On Indexing Mobile Objects

George Kollios
Polytechnic University
gkollios@milos.poly.edu

Dimitrios Gunopulos
University of California, Riverside
dg@cs.ucr.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

Abstract

We show how to index mobile objects in one and two dimensions using efficient dynamic external memory data structures. The problem is motivated by real life applications in traffic monitoring, intelligent navigation and mobile communications domains. For the 1-dimensional case, we give (i) a dynamic, external memory algorithm with guaranteed worst case performance and linear space and (ii) a practical approximation algorithm also in the dynamic, external memory setting, which has linear space and expected logarithmic query time. We also give an algorithm with guaranteed logarithmic query time for a restricted version of the problem. We present extensions of our techniques to two dimensions. In addition we give a lower bound on the number of I/O's needed to answer the d-dimensional problem. Initial experimental results and comparisons to traditional indexing approaches are also included.

1 Introduction

Traditional database management systems assume that data stored in the database remain constant until explicitly changed through an update. While this model serves well many applications where data changes in discrete steps, it is not appropriate for applications with continuously changing data. One such application is a "motion" database that stores the location of mobile objects (e.g. cars). Since objects change location continuously, one would have to update the database at every unit of time. This is clearly an inefficient and infeasible solution considering the prohibitively large update overhead.

A better approach is to abstract each object's location as a function of time $f(t)$, and update the database only when the parameters of f change (for example when the speed or the direction of a car changes). Using $f(t)$ the "motion" database can compute the location of the mobile object at any time in the future. While this approach minimizes the update overhead, it introduces a variety of novel problems (such as the need for appropriate data models, query languages and query processing and optimization techniques) since the database is not directly storing data values but functions to compute these values. Motion database problems have recently attracted the interest of the research community: ([33, 36, 37]) present the Moving Objects Spatio-Temporal (MOST) model and a language (FTL) for querying the current and future locations of mobile objects; ([15]) proposes a model that tracks and queries the history (past routes) of mobile objects, based on new spatio-temporal data types. Another spatiotemporal model appears in [11]. Spatio-temporal queries about mobile objects have important applications in traffic monitoring, intelligent navigation and mobile communications domains. For example in databases that track cars in a highway system, we can detect future congestion areas. In mobile communications we can allocate more bandwidth for areas where high concentration of mobile phones is approaching. There is already a GIS system [4] that supports tracking and querying of mobile objects.

In this paper we focus on the problem of indexing mobile objects. In particular we examine how to efficiently address range queries over the object locations into the future. An example of such a spatio-temporal query is: "Report all the objects that will be inside a query region P after 10 minutes from now". Note that the answer to this query 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.

While in general an object could move anywhere in the 3-dimensional space using some rather complex motion, we limit our treatment to objects moving in 1- and 2-dimensional spaces and whose location is described by a linear function of time. There is a strong motivation for such an approach based on the real-world applications we have in mind: straight lines are usually the faster way to get from one point to another; cars move in networks of highways which can be approximated by connected straight line segments on a plane; this is also true for routes taken by airplanes or ships. In addition, solving these simpler 1- and 2-dimensional problems may provide intuition for addressing the more difficult problem of indexing general multidimensional functions.

2 Problem Description

We consider a database that keeps track of mobile objects moving in one and two dimensions. We model the objects as points that move with a constant velocity starting from a specific location at a specific time instant. Using this information we can compute the location of an object at any time in the future for as long as its movement characteristics remain the same. In one dimension, an object started from location y_0 at time t_0 with a velocity v (v can be positive or negative) will be in location $y_0 + v(t - t_0)$ at time $t > t_0$. Similarly for objects moving in two dimensions. Objects are responsible to update their motion information, every time when their speed or direction changes. Also, we assume that the objects can move inside a finite terrain (a line segment in one dimension or a rectangle in two). Thus when an object has reached the limits, it has to issue an update (either because it is deleted or it is reflected). Finally, we allow to insert a new object or to delete an old one, eg. the system is dynamic.

