Indexing mobile objects using dual transformations

George Kollios[1,*]**, Dimitris Papadopoulos**[2]**, Dimitrios Gunopulos**[2,**]**, Vassilis J. Tsotras**[2,***]

[1] Department of Computer Science, Boston University, Boston, MA 02215, USA
(e-mail: gkollios@cs.bu.edu)
[2] Department of Computer Science & Engineering, University of California Riverside, Riverside, CA 92521, USA
(e-mail: {dimitris,dg,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 a high concentration of mobile phones is imminent. We address the problem in external memory and present dynamic solutions, both for the one-dimensional and 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

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 that 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) [20]. 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 that 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 remain 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 change in discrete steps, but it is inefficient for applications with continuously changing data [44]. 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 storing not data values but rather functions to compute these values. Motion database problems have recently attracted the interest of the research community. Sistla et al. [44] and Wolfson et al. [53,54] present the Moving Objects Spatio-Temporal (MOST)

model and a language (FTL) for querying the current and future locations of mobile objects; Güting et al. [20] propose a model that tracks and queries the history (past routes) of mobile objects based on new spatiotemporal data types. Another spatiotemporal model appears in [11]. Spatiotemporal queries about mobile objects have important applications in traffic monitoring, intelligent navigation, and mobile communication 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 a 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 spatiotemporal query is: "Report all the objects that will be inside a query region P 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 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 orders of magnitude 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. Another important issue in spatiotemporal databases is related to the protection of the privacy of mobile users. Recent directives and regulations, such as the European directive 58/2002/EC [15], specify that the location information of mobile users constitutes sensitive private information and must be protected against unauthorized use. Note that, in our setting, we assume that after an object updates its motion information, the past locations are deleted from the database. Therefore, the database keeps a given location of an object or subject for only a limited time. However, since range queries provide the location and object IDs of moving objects, the privacy of these object can be compromised if we allow someone to ask many queries for different time instants. In this paper we do not consider the above privacy issues since our methods are aimed at applications where object identification does not raise privacy concerns (e.g., military ones, where objects may be related to actual soldiers or vehicles in the field). If privacy is important for a specific application, additional steps are required to guarantee privacy protection of the mobile users (e.g., anonymity). Another approach is to allow only aggregate queries (for example COUNT, SUM, and AVG queries) that do not reveal object IDs [22,23,32,47,51].

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 [27, 54], where the initial location of the moving objects along with
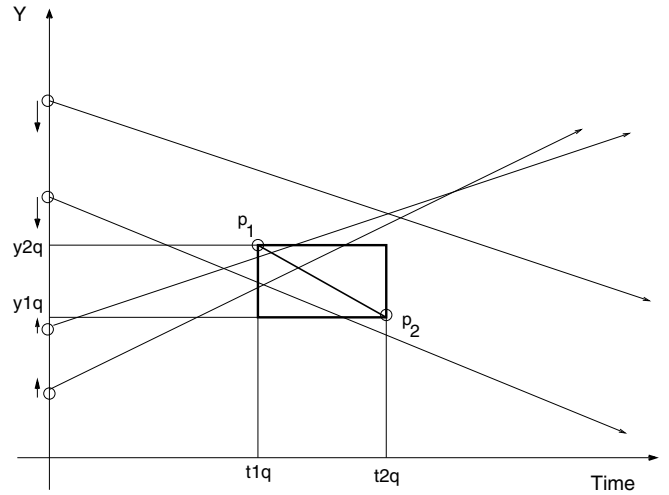

Fig. 1. Trajectories and query in (t, y) plane

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 tradeoff 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 related work, while the one-dimensional case is addressed in Sect. 4. The technique for indexing objects that move freely in two dimensions is described in Sect. 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 Sect. 6. Finally, Sect. 7 concludes the paper.

2 Preliminaries

In this section we formally define the problem of indexing two-dimensional moving objects. Then, 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 [54,40,27], 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 $P(t) = [x(t), y(t)] = [x_0 + v_x(t - t_0), y_0 +$
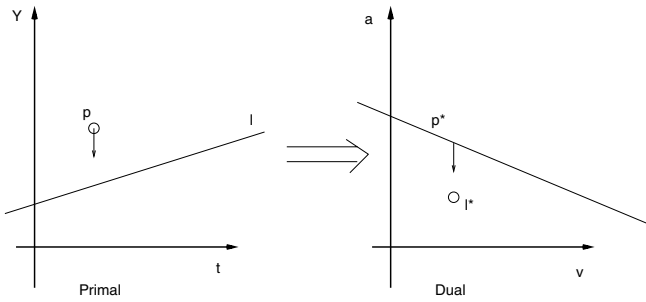
Fig. 2. Hough-X dual transformation: primal plane (*left*), dual plane (*right*)

$v_y(t - t_0)]$, where $V = [v_x, v_y]$ is its velocity vector. An example for the one-dimensional case is shown in Fig. 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 $t_{\text{now}} \leq t_{1q} \leq t_{2q}$), given the current motion information of all objects" (i.e., the *two-dimensional moving objects range (MOR) query* [27]).

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/Os needed to perform an operation. If N is the number of 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/Os 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/Os. 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 hyperplane 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 (Fig. 2). Thus we get the dual point (v, a) (this is called Hough-X transform in [24]). Similarly, a point $p = (t, y)$ in the primal space is mapped to line $a(v) = -tv + y$ in the dual space. An important property of the duality transform is that it preserves the above–below relationship. As is shown in Fig. 2, the dual line of point p is above the dual point l^* of line l.
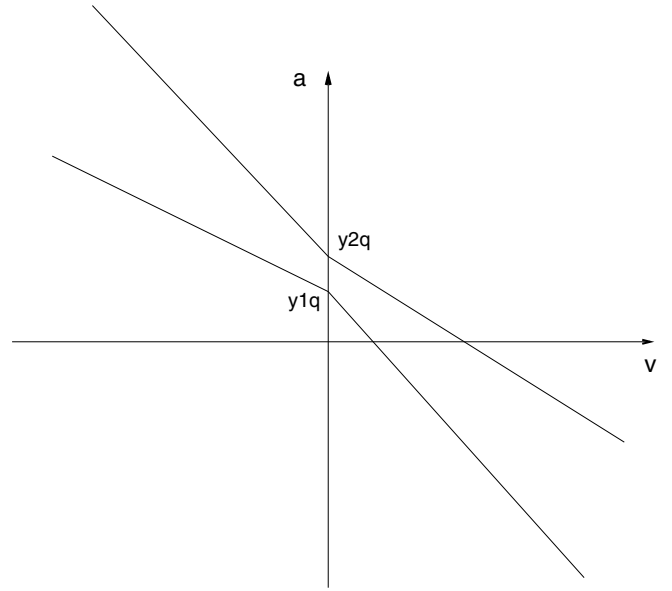

Fig. 3. Query on the Hough-X dual plane

Based on the above property, it is easy to show that the one-dimensional 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 $p_2 = (t_{2q}, y_{1q})$ (Fig. 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 [18], the query Q in the dual Hough-X plane (Fig. 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 at 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 [24]). Coordinate b is the point where the line intersects 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 a one-dimensional line is by plotting the trajectories as lines in the time-location (t, y) plane [same for the (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 (Fig. 1). In this setting, the query is expressed as the two-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 [21] or an R*-tree [8]. In this setting each line is approximated by a minimum bounding rectangle (MBR). Obviously, the MBR approximation has a 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 to "infinity", i.e., they share an endpoint on the time dimension.

