the simplicial partition at the root and check if (i) it is inside the query region, (ii) it is outside the query region, or, (iii) it intersects one of the lines that define the query. In the first case all points inside the triangle are reported, in the second case the triangle is discarded, while in the third case we continue the recursion on this triangle. The number of triangles that the query can cross is bounded, since each line crosses at most $O(|S|^{\frac{1}{4}})$ triangles at the root. The query time is $O(N^{\frac{1}{2}+\epsilon})$, with the constant factor depending on the choice of ϵ.

Agarwal et al. [2] give an external memory version of static partition trees that answers queries in $O(n^{\frac{1}{2}+\epsilon} + k)$ I/Os. To adapt this structure to our environment, we have to make it dynamic. Using a standard technique by Overmars [28] for decomposable problems we can show that we can insert or delete points in a partition tree in $O(\log_2^2 N)$ I/Os and answer simplex queries in $O(n^{\frac{1}{2}+\epsilon} + k)$ I/O's. A method that achieves $O(\log_B^2(\frac{N}{B}))$ amortized update overhead is presented in [1].

4.3 Achieving Logarithmic Query Time

For many applications, the relative positions of the moving objects do not change often. Consider for example the case where objects are moving very slowly, or with approximately the same velocity. In this case the lines in the time-space plane do not cross until well forward in the future. If we restrict our queries to occur before the first time that a point overtakes (passes) another, the original problem is equivalent to 1-dimensional range searching.

This is one of our motivations to consider a restricted version of the original problem, namely to index mobile objects in a bounded time interval T in the future. As we have seen, there exist lower bounds for the original problem which show that we cannot achieve query time better than $\Omega(\sqrt{n})$ given linear space. However, using the above restriction, we achieve a logarithmic query time, with space that can be quadratic in the worst case, but is expected to be linear in practice.

Formally, the problem we are considering in this section is the following: given a set of objects that are moving on a line, and a time limit T, find all the objects that lie in the segment $[y_l, y_r]$ at time t_q (where $t_0 \leq t_q \leq t_0 + T$). Equivalently, this a standard one dimensional MOR query where $t_{1q} = t_{2q}$. We will call it an one dimensional MOR1 query.

Our method is to find all the times when an object overtakes another. These events correspond to line segment crossings in the time-space plane. Note that between two consecutive crossing events the relative ordering of the objects on the plane remains the same.

First we show the following lemma:

Lemma 1 *If we have the relative ordering of all the objects at time t_q, the position of the objects at time T_c that corresponds to the closest crossing event before t_q, and the speed of the objects, we can find the objects that are in $[y_l, y_r]$ in $O(log_2 N + K)$ time.*

Proof: Assume that the objects are $\{p_1, p_2, \ldots, p_N\}$, where p_i has a position y_i at time T_c and a velocity v_i. Without loss of generality, assume that, at time t_q, the relative order of the objects from left to right is $p_1, p_2, \ldots, p_N$.

Consider a binary tree storing the objects sorted by their original positions at time T_c. The object at the root of the tree, say p_i, is going to be at position $y_i + v_i \times t_q$ at time t_q. Since the objects in the binary tree are stored by order at the time t_q, if $y_i + v_i \times t_q < y_l$ then this is also true for all the objects to the left child of the root, in which case we eliminate the left child and recurse in the right child. Otherwise we recurse on the left child of the tree. Thus in $O(log_2 N)$ time we can find the positions of y_l and y_r relative to the objects at time t_q, and we report the objects that lie between. ■

The following lemma finds all object crossings efficiently.

Lemma 2 *We can find all object crossings in time $O(Nlog_2 N + Mlog_2 M)$, where M is the number of crossings in the time period $[0, T]$.*

Proof: Let $\{p_1, \ldots, p_N\}$ be the ordering of the N objects at time 0, sorted by their position. Assume we maintain this ordering in a linked list L_0. At time T, the position of object i is $y_i + v_i \times T$. Assume we order the object positions as of time T, and keep them in another linked list L_T; let $\{p_{t(1)}, \ldots, p_{t(N)}\}$ be this ordering. Clearly, objects i and j $(i < j)$ cross if and only if $t(j) < t(i)$.

The algorithm to find all M crossings follows. The first object p_1 is read from L_0 and removed from this list. List L_T is scanned until the position of object p_1 is found; all the crossings from this object are then

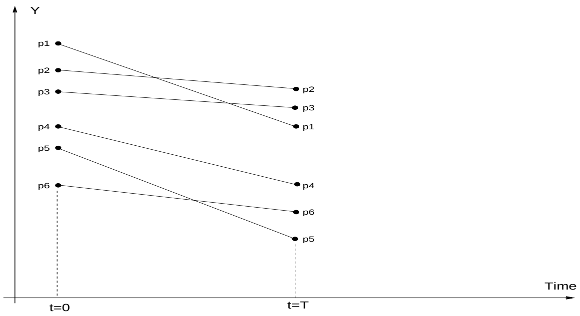

Figure 4: Object trajectory crossings.

reported. Object p_1 is removed from L_T and the process is repeated for the next item in L_0. This procedure reports all M crossings in $O(N + M)$ time [13]. After all crossings are reported they are sorted by the time when each crossing occurred. ∎

An example is shown in figure 4; here $N = 6$ and $M = 3$. From the order of the object positions at time T we can easily find that object p_1 crossed objects p_2 and p_3 while p_5 crossed object p_6.

In the next lemma we show how we can efficiently store and search these lists in external memory.

Lemma 3 *We can store the $O(M)$ ordered lists of N objects in $O(n + m)$ blocks and perform a search on any list in $O(log_B(n + m))$ I/O's, where $n = \frac{N}{B}$ and $m = \frac{M}{B}$.*

Proof: Let $L(t)$ be the list of objects at time t. Consider $CS = t_1, \ldots, t_M$ the ordered sequence of the time instants where crossings occur during the interval $(0, T)$. The problem of storing the M ordered lists $L(t_1)$ through $L(t_M)$ can be "visualized" as storing the history of a list $L(t)$ that evolves over time, i.e., a partial persistence problem [14]. That is, list $L(t)$ starts from an initial state $L(0)$ and then evolves through consecutive states $L(t_1), L(t_2), \ldots, L(t_M)$, where $L(t_{i+1})$ is produced from $L(t_i)$ by applying the crossing that occurred at t_{i+1} ($i = 0, \ldots, M - 1$, and $t_0 = 0$).

A common characteristic in the list evolution is that each $L(t)$ has exactly N positions, namely positions 1 through N, where position j stores the $j - th$ element of $L(t)$. To perform a binary search on a given $L(t)$ we could implement it using a binary tree with N nodes, where each node is numbered by a position (the root node corresponds to the middle position in the list and so on) and holds the element of $L(t)$ at that position. One obvious solution to the problem would be to store the binary tree of the original list $L(0)$ and the binary tree of each $L(t_i)$ for all t_i in CS. Then, a query about list $L(t)$ is addressed by using the binary tree of $L(t_i)$, where t_i is the largest instant in CS that is less or equal to t. While this achieves $O(log_2(N + M))$ query time, it uses (MN) space.

To reduce the space to $O(N + M)$ we must take advantage of the fact that subsequent lists do not differ much. A main-memory solution to this problem appears in [13]. Here we present an efficient external memory solution. In particular, we first embed the binary tree structure inside a B-tree. This is easily done since the structure of the list (and its corresponding binary tree) does not change over time. Consider for example $B(0)$ that corresponds to the initial list $L(0)$. Tree $B(0)$ uses $O(n)$ nodes where each node can hold B entries. An entry is now a record (*position, occupant, pointer, t*), where *position* corresponds to a position in the list, *occupant* contains the element at that position, *pointer* points to a child node and *t* corresponds to the time this element was at that position, in this case $t = 0$.