We would like to answer efficiently proximity queries among the mobile objects. In particular, we are interesting to answering queries of the form: "Report the objects that reside inside the interval $[y_{1q}, y_{2q}]$ (or the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$ in two dimensions) at the time instants between time t_{1q} and t_{2q}, (where $t_{now} \leq t_{1q} \leq t_{2q}$), given the current motion information of all objects". We call this type of queries the *one dimensional MOR query* for objects moving in one dimension and the *two dimensional MOR query* for objects moving in two dimensions.

We consider the problem in the standard external memory model of computation[3]. In this model each disk access (an I/O) transmits in a single operation B units of data. We call B the page capacity. We measure the efficiency of an algorithm in terms of the number of I/O's to perform an operation. If N is the number of the mobile objects and K is the number of objects reported by the MOR query, then the minimum number of pages to store the database is $n = \lceil \frac{N}{B} \rceil$ and the minimum number of I/O's to report the answer is $k = \lceil \frac{K}{B} \rceil$. 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 execute $O(\log_B n + k)$ I/O's. Note that $\log_B n$ is for the external memory model different than $\log_2 n$ since B is not a constant but a problem variable.

In the next section we consider the problem of indexing mobile objects on a line. First we present two geometric representations of the problem (sections 3.1, 3.2) and discuss why storing the trajectories of mobile objects as lines in traditional indexing methods (like R-trees) does not work. We then give a lower bound on the static version of the problem by reducing it to simplex range searching in two dimensions (section 3.3). We also give a dynamic external memory algorithm with matching upper bound that is based on partition trees [27] (section 3.4). Unfortunately this algorithm is not practical because it has a large hidden constant factor. Accordingly we turn our attention into algorithms designed to use linear space and to work well in the average case. We present two approaches, a simple one based on *kd*-trees and a more sophisticated one based on B+-trees (section 3.5). Our experimental results (section 5) show that the B+-trees based approach outperforms the *kd*-tree and the simple R-tree approach. In section 3.6, we impose the restriction that we can only answer queries within a fixed time window in the future. For this setting we present a data structure with logarithmic query time. The space requirement of our method varies between linear and quadratic, depending on the size of the time window and the distribution of the velocities of the mobile objects.

In section 4 we extend the previous results for two versions of the 2-dimensional case. In the first version we make the assumption, motivated from practice, that the objects move on a network of 1-dimensional routes (we call this version, the 1.5-dimensional problem). In the second version the objects are allowed to move arbitrarily on a plane (the general 2-dimensional problem).

3 Indexing in one dimension

We begin with the simpler problem of objects moving on an 1-dimensional line. We partition the mobile objects into two categories, the objects with low speed $v \approx 0$ and the objects with speed between a minimum v_{min} and maximum speed v_{max}. We consider here the "moving" objects, eg. the objects with speed greater than v_{min}. We discuss the case of slowly moving objects in section 3.6.

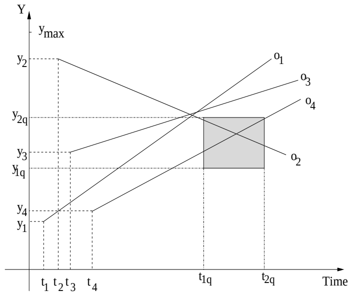


Figure 1: Trajectories and query in (t, y) plane.

We assume that the objects move on the $y-$axis between 0 and y_{max} and that an object can update its motion information whenever it changes. We treat an update as a deletion of the old information and an insertion of the new one. Next we give different geometric representations of the problem and for each one we discuss access structures to efficiently address MOR queries.

3.1 Space-time representation

In this representation we plot the trajectories of the mobile objects as lines in the time-location (t, y) plane. The equation of each line is $y(t) = vt + a$ where v is the slope (the velocity in our case) and a is the intercept, that can be computed by the motion information. Figure 1 shows a number of trajectories in the plane.

The query is expressed as a 2-dimensional interval $[(y_{1q}, y_{2q}), (t_{1q}, t_{2q})]$. The answer is the set of objects that correspond to lines that intersect the query rectangle.

While the space-time representation is quite intuitive, it leads to indexing long lines, a situation that causes significant shortcomings to traditional indexing techniques.