Another approach is to partition the space into disjoint cells and store in each cell those lines that intersect it [52,12]. This could be accomplished by using an index such as an R+-tree [43], a cell tree [19], or a PMR-quadtree [42]. 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 multilevel 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_v^y for the points S_v^y 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 subqueries, one on each of the two projections, and taking their dual, 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_v^y and reports all dual points, with respect to (t, y) projection, that satisfy 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, it is also shown that the same bounds hold for the window range query.

Elbassioni et al. [16] proposed a technique (MB index) that partitions the objects along each dimension in the dual space and uses B-trees 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 $s = (\frac{n}{\log_B n})^{\frac{1}{d}}$, where $v_{i,max}$ and $v_{i,min}$ are, respectively, the maximum and minimum velocities in dimension i.

Saltenis et al. [40] 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 indicated in [39], 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 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.

An improved version of the TPR-tree, called TPR*-tree, was proposed by Tao et al. [50]. The authors provide a probabilistic model to estimate the number of disk accesses for answering predictive window range queries on moving objects and using this model they provide a hypothetical "optimal" structure for answering these queries. Then they show that the TPR-tree insertion algorithm leads to structures that are much worse than the optimal one. Based on that, they propose a new insertion algorithm, which, unlike the TPR-tree, considers multiple paths and levels of the index in order to insert a new object. Thus, the TPR*-tree is closer to the optimal structure than the TPR-tree. The authors suggest that, although the proposed insertion algorithm is more complex than the TPR-treeinsertion algorithm, it creates better trees (MBRs with tighter parameterized extends), which leads to better update performance. In addition, the TPR*-tree employs improved deletion and node-splitting algorithms that further improve the performance of the TPR-tree.

---

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

The STAR-tree, introduced by Procopiuc et al. [38], 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 reorganizes the grandchildren 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 into the future. It provides a tradeoff 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 the (x, t) or (y, t) plane 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 any time instant.

The R^{EXP}-tree, which extends the TPR-tree, was proposed for indexing moving objects with an expiration time in [41]. 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 are 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.

Very recently, the dual transformation proposed in this paper was adapted in [34], where the advantages over the TPR-tree methods have also been observed. Using the idea in [27], trajectories of d-dimensional moving objects are mapped into points in the dual two-dimensional space and a PR-quadtree is built to store the two-dimensional points. Similarly with [27] a different index is used for each of two reference times that change at periodic time intervals. At the end of each period, the old index is removed and a new index with a new reference point is built.

Algorithms to process nearest-neighbor queries using the dual transformation are presented in [26]. Such queries (as well as range) are also examined in [36], where techniques using indexing in the primal space are presented. Song et al. [45] propose a sampling technique for moving-point nearest-neighbor queries. They incrementally compute the results at predefined positions, using previous results to avoid recomputation. This approach has limitations since it deals with static objects. In addition, 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. [48] address the problem of time-parameterized queries in a moving objects environment. Time-parameterized queries retrieve the actual result at the time the query is issued, the validity period of the result given the current motion of the query and the database objects, and 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 [49]. Also related is work on dynamic queries over mobile objects [28]. Here queries are assigned to mobile observers and the result changes as the observer moves; query-processing techniques that reuse previously stored results are presented. Recently, continuous range queries in the spatiotemporal environment have been addressed in [25].

Prabhakar et al. [37] 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. It also 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. Prabhakar et al. [37] proposed a scheme that combines the two techniques in order to facilitate processing of ongoing queries and fast updates.

Pfoser et al. [35] propose two R-tree-based schemes for indexing the past trajectories of 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. Zhu at al. [55] present an approach to index trajectories that divides the trajectory predicates into topological and nontopological 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 [51]. 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 multidimensional 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 one-dimensional selectivities (i.e., the probability that the projection p_i of point p on the ith 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. [51] propose cost models for selectivity estimation of spatiotemporal 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 nonuniform datasets by employing spatiotemporal 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 times, 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/Os 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. Nonchronological 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 tradeoff 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 [44,53,54]. In another recent work, Güting et al. [20] 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 spatiotemporal DBMS extension. A query language for moving object environments, based on generalized distances, is presented in [30]. 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 one-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 [27]), on each of which we can apply the one-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 points in two dimensions, 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 hyperplanes. Since ϵ can be arbitrarily small, any algorithm that uses linear space for d-dimensional range searching has a worst-case query time of $O(N^{(d-1)/d} + K)$.

Here we show that a similar bound holds for the I/O complexity of simplex searching. Following the approach in [46] 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/Os 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/Os, at least a constant fraction of that many blocks has a constant fraction of their points in the answer of the query. But if the set of the queries has a small intersection, it follows that, in order to answer this set of queries in time $O(n^\delta)$, at least $\Theta(n^\delta) \cdot \Omega(n^{d(1-\delta)-\delta-\epsilon}) = \Omega(n^{d(1-\delta)-\epsilon})$ nodes have to be visited. It remains to show that such a set of queries exists. 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

two-dimensional simplex range query and thus the one-dimensional MOR query requires $O(\sqrt{n} + k)$ I/Os. Next we will present a dynamic, external-memory algorithm that achieves near 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 A (near) optimal solution

Matousek [29] gave a near 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. 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 $\max_i |S_i| < 2 \min_i |S_i|$, where $|S_i|$ is the cardinality of the set S_i, we say that the partition is balanced. Matousek [29] 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 [31] for decomposable problems we are able to 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/Os. 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 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 one-dimensional range searching.

This is one of our motivations for considering 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 that show that we cannot achieve a 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 $t_0 \le t_q \le t_0 + T$). Equivalently, this is a standard one-dimensional MOR query where $t_{1q} = t_{2q}$. We will call it a 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 N 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 $R = [y_l, y_r]$ in $O(\log_2 N + K)$ time, where K is the number of objects inside R.*

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 $y_i + v_i \cdot t_q$ at time t_q. Since the objects in the binary tree are stored by order at time t_q, if $y_i + v_i \cdot t_q < y_l$, then this is also true for all the objects of the left child of the root, in which case we eliminate the left child and recurse in the right child. Otherwise, we recurse in 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 $O(N \log_2 N + M \log_2 M)$, where M is the number of crossings in the time period $[0, T]$.*