Conceptually, each B-tree node is permanently assigned B positions and is responsible for storing the occupants of these positions. Consider the evolution of such a node s through trees $B(0), B(t_1), \ldots, B(t_M)$. An obvious way to store this evolution is to store a copy of $s(0)$ and a "log" of changes that happen on the occupants of nodes s at later times. A change is simply another record that stores the position where a change occurred, the new occupant and the time of change. To achieve fast access to $s(t)$ we do not allow the "log" to get too large. Every $O(B)$ changes (in practice when the log fills one or two pages) we store a new, current copy of s. If we consider the history of node s independently, we can have an auxiliary array with records (*time, pointer*) that point to the various copies of node s. Locating the appropriate node $s(t)$ takes $O(log_B m)$ time (first find the record in the auxiliary array with the largest timestamp that is less or equal to t and then we access the appropriate copy of s and probably a (constant) number of "log" pages). The space remains $O(n+m)$ since every new node copy is amortized over the $O(B)$ changes in the "log"'.

While this solution works nicely for the history of a given B-tree node, it would lead to $O(log_B n \times log_B m)$ search I/O cost (since finding the appropriate version of a child node, when searching the B-tree, requires $O(log_B m)$ search in the child node's history). Instead of using the auxiliary array to index the copies of node s we post such entries as changes in the history of the parent node p. Assume that node s is pointed by the record on position l in node p. When a new copy of node s is created, a new record is added on the "log" of p that has the same position l, but a pointer to the new copy of s and the current time. Since new node copies are added after $O(B)$ changes, the overall space remains $O(n+m)$. The query time is reduced to $O(log_B(n+m))$ since performing a binary search on list $L(t)$ is equivalent to searching a path of $B(t)$; locating the root of $B(t)$ takes $O(log_B m)$ (searching the history of the B-tree root node) while all other nodes of $B(t)$ are found in $O(log_B n)$ using the appropriate parent to child pointers. ∎

The following theorem follows from the previous lemmas:

Theorem 2 *Given N objects and a time limit T, an one dimensional MOR1 query can be answered in time $log_B(n+m)$ using space $O(n+m)$, where $m = \frac{M}{B}$ and M is the number of crossings of objects in the time limit T.*

To solve the problem of answering queries within a time interval T into the future, we stagger the construction of our data structure. Thus, at time t_0 we construct a data structure that will answer queries in the time interval $[t_0, t_0 + 2T]$, and at time $t_0 + iT$ we construct a data structure that will answer queries in the time interval $[t_0 + (i+1)T, t_0 + (i+2)T]$.

Our approach works for any value of T. If the time limit is set too large however, all pairs of objects may cross, in which case the size of the data structure will be quadratic. It is therefore important to set the time limit appropriately so that only approximately a linear number of crossings occur. However, in many practical applications many objects move with approximately equal speeds (one example is cars on a freeway) and therefore do not cross very often.

4.4 Using Point Access Methods

Partition trees are not very useful in practice, because the query time is $O(n^{\frac{1}{2}+\epsilon} + k)$ and the hidden constant factor becomes large if we choose a small ϵ. In this section we present two different approaches that are designed to improve the average query time.

There is a large number of access methods that have been proposed to index point data [16]. All these structures were designed to address *orthogonal* queries, i.e. a query expressed as a multidimensional hyperrectangle. However, most of them can be easily modified to address non-orthogonal queries like simplex queries.

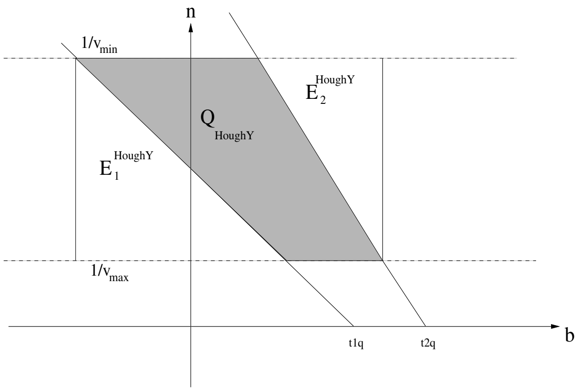

Figure 5: Query on the dual Hough-Y plane.

Goldstein et al. [17] presented an algorithm to answer simplex range queries using R-trees. The idea is to change the search procedure of the tree. In particular they gave efficient methods to test whether a linear constraint query region and a hyper-rectangle overlap. As mentioned in [17] this method is not only applicable to the R-tree family, but to other access methods as well. We can use this approach to answer the one dimensional MOR query in the dual Hough-X space.

We can improve on this approach by using a characteristic of the Hough-Y dual transformation. In this case, we assume that objects have a minimum and maximum speed, v_{min} and v_{max} respectively. The v_{max} constraint is natural in moving object databases that track physical objects. On the other hand, the v_{min} constraint comes from the fact that the Hough-Y transformation cannot represent static objects. For these objects, we use the Hough-X transformation, as it is explained above. In general, the b coordinate can be computed at different horizontal $(y = y_r)$ lines. The query region is described by the intersection of two half-plane queries (Figure 5). The first line intersects the line $n = \frac{1}{v_{max}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and the line $n = \frac{1}{v_{min}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, \frac{1}{v_{min}})$. Similarly the other line that defines the query intersects the horizontal lines at $(t_{2q} - \frac{y_{1q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and $(t_{2q} - \frac{y_{1q}-y_r}{v_{min}}, \frac{1}{v_{min}})$.

Since access methods are more efficient for rectangle queries, suppose that we approximate the simplex query with a rectangular one. In Figure 5 the query approximation rectangle will be $[(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, t_{2q} - \frac{y_{1q}-y_r}{v_{max}}), (\frac{1}{v_{max}}, \frac{1}{v_{min}})]$. Note that the query area is enlarged by the area $E = E^{HoughY} = E_1^{HoughY} + E_2^{HoughY}$ which is computed as:

$$E^{HoughY} = \frac{1}{2}(\frac{v_{max} - v_{min}}{v_{min} \times v_{max}})^2(\mid y_{2q} - y_r \mid + \mid y_{1q} - y_r \mid) \quad (1)$$

The objective is to minimize E, since it represents a measure of the extra I/O's that an access method will have to perform for solving an one dimensional MOR query. E is based on both y_r (i.e. where the b coordinate is computed) and the query interval (y_{1q}, y_{2q}) which is unknown. Hence, we propose to keep c indices (where c is a small constant) at equidistant y_r's. All c indices contain the same information about the objects, but use different y_r's. The i-th index stores the b coordinates of the data points using $y_i = \frac{y_{max}}{c} \times i, i = 0, \ldots, c-1$ (see Figure 6). Conceptually, y_i serves as an "observation" element, and its corresponding index stores the data as observed from position y_i. We call the area between subsequent "observation" elements a *sub-terrain*. A given one dimensional MOR query will be forwarded to, and answered exactly by, the index that minimizes E.

To process a general query interval $[y_{1q}, y_{2q}]$ we consider two cases depending on whether the query interval

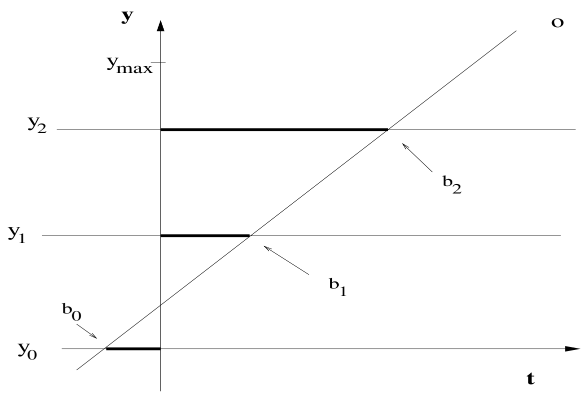

Figure 6: Coordinate b as seen from different 'observation' points

covers a sub-terrain:

(i) $y_{2q} - y_{1q} \leq \frac{y_{max}}{c}$: then it can be easily shown that area E is bounded by:

$$E \leq \frac{1}{2}(\frac{v_{max} - v_{min}}{v_{min} \times v_{max}})^2(\frac{y_{max}}{c}) \tag{2}$$

The query is processed at the index that minimizes $|y_{2q} - y_r| + |y_{1q} - y_r|$.