One way is to index the lines using a Spatial Access Method (SAM). Then each line is approximated by a minimum bounding rectangle (MBR) which is then indexed using an R-tree[20] or an R*-tree[8]. However, this approach is problematic because: (i) an MBR assigns to the moving object a much larger area than a line has, (ii) since objects retain their trajectory until being updated, all lines in figure 1 extend to "infinity", i.e. a common ending on the time dimension. Mapping a line segment as a point in four dimensions, by taking the coordinates of the end points, will also not work. Even if we partition the time dimension in time intervals ("sessions") of length ΔT (as in [35]) and index the part of each trajectory that falls in the current session, we still have segments with a common endpoint (the end time of the current session). Another shortcoming is that the SAM can only address queries until the end of the current session.

A different approach is to decompose the data space into disjoint cells and store with each cell the set of lines that intersect it. Indexes that follow this approach are the R+–tree[31], the cell–tree[19] and the PMR–quadtree[30]. The main drawback for these methods is that every line will have many copies; this becomes worse in our environment since lines are large. Storing many copies affects both the update performance (when an object changes its trajectory, its previous route has to be deleted from all cells it was contained), as well as space. [1]

In [23] a method is proposed to index line segments based on the dual transformation. The use of dual transformation to index mobile objects is also proposed in [37]. In the next section we consider this approach in our setting, namely using the dual transformation to index mobile objects.

3.2 The dual space-time representation.

Duality is a powerful and useful transform frequently used in the computational geometry literature; in general it maps a hyper-plane h from R^d to a point in R^d and vice-versa. The duality transform is useful because it allows to formulate a problem in a more intuitive manner.

In our case we can map a line from the *primal* plane (t, y) to a point in the *dual* plane. There is no unique duality transform, but a class of transforms with similar properties. Sometimes one transform is more convenient than another.

Consider a dual plane where one axis represents the slope of an object's trajectory and the other axis its intercept. Thus the line with equation $y(t) = vt + a$ is represented by the point (v, a) in the dual space (this is called the Hough-X transform in [23]). While the values of v are between $-v_{max}$ and v_{max}, the values of the intercept are depended on the current time. If the current time is t_{now} then the range for a is $[-v_{max} \times t_{now}, y_{max} + v_{max} \times t_{now}]$.

The query is transformed in a polygon in the dual space. We can express this polygon using a linear constraint query [18].

Proposition 1 *The one dimensional MOR query is expressed in the dual Hough-X plane as follows:*

- *For* $v > 0$ *the query is :* $Q = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, *where:* $C_1 = v \geq v_{min}$, $C_2 = v \leq v_{max}$, $C_3 = a + t_{2q}v \geq y_{1q}$ *and* $C_4 = a + t_{1q}v \leq y_{2q}$.
- *For* $v < 0$ *the query is :* $Q = D_1 \wedge D_2 \wedge D_3 \wedge D_4$, *where:* $D_1 = v \leq -v_{min}$, $D_2 = v \geq -v_{max}$, $D_3 = a + t_{1q}v \geq y_{1q}$ *and* $D_4 = a + t_{2q}v \leq y_{2q}$.

[1] [35] uses a method based on the PMR-quadtree; their experiments show that even for a small number of mobile objects (50K) the number of copies can become quite large (about 250 copies/object).

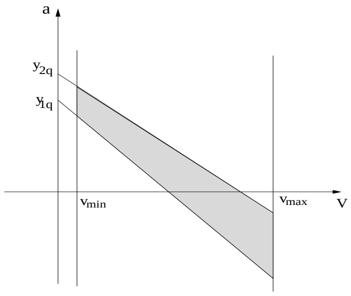


Figure 2: Query in the dual Hough-X plane.

Since the query is different for positive and negative slopes, we can use two structures to store the dual points. It is easy to see that the range of the a's values is now $[-v_{max} \times t_{now}, y_{max} - v_{min} \times t_{now}]$.

However since time is monotonically increasing, the values of the intercept are not bounded. If the value of the maximum speed is significant, the values of the intercept can become very large and this potentially can be a problem (i.e., representing unbounded ranges of real numbers).