Proof. Let $\{p_1, \ldots, p_N\}$ be the ordering of N objects at time 0, sorted by position. Assume we maintain this ordering in a linked list L_0. At time T, the position of object i is $y_i + v_i \cdot T$. 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 $t(j) < t(i)$.
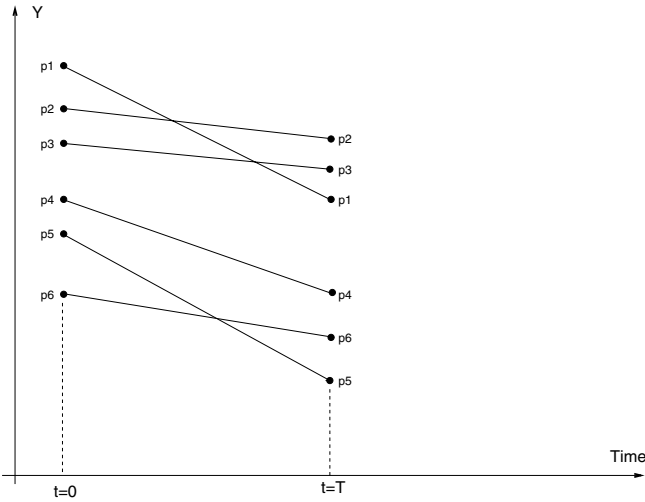
Fig. 4. Object trajectory crossings

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 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. $\square$

An example is shown in Fig. 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/Os, 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 jth 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 than or equal to t. While

this achieves $O(\log_2(N + M))$ query time, it uses $O(MN)$ 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, $B(0)$, which 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 we find the record in the auxiliary array with the largest timestamp that is less than 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 $O(\log_B n \cdot \log_B m)$ 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 $O(\log_B m)$ time (searching the history of the B-tree root node), while all other nodes of $B(t)$ are found in time $O(\log_B n)$ using the appropriate parent to child pointers. $\square$

The following theorem follows from the previous lemmas:

Theorem 2 *Given N objects and a time limit T, a one-dimensional MOR1 query can be answered in time $O(\log_B(n + m))$ 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 occurs. 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.

Fig. 5. Query on the dual Hough-Y plane

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 for indexing point data [17]. All these structures were designed to address *orthogonal* queries, i.e., queries expressed as a multidimensional hyperrectangle. However, most of them can be easily modified to address nonorthogonal queries like simplex queries.

Goldstein et al. [18] 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 hyperrectangle overlap. As mentioned in [18], this method is applicable not only 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 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 (Fig. 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 Fig. 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 $E = E^{\text{HoughY}} = E_1^{\text{HoughY}} + E_2^{\text{HoughY}}$, which is computed as:
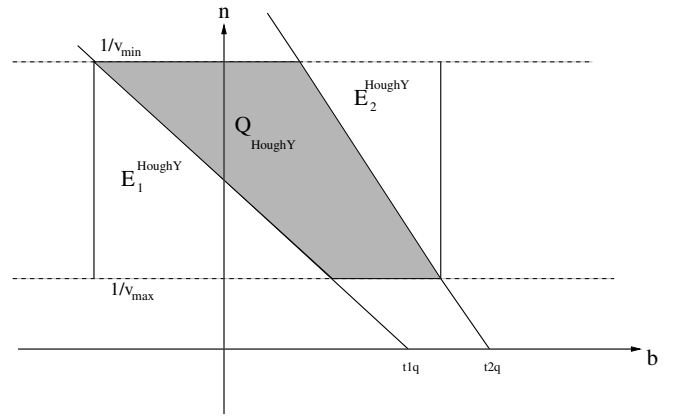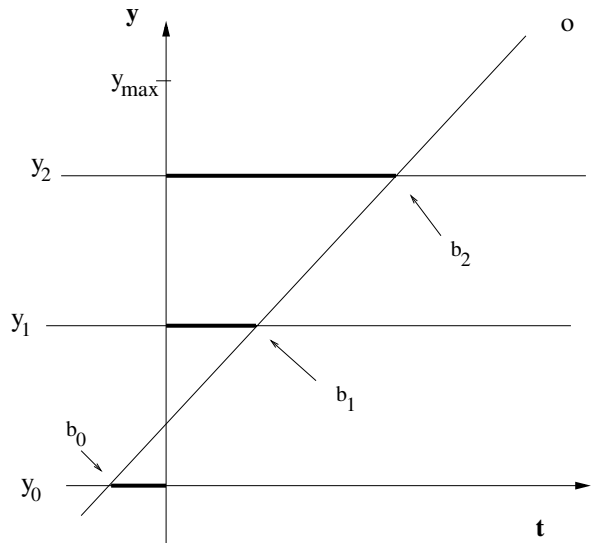

Fig. 6. Coordinate b as seen from different "observation" points

$$E^{\text{HoughY}} = \frac{1}{2}\left(\frac{v_{\max} - v_{\min}}{v_{\min} \cdot v_{\max}}\right)^2$$
$$(\mid y_{2q} - y_r \mid + \mid y_{1q} - y_r \mid). \tag{1}$$

The objective is to minimize E since it represents a measure of the extra I/Os that an access method will have to perform for solving a 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 ith index stores the b coordinates of the data points using $y_i = \frac{y_{\max}}{c} \cdot i, i = 0, \ldots, c-1$ (Fig. 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 *subterrain*. 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 covers a subterrain:

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

$$E \leq \frac{1}{2} \left(\frac{v_{\max} - v_{\min}}{v_{\min} \cdot v_{\max}} \right)^2 \left(\frac{y_{\max}}{c} \right). \tag{2}$$

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 subterrains, which implies that if a query is executed at a single observation index, area E becomes large. To bound E we index each subterrain, too. Each of the c subterrain indices records the time interval when a moving object was in the subterrain. Then the query is decomposed into a collection of smaller subqueries: one subquery per subterrain fully contained by the original query interval and one subquery for each of the original query's endpoints. The subqueries 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 subterrain, we could use an external memory interval tree [6], which would answer a subterrain 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 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 stems 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. One simple solution to this problem is to create a new index with a newer observation time every T time instants and at the same time remove the index with the oldest observation time [27,34]. Note that this problem does not exist in 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 (Fig. 7). lines in the plane. The reason is that a line in the space has 4 degrees of freedom. Therefore, if we apply the dual transformations directly, we get a 4- dimensional dual point. 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 two-dimensional 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.
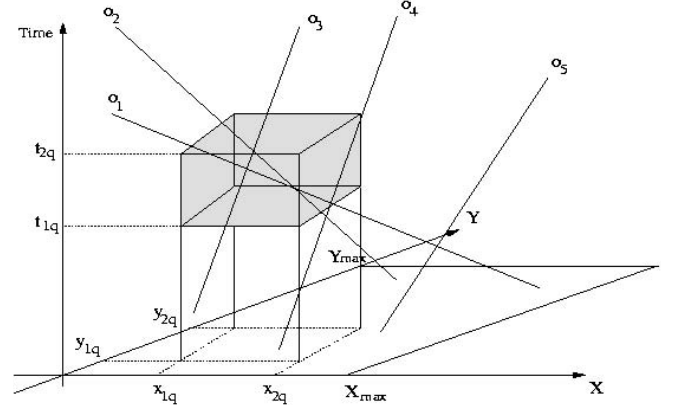

Fig. 7. Trajectories and query in (x, y, t) space

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 as 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 two-dimensional motion into two one-dimensional 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 three-dimensional hyperplanes, and the projections of the query on the (t, x) and (t, y) planes are wedges, as in the one-dimensional case.