(ii) $y_{2q} - y_{1q} > \frac{y_{max}}{c}$: the query interval contains one or more sub-terrains, which implies that if a query is executed at a single observation index, area E becomes large. To bound E we index each sub-terrain, too. Each of the c sub-terrain indices records the time interval when a moving object was in the sub-terrain. Then the query is decomposed into a collection of smaller sub-queries: one sub-query per sub-terrain fully contained by the original query interval, and one sub-query for each of the original query's endpoints. The sub-queries at the endpoints fall to case (i) above, thus they can be answered with bounded E using an appropriate "observation" index. To index the intervals in each sub-terrain we could use an external memory Interval tree [6] which will answer a sub-terrain query optimally (i.e. $E = 0$). As a result, the original query can be answered with bounded E. However, Interval trees will increase the space consumption of the indexing method.

The same approach can be used for the Hough-X transformation, where instead of different "observation" points we have different "observation" times. That is, we can compute the intercept a using different vertical lines $t = t_i, i = 0, \ldots, c-1$. For each different intercept we create a different index. Then, given a query, we have to choose one of the indices to answer the query (the one that is constructed for the "observation" time closest to the query time.) Note however that, if the query time(s) is far from the "observation" time of an index, then the index will not be very efficient, since the query in the Hough-X will not be aligned with the rectangles representing the index and data pages of this index. So, one problem with this approach comes from the fact that the time in general and the query time in particular, are always increasing. Therefore, an index that is efficient now will become inefficient later. That is not true for the Hough-Y case, since the terrain and the query domain do not change with time (or they change very slowly).

5 Indexing in two dimensions

For the two-dimensional problem, trajectories of the moving objects are lines in a three dimensional space (see Figure 7). We address the 2-dimensional problem by decomposing the motion of the object into two independent motions, one in the (t, x) plane and one in the (t, y) plane. Each motion is indexed separately. Next we present the procedure used in order to build the index, and then the algorithm for answering the 2-d query.

5.1 Building the index

We begin by decomposing the motion in (x, y, t) space into two motions on the (t, x) and (t, y) plane. Furthermore, on each projection, we partition the objects according to their velocity. Objects with small velocity magnitude are stored using the Hough-X dual transform, while the rest of them are stored using the Hough-Y transform, i.e into distinct index structures.

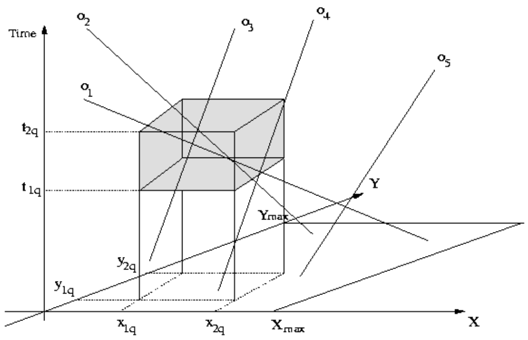

Figure 7: Trajectories and query in (x, y, t) space.

The reason for using different transforms is that motions with small velocities in the Hough-Y approach are mapped into dual points (b, n) having large n coordinates ($n = \frac{1}{v}$). Thus, since few objects have small velocities, by storing the Hough-Y dual points in an index structure such an R*-tree, MBRs with large extents are introduced, and the index performance is severely affected. On the other hand, by using a Hough-X index for the small velocities' partition, we eliminate this effect, since the Hough-X dual transform maps an object's motion to the (v, a) dual point. To partition the objects into slow and fast, we use a threshold VT.

When a dual point is stored in the index responsible for the object's motion in one of the planes, i.e. (t, x) or (t, y), information about the motion in the other plane is also included. Thus, the leaves in both indices for the Hough-Y partition store the record (n_x, b_x, n_y, b_y). Similarly, for the Hough-X partition in both projections we keep the record (v_x, a_x, v_y, a_y). In this way, the query can be answered by one of the indices; either the one responsible for the (t, x) or the (t, y) projection.

On a given projection, the dual points (i.e. (n, b) and (v, a)) are indexed using R*-trees [8]. The R*-tree has been modified in order to store points at the leaf level, and not degenerated rectangles. Therefore, we can afford storing extra information about the other projection. An outline of the procedure for building the index follows:

1. Decompose the 2-d motion into two 1-d motions on the (t, x) and (t, y) planes.
2. For each projection, build the corresponding index structure
    - Partition the objects according to their velocity:

(a) Objects with $|v| < VT$ are stored using the Hough-X dual transform, while objects with $|v| \geq VT$ are stored using the Hough-Y dual transform.
(b) Motion information about the other projection is also included in each point.

In order to choose one of the two projections and answer the simplex query, the technique described next is used.

5.2 Answering the query

The two dimensional MOR query is mapped to a simplex query in the dual space. The simplex query is the intersection of four 3-d hyper-planes and the projections of the query on the (t, x) and (t, y) planes are wedges, as in the 1-dimensional case.

The 2-d query is decomposed into two 1-d queries, one for each projection, and it is answered exactly. Furthermore, on a given projection, the simplex query is processed in both partitions, i.e. Hough-Y and Hough-X.

On the Hough-Y plane the query region is given by the intersection of two half-plane queries, as shown in Figure 5. Consider the parallel lines $n = \frac{1}{v_{min}}$ and $n = \frac{1}{v_{max}}$. Note that a minimum value for v_{min} is VT. As illustrated in section 4, if the simplex query was answered approximately, the query area would be enlarged by $E^{HoughY} = E_1^{HoughY} + E_2^{HoughY}$ (the triangular areas in Figure 5). Also, let the actual area of the simplex query be Q^{HoughY}. Similarly, on the dual Hough-X plane (Figure 3), let Q^{HoughX} be the actual area of the query, and E^{HoughX} be the enlargement. The algorithm chooses the projection which minimizes the following criterion κ:

$$\kappa = \frac{E^{HoughY}}{Q^{HoughY}} + \frac{E^{HoughX}}{Q^{HoughX}} \tag{3}$$

The intuition for this heuristic [29] is that simplex queries in the dual space are not aligned with the MBRs of the underlying index (see Figure 8). Therefore, we would like to ask the query in the projection, where the query is as much aligned with the MBRs as possible. The empty space, as used in the aforementioned criterion definition, gives an indication of that.

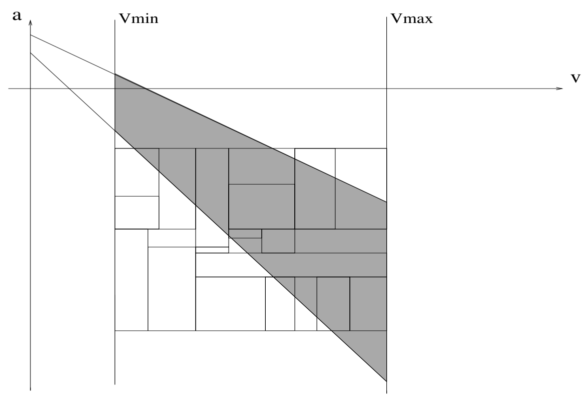

Figure 8: Simplex query in dual space, not aligned with MBRs of underlying index.

Since the whole motion information is kept in the indices, it can be used to filter out objects that do not satisfy the query. An outline of the algorithm for answering the exact 2-d query is presented next:

1. Decompose the query into two 1-d queries, for the (t, x) and (t, y) projection.
2. Get the dual query for each projection (i.e. the simplex query).
3. Calculate the criterion κ for each projection, and choose the one (say p) that minimizes it.
4. Answer the query by searching the Hough-X and Hough-Y partition, using projection p.
5. Put an object in the result set, only if it satisfies the query. Use the whole motion information to do the filtering "on the fly".