To solve this problem we use our assumption that when an object crosses a border it issues an update (i.e. it is deleted or reflected). Combining this assumption with the minimal speed, we can assure that all objects have updated their motion information at least once during the last T_{period} time instants, where $T_{period} = \frac{y_{max}}{v_{min}}$. We can then use two distinct index structures. The first index stores all objects that have issued their last update in the period $[0, T_{period}]$. The second index stores objects that have issued their last update in the interval $[T_{period}, 2T_{period}]$. Each object is stored only once, either in the first or in the second index. Before time T_{period}, all objects are stored in the first index. However, every object that issues an update after T_{period} is deleted from the first index and it is inserted in the second index. The intercept of the first index is computed by using the line $t = 0$ and for the second index using the line $t = T_{period}$. Thus we are sure that the intercept will always have values between 0 and $v_{max} \times T_{period}$. To query the database we use both indices. After time $2T_{period}$ we know that the first index is empty and all objects are stored in the second index, since every object have issued at least one update from T_{period} to $2T_{period}$. At that time we remove the empty index and we initiate a new one with time period $[2T_{period}, 3T_{period}]$. We continue in the same way and every T_{period} time instants we initiate a new index and we remove an empty one. Using this method the intercept is bounded while the performance of the index structures remain asymptotically the same as if we had only one structure.

Another way to represent a line $y = vt + a$, is to write the equation as $t = \frac{1}{v}y - \frac{a}{v}$. Then we can map this line to a point in the dual plane with coordinates $n = \frac{1}{v}$ and the $b = -\frac{a}{v}$ (Hough-Y in [23]). Note that b is the point where the given line intersects the line $y = 0$. Note also that this transform cannot represent horizontal lines (similarly, the Hough-X transform cannot represent vertical lines). However, this is not a problem since our lines have a minimum and a maximum slope.

3.3 Lower Bounds

The dual space-time representation transforms the problem of indexing mobile objects on a line to the problem of *simplex* range searching in two dimensions.

In simplex range searching we are given a set S of 2-dimensional points, and we want to answer efficiently queries of the following form: given a set of linear constraints $a\dot{x} \leq b$, find all the 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. This problem has been extensively studied before in the static, main-memory setting (see for example the excellent survey in [2] and the related work section).

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 ([10]). 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 the 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 [10]. The bound holds for the static case, even if the query region is the intersection of just two hyper-planes. Since ϵ can be arbitrary small, any algorithm that uses linear space for d-dimensional simplex range searching has worst case query time of $O(N^{(d-1)/d} + K)$.

Here we show that a similar bound holds for the input-output complexity of simplex searching. Following the approach in [34] we use the external memory pointer machine as our model of computation. This is a generalization of the pointer machine suitable for analyzing external memory algorithms. In this model, a data structure is modeled as a directed graph $G = (V, E)$, with a source w. Each node of the graph represents a disk block and is therefore allowed to have B data and pointer fields. The points are stored in the nodes of G. Given a query, the algorithm traverses G starting from w, examining the points at the nodes it visits. The algorithm can only visit nodes that are neighbors of already visited nodes (with the exception of the root) and, when it terminates the answer to the query must be contained in the set of visited nodes. The running time of the algorithm is the number of nodes it visits.

Theorem 1 *Simplex reporting in d-dimensions with a query time of $O(n^\delta + k)$ I/O's, where N is the number of points, $n = N/B$, K is the number of the reported points, $k = K/B$, and $0 < \delta \leq 1$, requires $\Omega(n^{d(1-\delta)-\epsilon})$ disc blocks, for any fixed ϵ.*

Proof. (Sketch) To prove the lower bound we need to show that, given δ, there exists a set of N points, and a set of $\Omega(n^{d(1-\delta)-\delta-\epsilon})$ queries such that, each query has $\Theta(Bn^\delta)$ points, and the intersection of any pair of query results is small. To answer a query with $\Theta(Bn^\delta)$ points, the answering algorithm must visit $\Omega(n^\delta)$ nodes. To answer this query in $O(n^\delta)$ I/O's, at least a constant fraction of that many blocks have a constant fraction of their points in the answer of the query. But if the set of the queries has small intersection, it follows that to answer this set of queries in time $O(n^\delta)$ at least $\Theta(n^\delta) \times \Omega(n^{d(1-\delta)-\delta-\epsilon}) = \Omega(n^{d(1-\delta)-\epsilon})$ nodes have to be visited. It remains to show that such a set of queries exist. To do so we simply modify the existing construction by Chazelle and Rosenberg [10] by replacing each point in their point set by B copies. $\square$

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

3.4 An (Almost) Optimal Solution

Matousek ([27]) gave an almost optimal algorithm for simplex range searching, given a static set of points. This main memory algorithm is based on the idea of simplicial partitions.