The two-dimensional query is decomposed into two one-dimensional 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 Fig. 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 Sect. 4, if the simplex query was answered approximately, the query area would be enlarged by $E^{\mathrm{HoughY}} = E_1^{\mathrm{HoughY}} + E_2^{\mathrm{HoughY}}$ (the triangular areas in Fig. 5). Also, let the actual area of the simplex query be Q^{HoughY}. Similarly, on the dual Hough-X plane (Fig. 3), let Q^{HoughX} be the actual area of the query and E^{HoughX} be the enlargement. The algorithm chooses the projection that minimizes the following criterion κ:

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

The intuition for this heuristic [33] is that simplex queries in the dual space are not aligned with the MBRs of the underlying index (Fig. 8). Therefore, we would like to ask the query in the projection about 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.
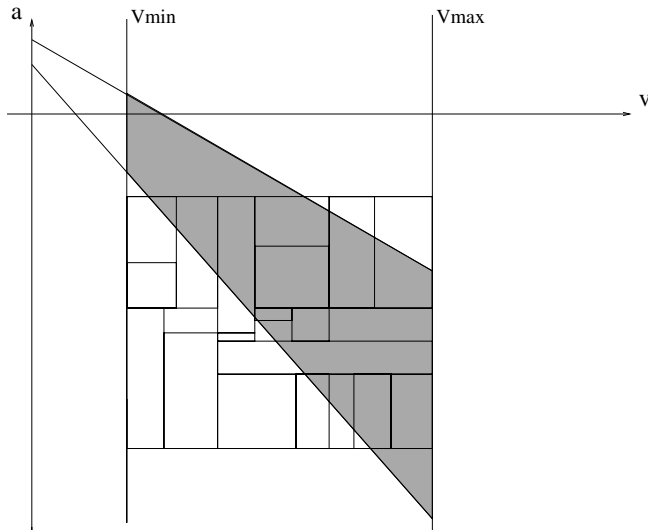

Fig. 8. Simplex query in dual space, not aligned with MBRs of underlying index

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 two-dimensional query is presented next.

1. Decompose the query into two one-dimensional 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 (Sect. 4) since they are components of the proposed two-dimensional solution (Sect. 5). For the two-dimensional space we compare our approach with the TPR-tree [40,41]. We chose the TPR-tree as a very efficient representative of the nondual transformation methods (Sect. 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 that stores trajectories as line segments.

First we describe the way experimental data are generated. At time $t = 0$ we generate the initial locations of N mobile objects uniformly distributed on the (line segment) terrain $[0, 1000]$. We vary N from $100K$ to $500K$. The speeds are generated uniformly from $v_{\min} = 0.16$ to $v_{\max} = 1.66$ and the direction is randomly positive or negative. (Note that 0.16 miles/min is equal to 10 miles/h and 1.66 miles/min is equal to 100 miles/h.) 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 ranges between 0 and *WT*. We actually generate two sets of query 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 implement the traditional R-tree approach using an R*-tree [8] with page size 4K. To represent a line segment, we use four 4-byte numbers (the two endpoints) 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 use two-dimensional R*-trees to index the dual points. These R*-trees are appropriately modified to index points instead of rectangles. We use R-trees over the B+-trees proposed in [27] since we got much better query performance. Thus 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 do not implement the interval trees since the cost of creating, storing, and updating these structures is high and they are needed only for very large queries, which are not typical.

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 three or four 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/Os per query for queries with varying *WT*, while Fig. 10 depicts results for queries with varying *YQMAX*. These experiments were 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

Fig. 11. One-dimensional case: query performance for varying number of objects

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 use $c = 1$, $c = 2$, and $c = 4$, and we observe that it outperforms the Hough-X query performance even with $c = 1$.

Figures 12 and 13 plot the space consumption and the average number of I/Os 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" 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 methods with $c = 2$ and $c = 4$ use more space due the use of c observation indices. Regarding update processing, the line segment method exhibits the worst update performance that increases drastically as the number of objects increases. Most of this update cost comes from deletions where many tree paths are typically visited. The update performance of the Hough-
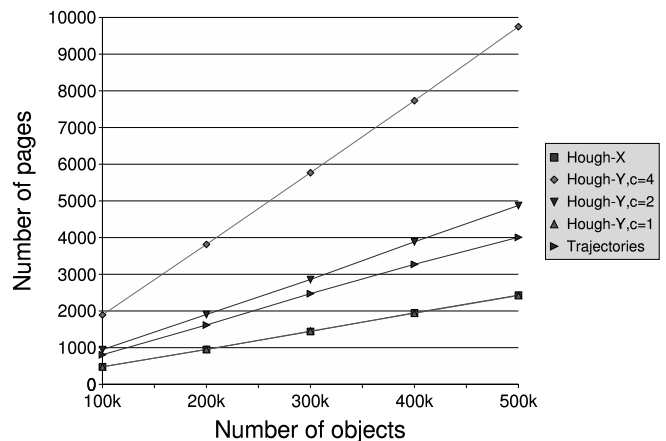

Fig. 9. One-dimensional case: query performance for varying WT
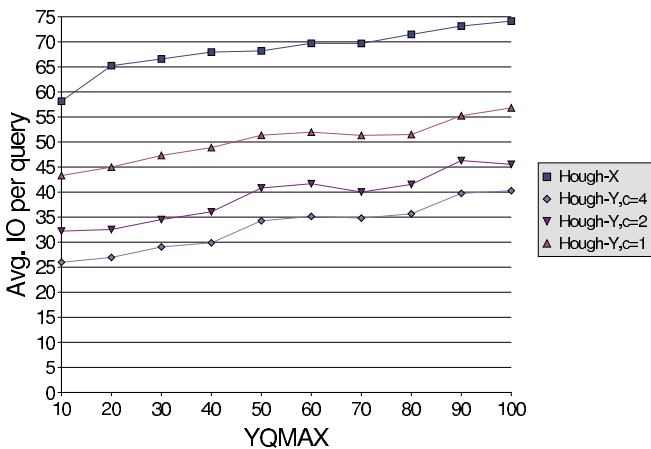
Fig. 10. One-dimensional case: query performance for varying YQmax

Fig. 12. One-dimensional case: space consumption for varying number of objects

G. Kollios et al.: Indexing mobile objects using dual transformations
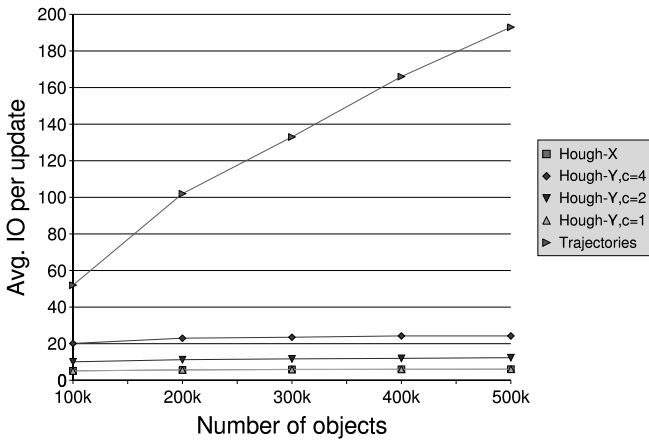
Fig. 13. One-dimensional case: update performance for varying number of objects

X and the Hough-Y approaches 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 ($c = 1$) increases slightly 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 segment method. Figures 9–13 show the clear tradeoff between c and query/update performance for the Hough-Y method.