6 Performance Evaluation

In this section we present experimental results for objects moving in one and two dimensional spaces. We use the simpler, one dimensional experiments to reveal the behavior of the Hough-X and Hough-Y approaches (section 4) since they are components of the proposed two-dimensional solution (section 5). For the two-dimensional space we compare our approach with the TPR-tree [35, 36]. We chose the TPR-tree as a very efficient representative of the non-dual transformation methods (section 3).

6.1 One dimensional case

Experimental Setting. We present results for the one dimensional MOR query, comparing the Hough-Y approach (multiple indices), the Hough-X method and a traditional R-tree based approach which stores trajectories as line segments.

First we describe the way experimental data is generated. At time $t = 0$ we generated the initial locations of N mobile objects uniformly distributed on the (line segment) terrain $[0, 1000]$. We varied N from $100K$ to $500K$. The speeds were generated uniformly from $v_{min} = 0.16$ to $v_{max} = 1.66$ and the direction randomly positive or negative. (Note that 0.16miles/min is equal to 10 miles/hour and 1.66 miles/min is equal to 100 miles/hour.) Then the objects start moving. When an object reaches a border, it simply changes its direction. We generate 10 different time instants that represent the times when queries are executed. At each time instant we execute 200 random queries, where the length of the y-range is chosen uniformly between 0 and $YQMAX$ and the length of the time range between 0 and WT. We actually generated two sets of queries workloads: one with fixed $YQMAX = 10$ and WT varying from 10 up to 100, and one with fixed $WT = 10$ and $YQMAX$ varying again from 10 up to 100. In both sets, the query workload has average selectivity that spans from 0.5% up to 3.5%. We run this scenario using a particular access method for 2000 time instants.

We implemented the traditional R-tree approach using an R*-tree [8] with page size 4K. To represent a line segment we used four 4-byte numbers (the two end points) and one more number as a pointer to the real object, resulting in a page capacity of $B = 204$ records. For the Hough-Y and Hough-X methods, we used two-dimensional R*-trees to index the dual points. These R*-trees were appropriately modified to index points instead of rectangles. We used R-trees over the B+-trees proposed in [24] since we got much better query performance. So, we show only the results for the R-trees. The page capacity was $B = 341$ records, since we need two 4-byte numbers to represent the points plus one more number as a pointer. We did not implement the Interval trees, since the cost of creating, storing and updating these structures is high and are needed only for very large queries which are not typical.

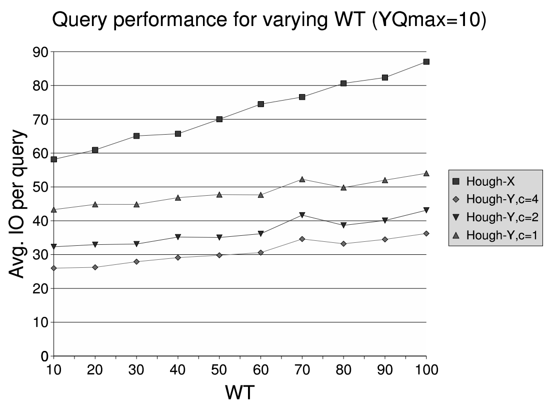

Figure 9: 1D-case: Query performance for varying WT

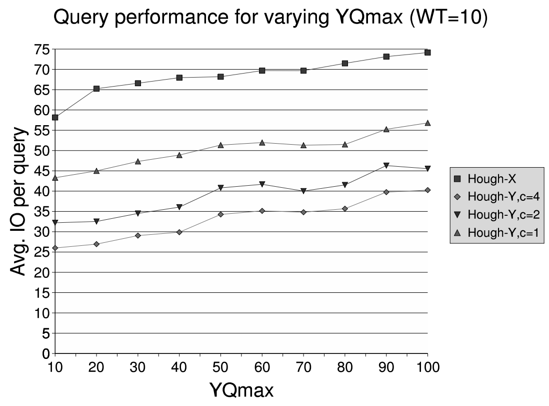

Figure 10: 1D-case: Query performance for varying YQmax

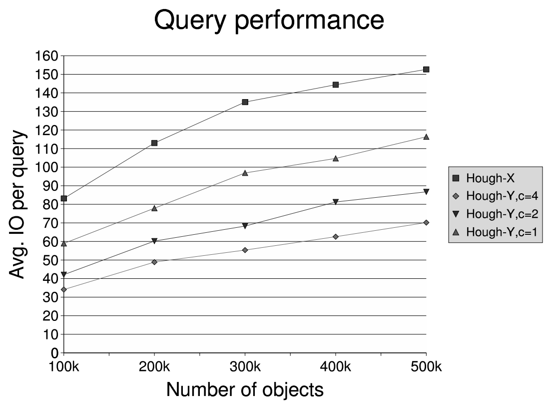

Figure 11: 1D-case: Query performance for varying number of objects

We consider a simple buffering scheme for the results we present here. For each tree we buffer the path from the root to a leaf node, thus the buffer size is only 3 or 4 pages. For the queries we always clear the buffer pool before we run a query. An update is performed when the motion information of an object changes.

Performance Results. Figure 9 presents the results for the average number of I/O's per query for queries with varying WT, while Figure 10 depicts results for queries with varying $YQMAX$. These experiments where run for $100K$ objects. Figure 11 shows how the query performance scales-up as the number of moving objects increases. For this set of experiments we set $WT = 80$ and $YQMAX = 10$, yielding an average selectivity close to 2%. In all these figures the results for the traditional R-tree storing line segments, are not depicted since, as anticipated, this method exhibits excessively high overhead (over 400 page accesses). For the Hough-Y method we used c=1, 2 and 4 R*-trees, and we observe that it outperforms the Hough-X query performance even with $c = 1$.

Figures 12, 13 plot the space consumption and the average number of I/O's per update respectively, as a function of the number of moving objects. The space of all methods is linear to the number of objects. The space consumption of the Hough-X and Hough-Y ($c = 1$) are almost identical, which is expected since in both methods objects are stored only once. The method that stores line segments (shown as "Trajectories"

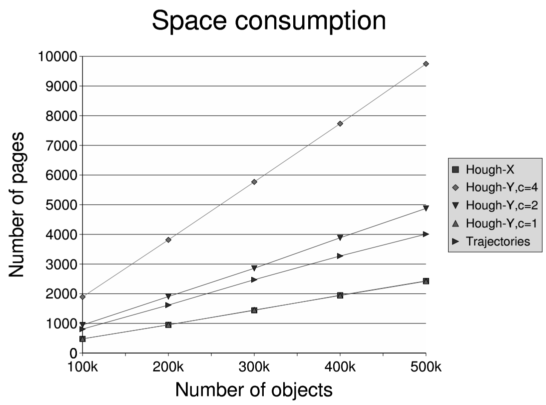

Figure 12: 1D-case: Space consumption for varying number of objects

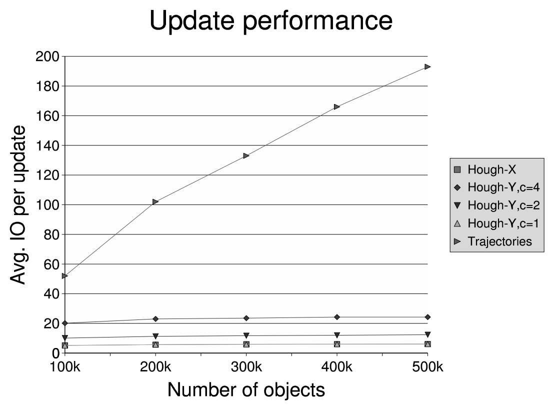

Figure 13: 1D-case: Update performance for varying number of objects

in the legend) uses somewhat more space than Hough-X and Hough-Y ($c = 1$), even though it also stores objects only once. However, the clustering of long segments is not ideal, forcing the R-tree to use more space. The Hough-Y approach with $c = 2, 4$, uses more space due the use of c observation indices. Regarding update processing, the line segments method exhibits the worst update performance that increases drastically as the number of objects increase. Most of this update cost comes from deletions where many tree paths are typically visited. The update performance of the Hough-X and the Hough-Y approach remain virtually constant while varying the number of mobile objects. Again, Hough-X and Hough-Y ($c = 1$) have almost identical update processing. In actual values, the update of Hough-X and Hough-Y ($c = 1$) is slightly increasing from 5.2 I/Os (100K objects) to around 6.1 I/Os (500K objects) but this is not seen in the figure due to the large update I/O of the line segments method. By looking in Figures 9 through 13 we see the clear tradeoff between c and query/update performance for the Hough-Y method.