We briefly describe this approach here. For a set S of N points, a simplicial partition of S is a set $\{(S_1, \Delta_1), \ldots (S_r, \Delta_r)\}$ where $\{S_1, \ldots, S_r\}$ is a partitioning of S, and Δ_i is a triangle that contains all the points in S_i. If $\max_i |S_i| < 2 \min_i |S_i|$, we say that the partition is balanced. Matousek ([27]) 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 it is inside the query region, outside the query region, or 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, and in the third case we recurse on the triangle. The number of triangles that the query can cross is bounded however, since each line crosses at most $O(|S|^{\frac{1}{4}})$ triangles at the root. The query time is $O(N^{\frac{1}{2}+\epsilon} + K)$, with the constant factor depending on the choice of ϵ.

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

3.5 Improving the average query time.

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 chose a small ϵ. In this section we present two different approaches that are designed to improve the average query time.

3.5.1 Using Point Access Methods

There is a large number of access methods that have been proposed to index point data[17]. All these structures were designed to address *orthogonal* range queries, eg. a query expressed as a multidimensional hyperrectangle. However, most of them can be easily modified to address non-orthogonal queries like simplex queries.

Recently, Goldstein at 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 hyper-rectangle overlap. As mentioned in [18] this method is not only applicable to the R-tree family, but to other access methods as well.

We use this approach to answer the one dimensional MOR query in the dual Hough-X space (Figure 2). However it is not clear what structure would be more suitable here, given that the distribution of points in the dual space is highly skewed. We argue that an index structure based on kd-trees (like the LSD-tree [21] and the hB^{Π}-tree [16]) is more suitable than a method based on R-trees. The reason is that since R-trees try to cluster data points into squarish regions [24], they will split using only one dimension (the intercept). On the other hand a *kd*-tree based method will use both dimensions to split (see Figure 3). Thus it is expected to have better performance for the MOR query.

3.5.2 A Query Approximation Algorithm.

A different approach is based on a *query approximation* idea using the Hough-Y dual plane. In general, the b coordinate can be computed at different horizontal ($y = y_r$) lines. The query region is described by the intersection of two half-plane queries (Figure 4). The one line intersects the line $n = \frac{1}{v_{max}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and the line $n = \frac{1}{v_{min}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, \frac{1}{v_{min}})$. Similarly the other line that defines the query intersects the horizontal lines at $(t_{2q} - \frac{y_{1q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and $(t_{2q} - \frac{y_{1q}-y_r}{v_{min}}, \frac{1}{v_{min}})$.

Since access methods are more efficient for rectangle queries, suppose that we approximate the simplex query with a rectangular one. In Figure 4 the query 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 $E = E_1 + E_2$ which is computed as:

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

We are interested in minimizing E since it represents a measure of the extra I/O's that an access method will have to perform for solving an one dimensional MOR query. E is based on both y_r (i.e. where the b coordinate is computed) and the query interval (y_{1q}, y_{2q}) which is unknown. Hence, we propose to keep c indices (where c is a small constant) at equidistant y_r's. All c indices contain the same information about the objects, but use different y_r's. The i-th index stores the b coordinates of the data points using $y = \frac{y_{max}}{c} \times i$, $i = 0, .., c-1$. 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 the index(es) that minimize E. Since all 2-dimensional approximate queries have the same rectangle side $(\frac{1}{v_{max}}, \frac{1}{v_{min}})$ (Figure 4) the rectangle range search is equivalent to a simple range search on the b coordinate axis. Thus each of the c "observation" indices can simply be a B+-tree [13].