6.2 Two-dimensional case

Experimental setting. For the two-dimensional MOR query we generated a variety of datasets using the TPR-tree's generator [40] as well as our own generator.

The datasets created with the TPR generator use parameters suggested in [40], that is, we assume objects moving on a finite terrain having size 1000×1000 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 equal probability to one of three possible groups having a maximum velocity of 0.75 (slow), 1.5 (medium), and 3 km/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 one sixth of the route), then they maintain this speed (for the next two thirds of the route), and finally they decelerate to 0 km/min (during the last one sixth of the route). We also generated a dataset in which objects can move randomly on the terrain without destinations (this is termed UNI in [40]).

Each simulation scenario runs for 600 time instants, where each instant corresponds to 1 min [40] (i.e., the simulation corresponds to 10 h). 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 first need to insert the 100K objects into the index. For example, the ND60 dataset had 1.07 million insertions and 0.97 million deletions. The other datasets had similar insertion/deletion ratios.

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 ten instants.

The datasets generated using our own generator assume a network of routes that 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 1000×1000 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 generate 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 generate 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 [40]). Performance is optimized for queries issued within H from the last update. Recall that the computation of H uses the *average* update interval UI. Hence, there may be 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 [41]. 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 takes to receive the last B entries. The parameter W is
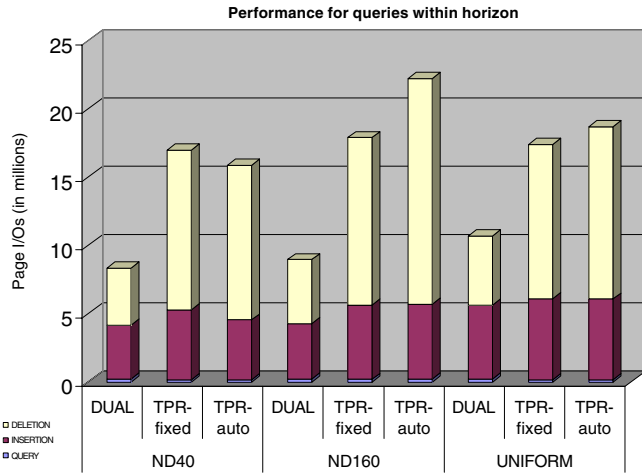
Fig. 14. ND/UNI datasets: queries within the horizon, overall I/O comparison

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 Sect. 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 Figs. 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 Sect. 5.

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 ten updates during the simulation [40]; 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 a 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 it 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, Figs. 16 and 17 depict the average page I/O per query and update, respectively.
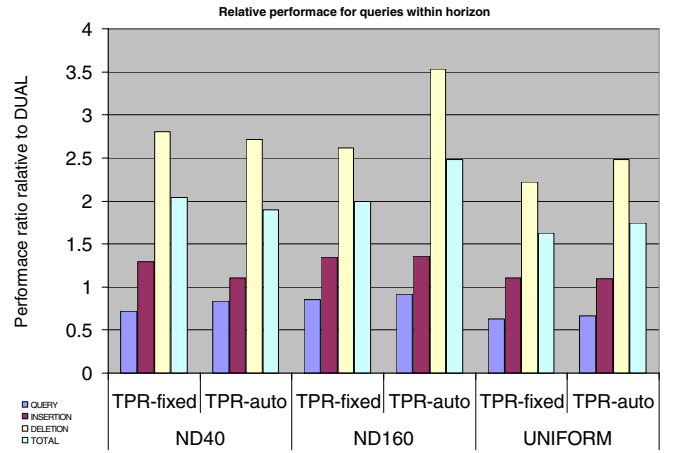

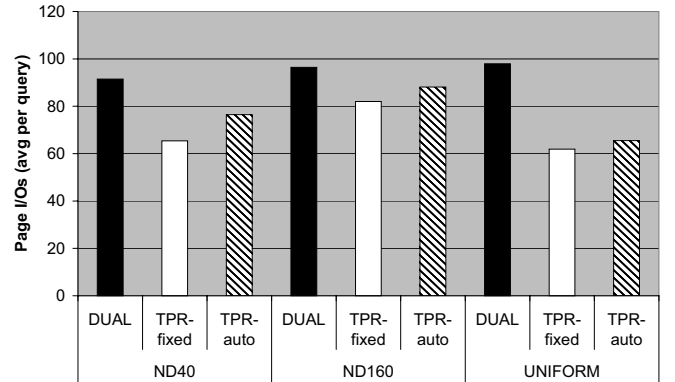**Fig. 15.** ND/UNI datasets, queries within the horizon, ratio of performance relative to DUAL

Fig. 16. ND/UNI datasets, queries within the horizon, average I/O per query
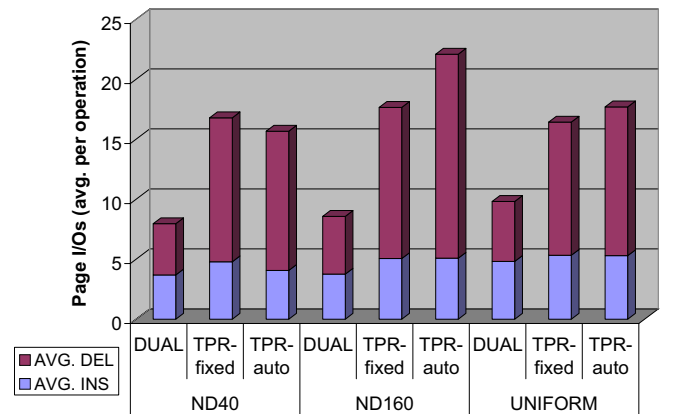

Fig. 17. ND/UNI datasets, queries within the horizon, average I/O per update

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. All methods seem to scale up graciously (the relative ratios remain similar). Again, the
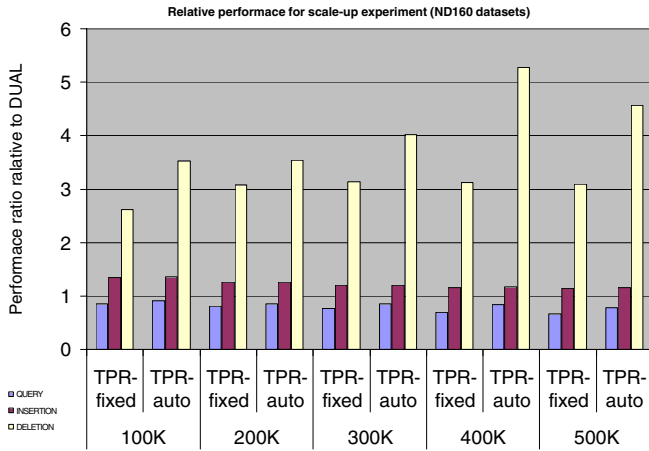
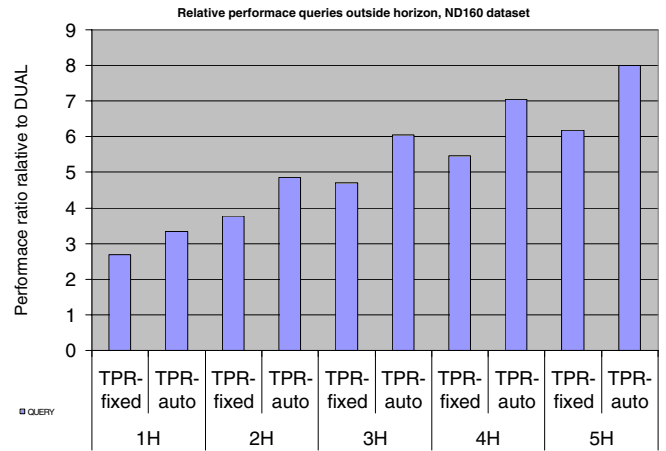**Fig. 18.** Varying the number of moving objects