6.2 Two dimensional case

Experimental Setting. For the 2-dimensional MOR query we generated a variety of datasets using the TPR-tree's generator [35] as well as our own generator.

The datasets created with the TPR generator use parameters suggested in [35]. That is, we assume objects moving on a finite terrain having size 1000 x 1000 km. The terrain contains a fully connected graph, whose edges are the routes objects can move along. Each dataset is distinguished by the number of vertices, or destinations ND (ND was set to 40 or 160). The objects are initially positioned on the routes in a random fashion. They are assigned with equally probability to one of three possible groups having maximum velocity of 0.75 (slow), 1.5 (medium) and 3km/min (fast). Within each group, objects are assigned uniform velocities between 0 and the group's maximum velocity. Objects achieve this velocity by initially accelerating (during the first 1/6th of the route), then they maintain this speed (for the next 2/3rds of the route), and finally they decelerate to 0km/min (during the last 1/6th of the route). We also generated a dataset in which objects can move randomly on the terrain without destinations (this is termed as UNI in [35]).

Each simulation scenario runs for 600 time instants, where each instant corresponds to one minute [35] (i.e., the simulation corresponds to 10 hours). Unless otherwise indicated each dataset involves $100K$ objects. An update in this environment corresponds to a deletion followed by an insertion. Updates are generated so that the average time interval between two updates is fixed to a parameter UI. Queries consist of time-slice and window queries, and are issued within a time window W from the current time. For these workloads we used $UI = 60$ and $W = 40$. These parameters are used by the TPR-tree to compute its fixed horizon H ($H = UI + W$). Four queries are issued every time instant, intermixed with around one million updates in total. Note that the total number of insertions is slightly higher than the number of deletions, since we need first to insert the 100K objects to the index. For example, the ND60 dataset had 1.07M insertions and 0.97M deletions. The other datasets had similar insertion/deletion mixture.

Queries are randomly selected with the spatial predicate covering on average 0.25% of the spatial universe, while the temporal predicate has an average length of 10 instants.

The datasets generated using our own generator assume a network of routes which intersect in "cities" (similar to the destinations of the TPR generator) and form a fully connected graph (a network of "freeways"). The terrain is again 1000 x 1000 km. Objects are randomly positioned on the routes. One difference with the TPR generator is that velocity magnitudes follow either Uniform or Gaussian distribution. In the uniform case velocities are chosen from $[0.16, 1.83]$ while in the Gaussian the mean is 1.16 and the standard deviation is 0.5. The simulation scenario runs also for 600 time instants and involves $100K$ objects. At each time instant 1% of the objects update their motion information instantly (i.e., there is no acceleration or deceleration). The simulation creates an average update interval $UI = 100$ while the query window W was 130 (therefore $H = 230$). These parameters were then input to the TPR-tree. Four queries are issued every time instant, as well. In these datasets the spatial predicate is on average 1% of the spatial universe, while the temporal predicate is 30 instants long.

The performance of the TPR-tree is best for queries within the prespecified horizon. Thus we first generated workloads with queries posted within H. In some applications however, the user may not be able to accurately predict the horizon beforehand. To examine how the behavior of the TPR-tree deteriorates for queries outside the predefined horizon, we also generated workloads where the query temporal attributes (t_{1q} and t_{2q}) are gradually shifted in increments of $1H$ up to $5H$.

There is one more reason for experimenting with "out-of-horizon" queries. This behavior is similar to the TPR-tree query performance for time periods between distant updates. The TPR-tree partially reorganizes its structure during each update (this is the "update-time" setting in [35]). Performance is optimized for queries issued within H from the last update. Recall that the computation of H uses the *average* update interval UI ($H = UI + W$). Hence, there maybe cases where the next update is much further than UI and queries can exceed the prespecified horizon. When updates are infrequent, the size of the time-parameterized MBRs increases over time, which deteriorates query performance.

We also experimented with a TPR-tree that uses automatic horizon estimation [36]. Here a heuristic for dynamically maintaining the time horizon is introduced and involves tracking the operations in the index. The parameter UI is approximated by $(\Delta t/B)l$, where l is the current number of leaf entries, B is the number of entries per leaf page, and Δt is the time it took to receive the last B entries. The parameter W is approximated as a function of UI: $W = \alpha UI$, where $0 < \alpha < 1$ (typically $\alpha = 0.5$).

We implemented the DUAL approach as described in section 5. For the VT threshold we used 0.16. Different values of VT do not change the performance much, so we kept $VT = 0.16$ for all experiments. For all methods the page size was set to $4K$ and a buffer pool of 50 pages was used while the leaf capacity was 204.

Performance Results. Our experimental results are shown in Figures 14- 27; here TPR-fixed denotes the TPR-tree using a fixed horizon, TPR-auto stands for TPR-tree with automatic horizon estimation while DUAL corresponds to the method described in section 5.

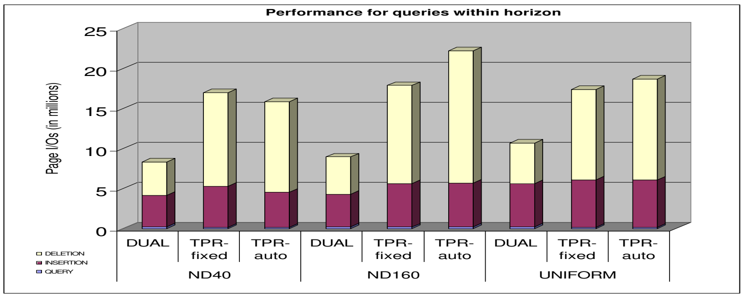

Figure 14: ND/UNI datasets: queries within the horizon, overall I/O comparison

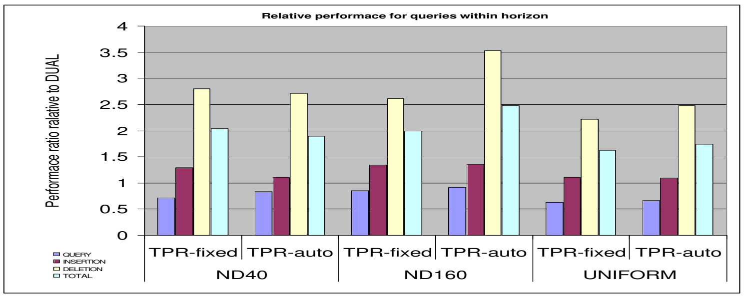

Figure 15: ND/UNI datasets, queries within the horizon, ratio of performance relative to DUAL

Figure 14 presents the overall page I/O for updates (insertions and deletions) and queries (within the horizon) for three datasets, namely ND40 (i.e., ND = 40), ND160 and UNI, with 100K objects. The purpose of this figure is to depict the importance of updates in this dynamic environment. Note that each object issued an average of 10 updates during the simulation [35]; when projected to a practical scenario, this is a rather low update rate. The number of queries is about 2.4K, which corresponds to a rate of four queries per minute. Nevertheless, it is apparent that updating consumes the largest processing part among all indices. Since the number of insertions is very close to the number of deletions, it is further observed that deletions are much more expensive for the TPR trees than insertions. This is to be expected since the TPR-tree uses deletions for index reorganizations.

Figure 15 shows the ratios of the query, insertion and deletion operations of the TPR-trees relative to the DUAL method. Clearly, both TPR-trees have faster query time than the DUAL method for queries within the horizon (and for all datasets shown). They use however, considerably more update time, especially for deletions (around 2.5 times more). The TPR-auto uses slightly more query and update processing than the TPR-fixed given the horizon estimation is performs. In the figure we also indicate the "total" ratio which corresponds to the overall I/O of each TPR-tree divided by the overall I/O of the DUAL method. For the above experiments, Figures 16 and 17 depict the average page I/O per query and update respectively.