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:

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

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[5] which will answer a subterrain query optimally (i.e., $E = 0$). As a result, the original query can be answered with bounded E. Thus we have the following lemma:

Lemma 1 *The one dimensional MOR query can be answered in time $O(\log_B n + (K+K')/B)$, where K' is the approximation error. The space used is $O(cn)$ where c is a small constant, and the update is $O(c \log_B n)$.*

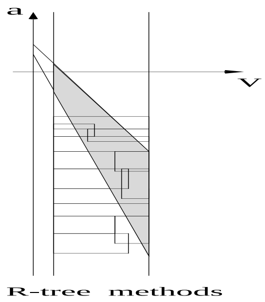
Note that assuming that the points are distributed uniformly over the b-axis, then the approximation error is bounded by $1/c$, eg. $K' = O(1/c)$.

3.6 Achieving Logarithmic Query Time

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

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

Formally, the problem we are considering in this section is the following: given a set of objects that are moving on a line, and a time limit T, find all the objects that lie in the segment $[y_l, y_r]$ at time t_q (where $t_0 \leq t_q \leq t_0 + T$). Equivalently, this is a standard one



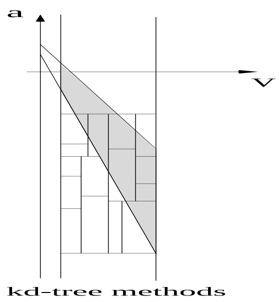


Figure 3: Data regions for R-tree like and *kd*-tree like methods .

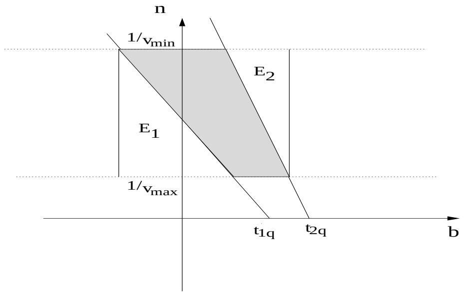


Figure 4: Query in the dual Hough-Y plane.

dimensional MOR query where $t_{1_q} = t_{2_q}$. We will call it an one dimensional MOR1 query.

Our method is to find all the times when an object overtakes another. These events correspond to line segment crosses 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 2 *If we have the relative ordering of all the objects at time t_q, the position of the objects at time t_c that corresponds to the closest crossing event before t_q, and the speed of the objects, we can find the objects that are in $[y_l, y_r]$ in $O(\log_2 N + K)$ time.*

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

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

The following lemma finds all the crossings of points efficiently.

Lemma 3 *We can find all the crossings of objects in time $O(N \log_2 N + M \log_2 M)$, where M is the number of crosses in the time period $[0, T]$.*

Proof. Let $\{p_1, \ldots, p_N\}$ be the ordering of the N objects at time 0. At time T, the position of object i is $y_i + v_i \times T$. To find the ordering of the objects at time T we have to sort their positions. Let $\{p_{t(1)}, \ldots, p_{t(N)}\}$ be the ordering of the same N objects at time T. Then objects i and j cross if and only if $t(j) < t(i)$.

Keep the objects in a linked list, in the same order they were at time $t = 0$. Scan the sorted list of objects at time T. Find object $p_{t(1)}$ in the list. This object crosses all the objects ahead of it in the list. After reporting these crosses, we remove it from the list, and repeat this process with the next point. This procedure reports all the crossings in $O(N + M)$ time. After all the crossings are reported we can find when each occurs and sort them on their time attributes. □

These M crosses define M ordered lists of the N objects. Each two consecutive lists differ in exactly two positions, the positions that correspond to the objects that cross. The total sum of the differences between consecutive lists is therefore $O(M)$. In the next lemma we show how we can efficiently store and search these lists in external memory.

Lemma 4 *We can store the $O(M)$ ordered lists of N objects in $O(n+m)$ blocks and perform a search on any list in $O(\log_B(n+m))$ I/O's, where $n = \frac{N}{B}$ and $m = \frac{M}{B}$.*

Proof. Let $L(t)$ be the list of objects at time t. Consider $CS = t_1$,..., 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 $L(t_1)$, $L(t_2)$,..., $L(t_M)$, where $L(t_{i+1})$ is produced from $L(t_i)$ by applying the crossing that occurred at t_{i+1} (i=0,...,M-1, and t_0=0).