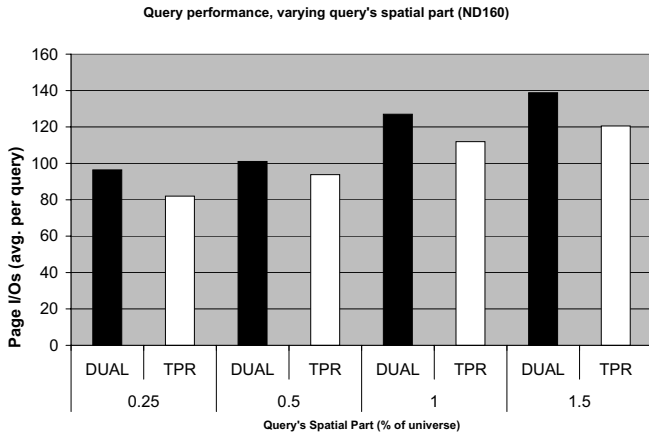**Fig. 20.** ND160 dataset, queries outside the horizon, ratio of performance relative to DUAL

Fig. 19. Varying the size of the spatial predicate

Fig. 21. Space consumption for ND/UNI

Fig. 22. Scale-up experiment: space consumption

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 at ten 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, Fig. 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 to 5H outside the horizon H. The update times are not shown as they are similar to those of Fig. 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 decreases the further away it moves from the fixed horizon.

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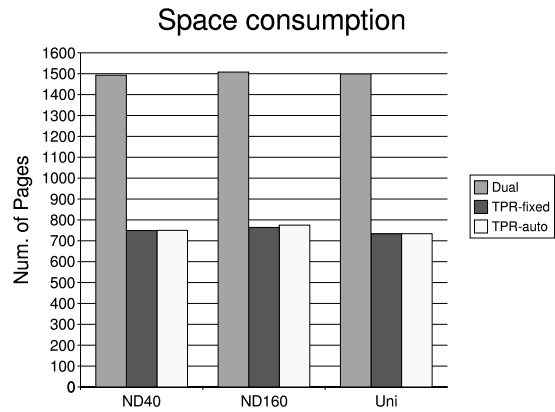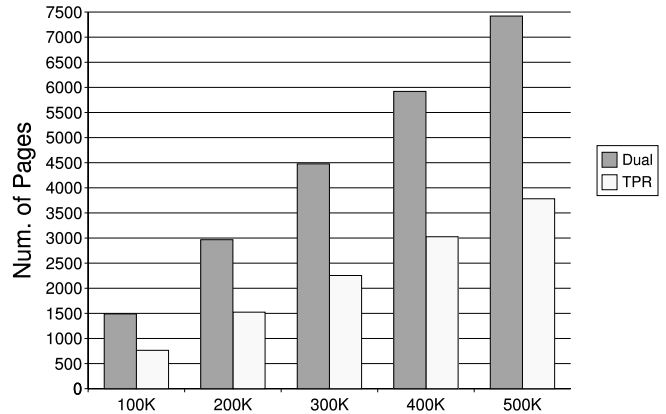
of the TPR-trees since each point is stored in two indices – one for each dimension. Figure 22 depicts how the space consumption scales up as the number of objects increases for the ND160 dataset. As expected, the space consumption of all methods increases linearly with the number of moving objects.

The next figures present the results for the "freeway" datasets created with our own generator. In general, we get results very similar to those obtained with the TPR-generator datasets. Figure 23 depicts the performance of the TPR-trees as
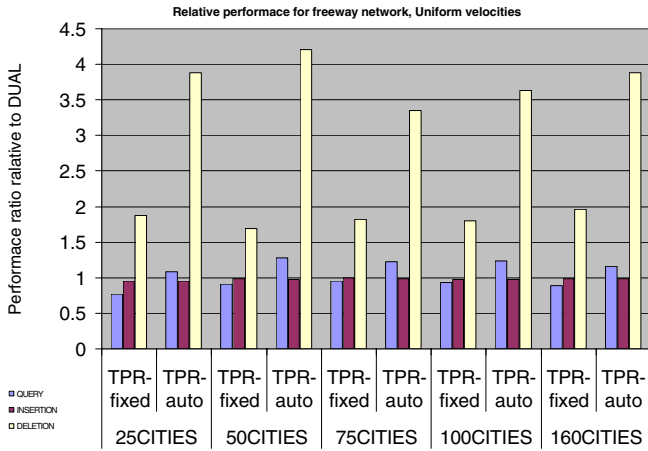
Fig. 23. Freeway network, uniform velocities, queries within the horizon, ratio of performance relative to DUAL
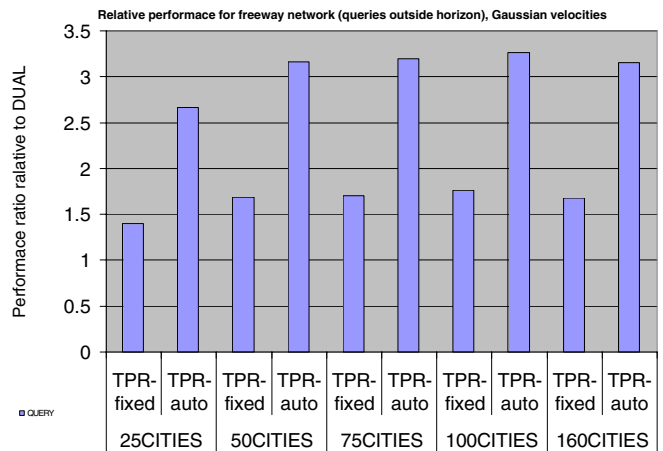