Figure 18 shows how the methods scale-up as the average number of moving objects increases from 100K to 500K. The ND160 dataset was used for these experiments and queries inside the horizon are depicted.

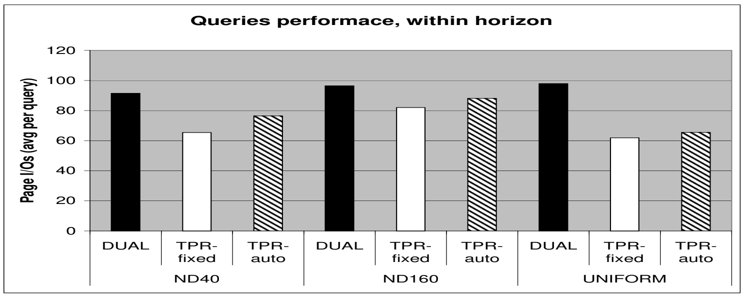

Figure 16: ND/UNI datasets, queries within the horizon, average I/O per query

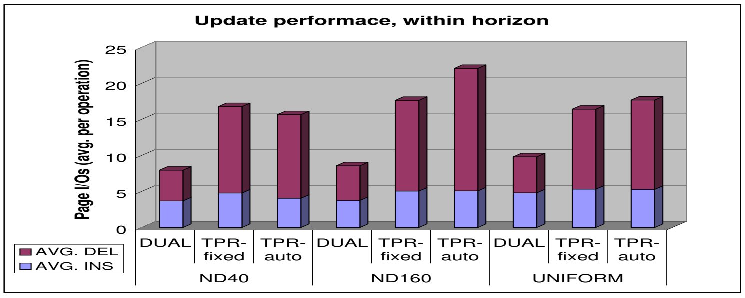

Figure 17: ND/UNI datasets, queries within the horizon, average I/O per update

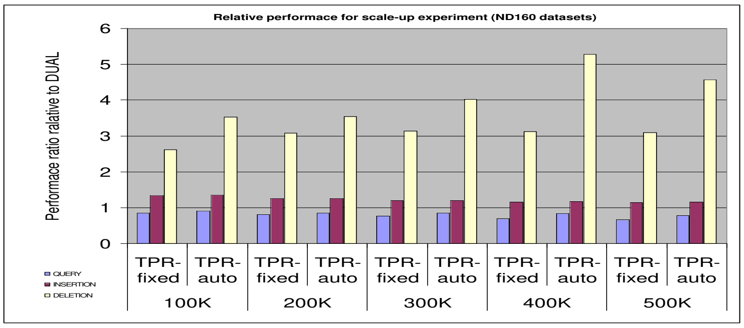

Figure 18: Varying the number of moving objects

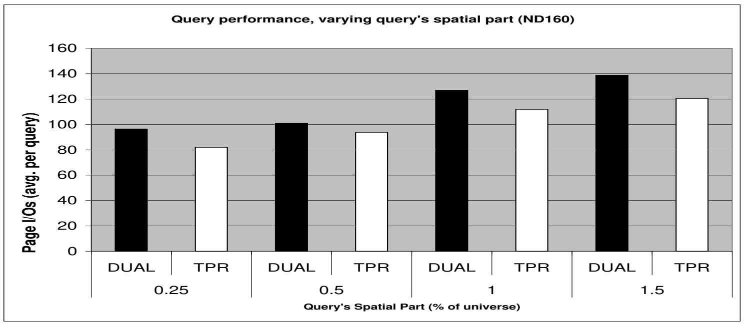

Figure 19: Varying the size of the spatial predicate

All methods seem to scale up graciously (the relative ratios remain similar). Again, the TPR-tree query time is around 75% the query time of the DUAL method, but its update time is much worse (above 2.5 times for deletions).

To test how the methods are affected by the query size, we run experiments using the ND160 dataset and varying the query spatial predicate from 0.25 to 1.5% of the spatial universe. Queries were again posted within the predefined horizon and the temporal predicate was maintained to 10 instants. Figure 19 depicts the results for the DUAL and TPR-fixed methods. In both methods the query time increases gradually (which is to be expected as the answer size increases since more objects will satisfy the query).

Next, Figure 20 shows the performance (again as ratios relative to DUAL) for queries outside the horizon on the ND160 dataset (we got similar results for ND40 and UNI datasets). The queries were placed from 1H until 5H outside the horizon H. The update times are not shown as they are similar with Figure 15. As expected, queries in the TPR-trees outside the horizon deteriorate as the query moves further from the horizon. Even for queries within 1H outside the horizon, the TPR tree uses about twice the query time of the DUAL method. The query time of the TPR-auto deteriorates faster than the TPR-fixed since the estimation quality reduces the further away from the fixed horizon.

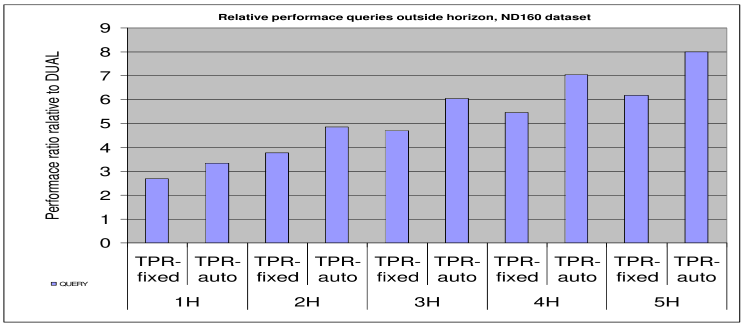

Figure 20: ND160 dataset, queries outside the horizon, ratio of performance relative to DUAL

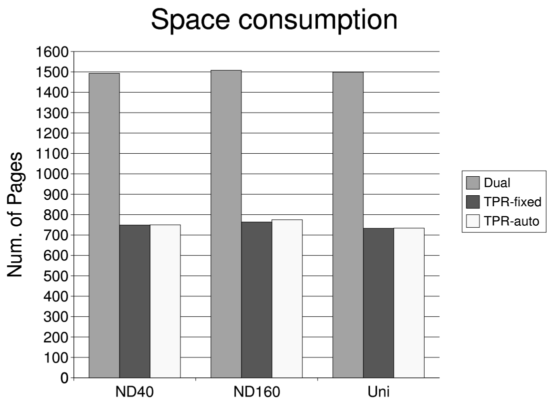

Figure 21: Space consumption for ND/UNI

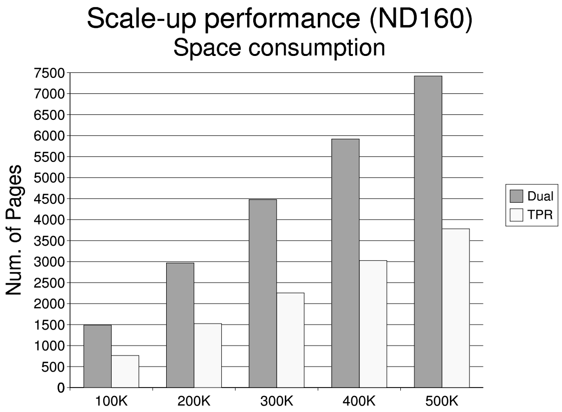

Figure 22: Scale-up experiment: space consumption

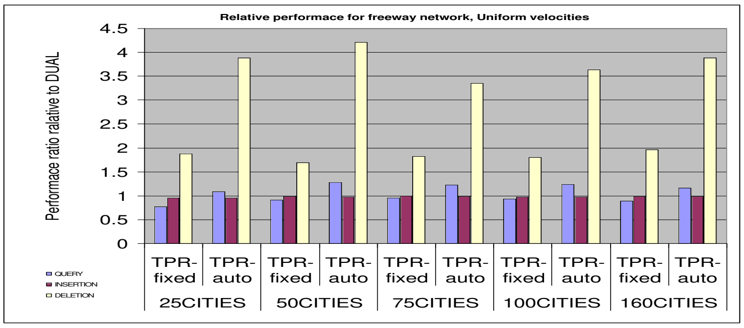

Figure 23: Freeways network, unif. velocities, queries within the horizon, ratio of performance relative to DUAL