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

To reduce the space to $O(N+M)$ we must take advantage of the fact that subsequent lists do not defer much. A main-memory solution to this problem appears in [12]. 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 tree $B(0)$ that corresponds to the initial list $L(0)$. Tree $B(0)$ uses $O(n)$ nodes where each node can hold B entries. An entry is now a record: (*position, occupant, pointer, t*), where *position* corresponds to a position in the list, *occupant* contains the element at that position, *pointer* points to a child node and t corresponds to the time this element was at that position, in this case t=0.

Conceptually, each B-tree node is permanently assigned B positions and is responsible for storing the occupants of these positions. Consider the evolution of such a node s through trees $B(0)$,$B(t_1)$,$B(t_2)$,...,$B(t_M)$. 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 node 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 the change. To achieve fast access to $s(t)$ we do not allow the "log" to get too large. Every $O(B)$ changes (in practice when the log fills one or two pages) we store a new, current copy of s. If we consider the history of node s independently, we can have an auxiliary array with records (*time*, *pointer*) that point to the various copies of node s. Locating the appropriate node $s(t)$ takes $O(\log_B m)$ time (first find the record in the auxiliary array with the largest timestamp that is less or equal to t and then we access the appropriate copy of s and probably a (constant) number of "log" pages). The space remains $O(n+m)$ since every new node copy is amortized over the $O(B)$ changes in the "log".

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

The following theorem follows from the previous lemmas:

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

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

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

4 Indexing in two dimensions

In this section we consider the problem of mobile objects in the plane. Again we consider only "moving" objects, namely objects with a speed between v_{min} and v_{max}. We assume that objects move in the (x, y) plane inside the finite terrain $[(0, x_{max})(0, y_{max})]$. The initial location of the object o_i is (x_{i_0}, y_{i_0}) and its velocity is a vector $\vec{v} = (v_x, v_y)$.

We distinguish two important cases. The first considers objects moving in the plane but their movement is restricted on using a given collection of routes (roads) on the finite terrain. Due to its restriction, we call this case the 1.5-dimensional problem. There is a strong motivation for such an environment; for the applications we have in mind, objects (cars, airplanes etc.) move on a network of specific routes (highways, airways). In the second case the objects move anywhere in the plane.

4.1 The 1.5-dimensional problem

The 1.5-dimensional problem can be reduced to a number of 1-dimensional queries. In particular, we propose representing each predefined route as a sequence of connected (straight) line segments. The positions of these line segments on the terrain are indexed by a standard SAM. (Maintaining this SAM does not introduce a large overhead since for most practical applications: (a) the number of routes is much smaller than the number of objects moving on them, (b) each route can be approximated by a small number of straight lines, and, (c) new routes are added rather infrequently.) Indexing the objects moving on a given route is an 1-dimensional model and will use techniques from the previous section.

Given a two dimensional MOR query, the above SAM identifies the intersection of the routes with the query's spatial predicate (the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$). Since each route is modeled as a sequence of line segments, the intersection of the route and the query's spatial predicate is also a set of line segments, possibly disconnected. Each such intersection corresponds to the spatial predicate of an 1-dimensional query for this route. In this setting we assume that when routes intersect, objects remain in the route previously traveled (otherwise an update is issued).

4.2 The 2-dimensional problem

The full 2-dimensional problem (i.e., allowing objects to move anywhere on the finite terrain) is more difficult. As with the 1-dimensional case, we discuss different representations of the problem and we propose methods to address the two dimensional MOR query.

In the space-time representation the trajectories of the mobile objects are lines in the space. The lines can be computed by the motion informations of each object. The two dimensional MOR query is expressed as a cube in the 3-dimensional (x, y, t) space and the answer is the set of objects with lines that cross the query cube.

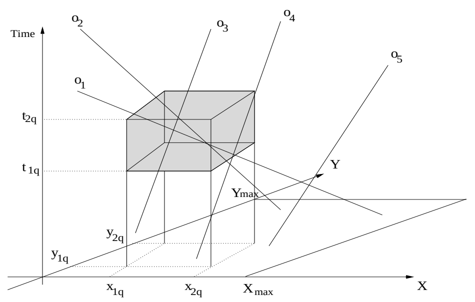


Figure 5: Trajectories and query in (x, y, t) plane.