Fig. 25. Freeway network, Gaussian velocities, queries outside the horizon by 1H, ratio of performance relative to DUAL
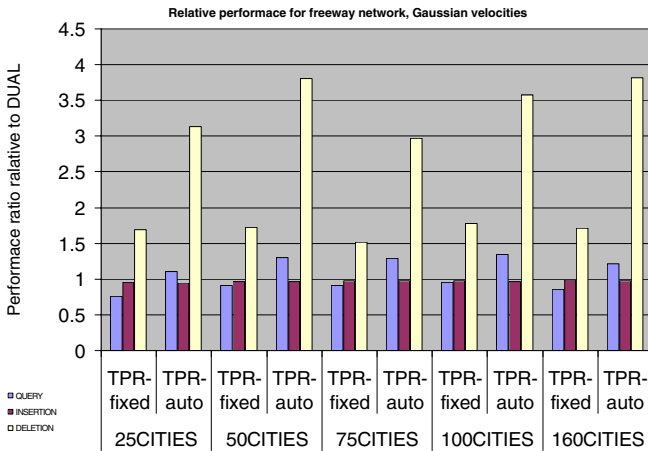

Fig. 24. Freeway network, Gaussian velocities, queries within the horizon, ratio of performance relative to DUAL

Fig. 26. Query performance for increasing current time
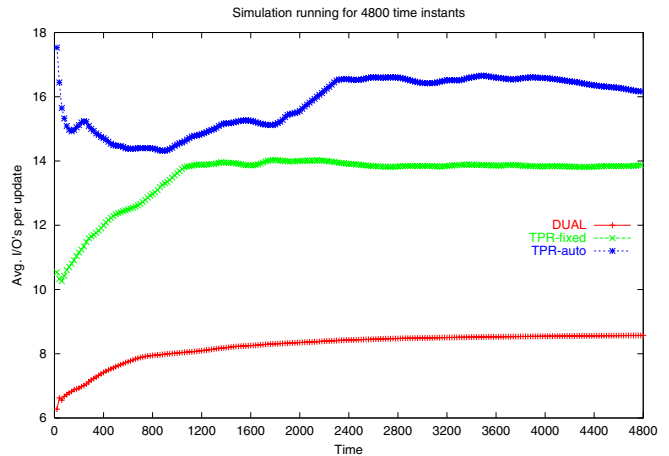
Fig. 27. Update performance for increasing current time

a ratio relative to DUAL for uniformly chosen velocities, with a varying number of cities (destinations) and queries within the horizon. The TPR-tree again has better query performance, but it is closer to DUAL than before. Interestingly, 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 Figs. 24 and 25.

Finally, we perform an experiment where the scenario runs for 4800 time instants. We measure the performance of the index every 20 time instants and compute the average query and update performance until the current time. In Fig. 26 we plot the query performance, for ND40 and queries inside the horizon. The query performance of the DUAL approach deteriorates with time since most of the objects are moving. On the other hand, the TPR-trees deteriorate fast at the beginning of the simulation, but at some point they stabilize, around an average of 80 I/Os 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 up-
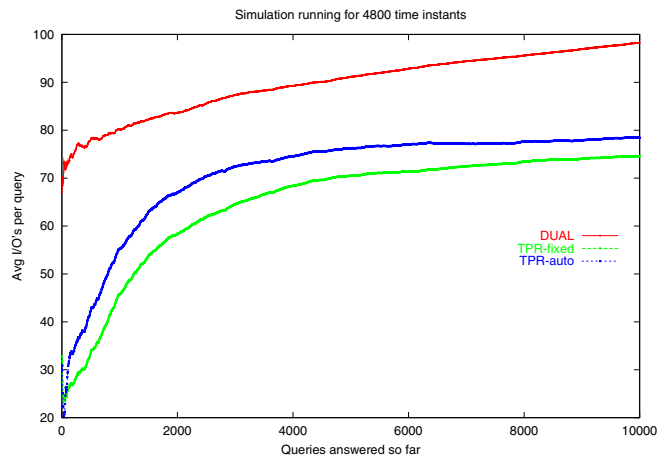
date performance per update for the same experiment. In that case, all indices stabilize after some initial time period. The update performance of DUAL is about 1.6 times better than the TPR-fixed tree and 1.85 times better than the TPR-auto tree.

Discussion. The two-dimensional 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 tree 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 a predefined horizon definition is possible, for others it may not be. In contrast, the DUAL method does not depend upon knowing the characteristics of the anticipated workload (i.e., the parameter *UI*), neither does it assume 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 tree, where the horizon is automatically selected based on the previous history of updates, did not seem to perform as well as the TPR-fixed tree; 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 costs 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 reorganizes 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 outside 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 direction of future 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. The relationship of indexing techniques and protection of privacy of mobile users is also a very interesting problem that we plan to consider. Finally, techniques for answering aggregate complex queries, such as predicting and reporting areas with high density of mobile objects, are also of high practical interest.

References

1. Agarwal PK, Arge L, Erickson J (2000) Indexing moving points. In: Proceedings of the 19th ACM symposium on principles of database systems, pp 175–186
2. Agarwal PK, Arge L, Erickson J, Franciosa PG, Vitter JS (1998) Efficient searching with linear constraints. In: Proceedings of the 17th ACM symposium on principles of database systems, pp 169–178
3. Agarwal PK, Har-Peled S (2001) Maintaining approximate exten measures of moving points. In: Proceedings of the 12th ACM-SIAM symposium on discrete algorithms, pp 148–157
4. Aggarwal A, Vitter JS (1988) The input/output complexity of sorting and related problems. Commun ACM 31(9):1116–1127
5. Arge L, Samoladas V, Vitter JS (1999) On two-dimensional indexability and optimal range search indexing. In: Proceedings of the 18th ACM PODS, pp 346–357
6. Arge L, Vitter JS (1996) Optimal dynamic interval management in external memory. In: Proceedings of the 37th annual symposium on foundations of computer science, pp 560–569
7. Basch J, Guibas L, Hershberger J (1997) Data structures for mobile data. In: Proceedings of the 8th ACM-SIAM symposium on discrete algorithms, pp 747–756
8. Beckmann N, Kriegel H, Schneider R, Seeger B (1998) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of ACM SIGMOD, Atlantic City, NJ, pp 322–331
9. Chazelle B, Rosenberg B (1992) Lower bounds on the complexity of simplex range reporting on a pointer machine. In: Proceedings of the 19th international colloquium on automata, languages and programming. Lecture notes in computer science, vol 623. Springer, Berlin Heidelberg New York, pp 439–449
10. Choi Y-J, Chung C-W (2002) Selectivity estimation for spatio-temporal queries to moving objects. In: Proceedings of ACM SIGMOD, Madison, WI, pp 440–451
11. Chomicki J, Revesz P (1999) A geometric framework for specifying spatiotemporal objects. In: Proceedings of the 6th international workshop on time representation and reasoning, pp 41–46
12. Chon HD, Agrawal D, El Abbadi A (2002) Query processing for moving objects with space-time grid storage model. In: Proceedings of the 3rd international conference on mobile data management, pp 121–126

13. Cole R (1986) Searching and storing similar lists. J Algorithms 7(2):202–220

14. Driscoll J, Sarnak N, Sleator D, Tarjan RE (1989) Making data structures persistent. J Comput Sys Sci 38(1):86–124

15. http://europa.eu.int/eur-lex/pri/en/oj/dat/2002/l_201/l_20120020731en00370047.pdf (2002)

16. Elbassioni KM, Elmasry A, Kamel I (2003) An efficient indexing scheme for multi-dimensional moving objects. In: Proceedings of the 9th international conference on database theory (ICDT), pp 425–439