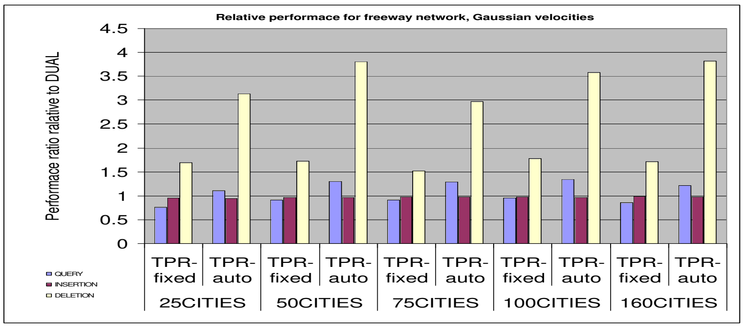

Figure 24: Freeways network, Gaussian velocities, queries within the horizon, ratio of performance relative to DUAL

Figure 21 shows the space consumption for the ND and UNI datasets. Clearly the DUAL method uses double the space of the TPR trees, since each point is stored in two indices—one for each dimension. Figure 22 depicts how the space consumption scales up as the number of objects increases for the ND160 dataset. As expected, the space consumption of all methods increases linearly with the number of moving objects.

The next figures present the results for the "freeway" datasets created with our own generator. In general, we get very similar results as with the TPR-generator datasets. Figure 23 depicts the performance of the TPR-trees as ratio relative to DUAL for uniformly chosen velocities, with varying number of cities (destinations) and queries within the horizon. The TPR-tree has again better query performance, but it is closer to DUAL than before. Interestingly, the TPR-auto has slightly worse query time than DUAL. The DUAL method has again much faster update processing times. The corresponding results for Gaussian velocity distributions appear in Figures 24 and 25.

Finally, we performed an experiment where the scenario run for 4800 time instants. We measure the performance of the index every 20 time instants and we compute the average query and update performance until the current time. In Figure 26 we plot the query performance, for ND40 and queries inside the horizon. The

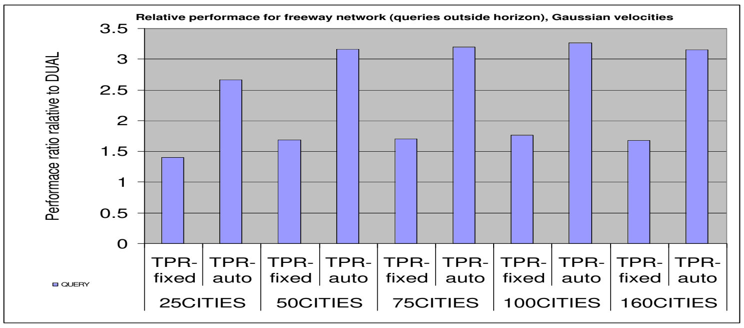

Figure 25: Freeways network, Gaussian velocities, queries outside the horizon by 1H, ratio of performance relative to DUAL

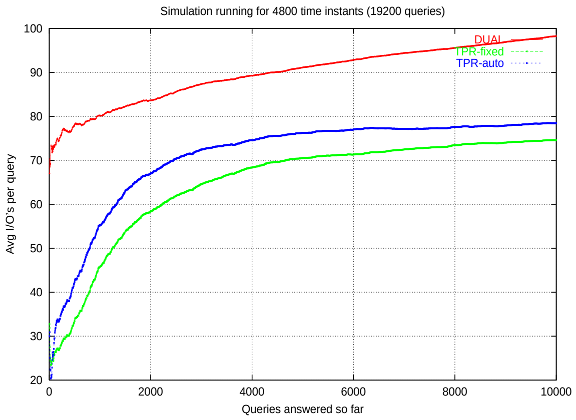

Figure 26: Query performance for increasing current time

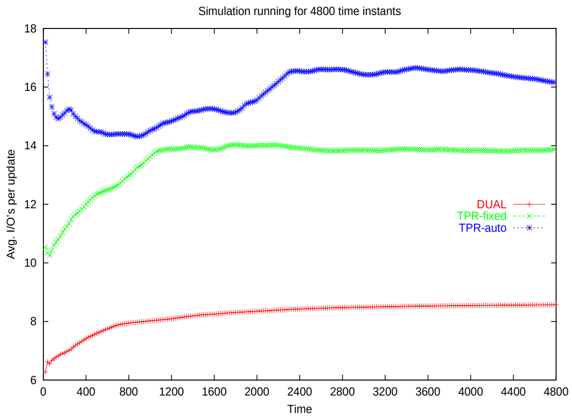

Figure 27: Update performance for increasing current time

query performance of the DUAL approach deteriorates with time since most of the objects are moving. On the other hand, the TPR-trees deteriote fast at the beginning of the simulation but at some point they stabilize, around an average of 80 I/O's per query. Note that this is the average until the current time. Therefore, the query performance is much worse than the performance at the initial time instants, but it stabilizes after some time instant. This figure suggests that the DUAL index must be rebuilt at periodic time intervals, in order to keep the query performance low. Figure 27 depicts the update performance per update for the same experiment. In that case, all indices stabilize after some initial time period. The update performance of the DUAL is about 1.6 times better than the TPR-fixed and 1.85 times better than the TPR-auto.

Discussion. The 2D experiments reveal that for queries posted within the predefined horizon, the TPR-fixed tree performs better than the DUAL method (on average by 20% for datasets generated using the TPR generator and around 15% for the "freeway" datasets). On the other hand, when the queries are posted outside the horizon, the TPR-tree performance is affected dramatically. Even for queries that are within 1H outside the predefined horizon, the TPR-fixed performs on average 2.5 times worse for the TPR datasets and 1.75 times worse for the "freeway" datasets. That is, the performance of the TPR-tree is very closely coupled to the predefined horizon. While for some applications such predefined horizon definition is possible, for others it may not. In contrast, the DUAL method does not depend upon knowing the characteristics of the anticipated workload (i.e. the parameter UI), neither it assumes any query window W. Actually, the DUAL method improves as queries move further into the future because the query selectivity drops. Moreover, the TPR-auto where the horizon is automatically selected based on the previous history of updates, did not seem to perform as good as the TPR-fixed; in the "freeway" datasets it had worse query performance than the DUAL, even for within the horizon queries.

We feel that an even more important comparison criterion for a moving objects environment is the update performance. Given the large number of objects, updates occur at a much higher rate than queries. Thus it is crucial for the index method to have fast update processing in order to maintain a realistic view of the observed environment. The dual transformation approach always exhibits significantly faster update performance. While the I/O cost for insertion operations is typically equivalent for both methods (with the TPR-fixed tree having insertion cost varying from 3% better up to 35% worse than our method), the I/O cost for deletion operations is always much higher for the TPR-tree (between 2.5 and 3 times larger for the TPR datasets and between 1.5 and 2 times larger for the "freeway" datasets). This is because the TPR-tree recalculates and re-organizes the time-parameterized MBRs in a bottom-up fashion, whenever an update is issued. These reorganizations (i.e., making the time parameterized MBRs tighter) are crucial for the TPR-tree to maintain its good query performance within the horizon. For periods with larger than average update intervals, the TPR-tree query behavior deteriorates (as when queries are out of the predefined horizon).

On the other hand, the DUAL method requires larger space, about twice what the TPR-tree uses. However, given the decreasing costs of disk space, it seems that trading space for update performance is rather useful.

7 Conclusions

We presented external memory techniques for indexing moving objects, in order to efficiently answer range queries about their location in the future. By employing dual transformations, we illustrated efficient indexing schemes for the one-dimensional (moving on a line), as well as the two-dimensional case. We further performed an extensive comparison of our approach with the TPR-tree, an efficient index that does not use duality transformation but instead time-parameterized nodes and a predefined query horizon. While our approach uses comparable query time processing (more for queries within the horizon but less for queries outside the horizon), it has much less update cost. Updating is an important consideration given the highly dynamic environment of moving objects. Moreover, the duality approach does not require the specification of a predefined horizon.

An interesting future direction of research is joins among relations of mobile objects. Furthermore, it would be worth considering the problem in the context of uncertainty in the position and velocity of the mobile objects. Finally, techniques for answering aggregate complex queries, such as predicting and reporting the areas with high density of mobile objects, are also of high practical interest.

Acknowledgement. We would like to thank Simonas Šaltenis for providing the source code for the TPR-tree and many helpful discussions.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. *In Proc. of the 19th ACM Symp. on Principles of Database Systems (PODS)*, pages 175–186, 2000.

[2] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. In *Proc. of the 17th ACM Symposium on Principles of Database Systems*, pages 169–178, 1998.

[3] P.K. Agarwal and S. Har-Peled. Maintaining Approximate Exten Measures of Moving Points. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001.

[4] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] L. Arge, V. Samoladas, and J.S. Vitter. On Two-Dimensinal Indexability and Optimal Range Search Indexing. In *Proc. of the 18th ACM Symp. on Principles of Database Systems (PODS)*, pages 346–357, June 1999.

[6] L. Arge and J.S. Vitter. Optimal Dynamic Interval Management in External Memory. In *Proc. 37th Annual Symp. on Foundations of Comp. Science*, pages 560–569, 1996.

[7] J. Basch, L. Guibas, and J. Hershberger. Data Structures for Mobile Data. In *Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, 1997.

[8] 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 the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, May 1998.

[9] B. Chazelle and B. Rosenberg. Lower bounds on the complexity of simplex range reporting on a pointer machine. In *Proc. 19th Intern. Colloquium on Automata, Languages and Programming*, volume 623 of *LNCS*, pages 439–449, Berlin, 1992. Springer-Verlag.

[10] Y.-J. Choi and C.-W. Chung. Selectivity Estimation for Spatio-Temporal Queries to Moving Objects. In *Proc. of the ACM SIGMOD*, Madison, Wisconsin, June 2002.

[11] J. Chomicki and P. Revesz. A Geometric Framework for Specifying Spatiotemporal Objects. *In Proc. 6th International Workshop on Time Representation and Reasoning*, 1999.

[12] H. D. Chon, D. Agrawal, and A. El Abbadi. Query Processing for Moving Objects with Space-Time Grid Storage Model. In *Proc. of the Int. Conf. on Mobile Data Management*, Jan 2002.

[13] R. Cole. Searchinf and Storing Similar Lists. *Journal of Algorithms*, 7(2):202–220, 1986.

[14] J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences, Vol. 38, No. 1*, pages 86–124, 1989.

[15] K.M. Elbassioni, A. Elmasry, and I. Kamel. An efficient indexing scheme for multi-dimensional moving objects. *In Proc. ICDT*, pages 425–439, 2003.

[16] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, Jun 1998.

[17] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.B. Yu. Processing Queries By Linear Constraints. In *Proc. 16th ACM PODS Symposium on Principles of Database Systems*, pages 257–267, Tuscon, Arizona, 1997.

[18] O. Günther. The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases. In *Proc. 5th IEEE Inter. Conf. on Data Engineering*, Los Angeles, CA, USA, February 1989.

[19] R.H. Güting, M.H. Böhlen, M. Erwing, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems (TODS)*, 26(1):1–42, March 2000.

[20] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD*, pages 47–57, Boston, Mass, June 1984.

[21] H. V. Jagadish. On Indexing Line Segments. In *Proc. 16th. International Conference on Very Large Data Bases*, pages 614–625, Brisbane, Queensland, Australia, August 1990.

[22] D.V. Kalashnikov, S. Prabhakar, S.E. Hambrusch, and W.G. Aref. Efficient evaluation of continuous range queries on moving objects. *In Proc. DEXA*, pages 731–740, 2002.

[23] G. Kollios, D. Gunopulos, and V. Tsotras. Nearest Neighbor Queries in a Mobile Environment. *In Proc. of the Spatio-Temporal Database Management Workshop, Edinburgh, Scotland*, pages 119–134, 1999.

[24] G. Kollios, D. Gunopulos, and V. Tsotras. On Indexing Mobile Objects. *In Proc. of the 18th ACM Symp. on Principles of Database Systems (PODS)*, pages 261–272, June 1999.

[25] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic Queries over Mobile Objects. *In Proc. EDBT*, pages 269–286, 2002.

[26] J. Matousek. Efficient Partition Trees. *Discrete and Computational Geometry*, 8:432–448, 1992.

[27] H. Mokhtar, J. Su, and O.H. Ibarra. On moving object queries. *In Proc. PODS*, pages 188–198, 2002.

[28] M. H. Overmars. *The Design of Dynamic Data Strucutures*, volume 156 of *LNCS*. Springer-Verlag, Heidelberg, West Germany, 1983.

[29] D. Papadopoulos, G. Kollios, D. Gunopulos, and V.J. Tsotras. Indexing Mobile Objects on the Plane. In *Proc. 5th International Workshop on Mobility in Databases and Distributed Systems (MDDS-DEXA)*, Aix-en-Provence, France, 2002.

[30] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. *In Proceedings of VLDB, Cairo Egypt*, September 2000.

[31] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. *In Proc. SSTD*, pages 59–78, 2001.

[32] S. Prabhakar, Y. Xia, D.V. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *In IEEE Transactions on Computers, Vol. 51, No. 10*, pages 1124–1140, 2002.

[33] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. Star-tree: An efficient self-adjusting index for moving objects. In *Proc. 4th Workshop on Algorithm Engineering and Experiments*, 2002.

[34] S. Saltenis, C. Jensen, S. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *Time-Center Technical Report*, November 1999.

[35] S. Saltenis, C. Jensen, S. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. *In Proceedings of the ACM SIGMOD*, pages 331–342, May 2000.

[36] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. 18th. Inter. Conference on Data Engineering*, San Jose, CA, Feb 2002.

[37] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, June 1990.

[38] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. 13th Inter. Conf. on Very Large Data Bases*, pages 507–518, Brighton, England, September 1987.

[39] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. *In Proceedings of the 13th ICDE, Birmingham, U.K*, pages 422–432, April 1997.

[40] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proc. SSTD 2001*, pages 79–96, Redondo Beach, CA, July 2001.

[41] S. Subramanian and S. Ramaswamy. The P-range Tree: A New Data Structure for Range Searching in Secondary Memory. In *Proc. of the 6th Annual Symposium on Discrete Algorithms*, New York, NY, USA, 1995.

[42] Y. Tao and D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *Proc. of ACM SIGMOD*, Madison, Wisconsin, June 2002.

[43] Y. Tao, D. Papadias, and S. Qiongmao. Continuous Nearest Neighbor Search. In *Proc. of VLDB*, August 2002.

[44] Y. Tao, J. Sun, and D. Papadias. Selectivity Estimation for Predictive Spatio-Temporal Queries. In *Proc. 19th. Inter. Conference on Data Engineering*, Bangalore, India, March 2003.

[45] J. Tayeb, O. Olusoy, and O. Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3):185–200, 1998.

[46] O. Wolfson, S. Chamberlain, S.Dao, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. 14th IEEE Inter. Conf. on Data Engineering*, pages 588–596, Orlando, Florida, February 1998.

[47] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proc. of 11th Int. Conf. on Scientfic and Statistical Database Management*, pages 111–122, Capri, Italy, Jul 1998.

[48] H. Zhu, J. Su, and O.H. Ibarra. Trajectory queries and octagons in moving object databases. *In Proc. CIKM*, pages 413–421, 2002.