17. Gaede V, Günther O (1998) Multidimensional access methods. ACM Comput Surv 30(2):170–231

18. Goldstein J, Ramakrishnan R, Shaft U, Yu JB (1997) Processing queries by linear constraints. In: Proceedings of the 16th ACM PODS symposium on principles of database systems, Tucson, AZ, pp 257–267

19. Günther O (1989) The design of the cell tree: an object-oriented index structure for geometric databases. In: Proceedings of the 5th IEEE international conference on data engineering, Los Angeles, pp 598–605

20. Güting RH, Böhlen MH, Erwing M, Jensen CS, Lorentzos NA, Schneider M, Vazirgiannis M (2000) A foundation for representing and querying moving objects. ACM Trans Database Sys 26(1):1–42

21. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD, Boston, pp 47–57

22. Hadjieleftheriou M, Kollios G, Gunopulos D, Tsotras V (2003) On-line discovery of dense areas in spatio-temporal databases. In: Proceedings of the 8th SSTD, pp 306–324

23. Hadjieleftheriou M, Kollios G, Tsotras V (2003) Performance evaluation of spatio-temporal selectivity estimation techniques. In: Proceedings of the 15th international conference on scientific and statistical database management, pp 202–211

24. Jagadish HV (1990) On indexing line segments. In: Proceedings of the 16th international conference on very large data bases, Brisbane, Queensland, Australia, pp 614–625

25. Kalashnikov DV, Prabhakar S, Hambrusch SE, Aref WG (2002) Efficient evaluation of continuous range queries on moving objects. In: Proceedings of the 13th international conference DEXA, pp 731–740

26. Kollios G, Gunopulos D, Tsotras V (1999) Nearest neighbor queries in a mobile environment. In: Proceedings of the 1st workshop on spatio-temporal database management, Edinburgh, UK, pp 119–134

27. Kollios G, Gunopulos D, Tsotras V (1999) On indexing mobile objects. In: Proceedings of the 18th ACM symposium on principles of database systems, pp 261–272

28. Lazaridis I, Porkaew K, Mehrotra S (2002) Dynamic queries over mobile objects. In: Proceedings of the 8th international conference on extending database technology, pp 269–286

29. Matousek J (1992) Efficient partition trees. Discrete Comput Geom 8:432–448

30. Mokhtar H, Su J, Ibarra OH (2002) On moving object queries. In: Proceedings of the 21st ACM PODS, pp 188–198

31. Overmars MH (1983) The design of dynamic data structures. Lecture notes in computer science, vol 156. Springer, Berlin Heidelberg New York

32. Papadias D, Tao Y, Kalnis P, Zhang J (2002) Indexing spatio-temporal data warehouses. In: Proceedings of the 18th international conference on data engineering, pp 166–175

33. Papadopoulos D, Kollios G, Gunopulos D, Tsotras VJ (2002) Indexing mobile objects on the plane. In: Proceedings of the 5th international workshop on mobility in databases and distributed systems (DEXA), Aix-en-Provence, France, pp 693–697

34. Patel J, Chen Y, Chakka VP (2004) STRIPES: an efficient index for predicted trajectories. In: Proceedings of ACM SIGMOD

35. Pfoser D, Jensen C, Theodoridis Y (2000) Novel approaches in query proceedings for moving objects. In: Proceedings of the 26th international conference on very large data bases, pp 395–406

36. Porkaew K, Lazaridis I, Mehrotra S (2001) Querying mobile objects in spatio-temporal databases. In: Proceedings of the 7th SSTD, pp 59–78

37. Prabhakar S, Xia Y, Kalashnikov DV, Aref W, Hambrusch S (2002) Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects. In: IEEE Trans Comput 51(10):1124–1140

38. Procopiuc CM, Agarwal PK, Har-Peled S (2002) Star-tree: an efficient self-adjusting index for moving objects. In: Proceedings of the 4th workshop on algorithm engineering and experiments, pp 178–193

39. Saltenis S, Jensen C, Leutenegger S, Lopez MA (1999) Indexing the positions of continuously moving objects. Time-Center Technical Report TR-44. http://www.cs.auc.dk/research/DP/tdb/TimeCenter/TimeCenterPublications/TR-44.pdf

40. Saltenis S, Jensen C, Leutenegger S, Lopez MA (2000) Indexing the positions of continuously moving objects. In: Proceedings of ACM SIGMOD, pp 331–342

41. Saltenis S, Jensen CS (2002) Indexing of moving objects for location-based services. In: Proceedings of the 18th international conference on data engineering, San Jose, CA, pp 463–472

42. Samet H (1990) The design and analysis of spatial data structures. Addison-Wesley, Reading, MA

43. Sellis T, Roussopoulos N, Faloutsos C (1987) The R+-tree: a dynamic index for multi-dimensional objects. In: Proceedings of the 13th international conference on very large data bases, Brighton, UK, pp 507–518

44. Sistla AP, Wolfson O, Chamberlain S, Dao S (1997) Modeling and querying moving objects. In: Proceedings of the 13th international conference on data engineering, pp 422–432

45. Song Z, Roussopoulos N (2001) K-nearest neighbor search for moving query points. In: Proceedings of the 7th SSTD, Redondo Beach, CA, pp 79–96

46. Subramanian S, Ramaswamy S (1995) The P-range tree: a new data structure for range searching in secondary memory. In: Proceedings of the 6th annual symposium on discrete algorithms, New York, pp 378–387

47. Tao Y, Kollios G, Considine J, Li F, Papadias D (2004) Spatio-temporal aggregation using sketches. In: Proceedings of the 20th international conference on data engineering, pp 214–226

48. Tao Y, Papadias D (2002) Time-parameterized queries in spatio-temporal databases. In: Proceedings of ACM SIGMOD, Madison, WI, pp 334–345

49. Tao Y, Papadias D, Qiongmao S (2002) Continuous nearest neighbor search. In: Proceedings of the 28th international conference on very large data bases, pp 287–298

50. Tao Y, Papadias D, Sun J (2003) The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In: Proceedings of the 29th international conference on very large data bases, pp 790–801

51. Tao Y, Sun J, Papadias D (2003) Selectivity estimation for predictive spatio-temporal queries. In: Proceedings of the 19th international conference on data engineering, Bangalore, India, pp 417–428

52. Tayeb J, Olusoy O, Wolfson O (1998) A quadtree-based dynamic attribute indexing method. Comput J 41(3):185–200

53. Wolfson O, Chamberlain S, Dao S, Jiang L, Mendez G (1998) Cost and imprecision in modeling the position of moving ob-

jects. In: Proceedings of the 14th international conference on data engineering, Orlando, FL, pp 588–596

54. Wolfson O, Xu B, Chamberlain S, Jiang L (1998) Moving objects databases: issues and solutions. In: Proceedings of the 11th international conference on scientific and statistical database management, Capri, Italy, pp 111–122

55. Zhu H, Su J, Ibarra OH (2002) Trajectory queries and octagons in moving object databases. In: Proceedings of the 11th ACM international conference on information and knowledge management, pp 413–421