Algorithms that are applied directly to the time-space representation do not work well in one-dimension, so the performance is likely to be even worse in two dimensions. Unfortunately we cannot use directly the dual transformations of the previous section, since these transforms map a hyper-plane in the space into a point and vice-versa, where here we have lines. We point out that the problems with lines in the space are much harder than lines in the plane. The reason is that a line in space has 4 degrees of freedom and therefore taking the dual we jump to a 4-dimensional space. To get the dual we project the lines on the (x, t) and (y, t) planes and then take the duals for the two lines on these planes. Thus now a line can be represented by the 4-dimensional point (v_x, a_x, v_y, a_y), where the v_x and v_y are the slopes of the lines on the (x, t) and (y, t) planes and the a_x and a_y are the intercepts respectively.

The two dimensional MOR query is mapped to a simplex query in the dual space. This query is the intersection of four 3-d hyper-planes and the projection of the query to (t, x) and to (t, y) planes are wedges, as in the 1-dimensional case. Thus we can use a 4-dimensional partition tree (section 3.4) and answer the MOR query in $O(n^{0.75+\epsilon} + k)$ I/O's that almost matches the lower bound for four dimensions. A simple approach to solve the 4-dimensional problem is to use an index based on the $kd - tree$. An alternative approach is to decompose the motion of the object into two independent motions, one in the x-axis and the other in the y-axis. For each axis we can use the methods for the 1-dimensional case and answer two 1-dimensional MOR queries. We must then take the intersection of the two answers to find the answer to the initial query. This method allows us to use the algorithms for the 1-dimensional case.

5 A Performance Study

We present initial results for the one dimensional MOR query, comparing our query approximation approach, the *kd*-tree method and a traditional R-tree based approach. First we describe the way experimental data is generated. At time $t = 0$ we generated the initial locations of N mobile objects uniformly distributed on the terrain $[0, 1000]$. We varied N from $100k$ to $500k$. The speeds were generated uniformly from $v_{min} = 0.16$ to $v_{max} = 1.66$ and the direction randomly positive or negative.[2] Then objects start moving. When an object reaches a border simply it changes its direction. At each time instant we choose 200 objects randomly and we randomly change their speed and/or direction. We generate 10 different time instants that represent the times when queries are executed. At each such time instant we execute 200 random queries, where the length of the y-range is chosen uniformly between 0 and $YQMAX$ and the length of the time range between 0 and TW. We

[2] Note that 0.16 miles/min is equal to 10 miles/hour and 1.66 miles/min is equal to 100 miles/hour.

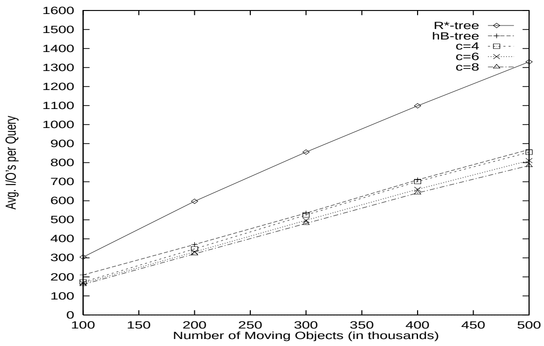


Figure 6: Query Performance for 10% Queries.

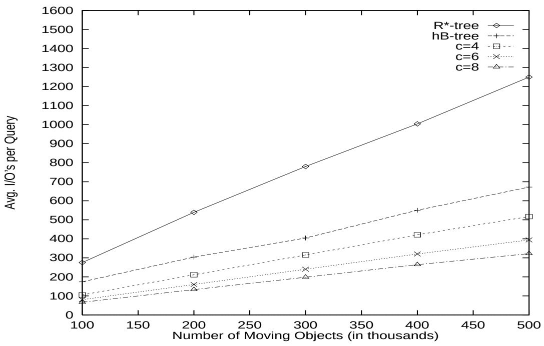


Figure 7: Query Performance for 1% Queries.

actually generated two sets of queries. One set of large queries with $YQMAX = 150$ and $TW = 60$ and one set of small queries with $YQMAX = 10$ and $TW = 20$. The first set of queries has average cardinality almost 10% and the second one close to 1%. We run this scenario using a particular access method for 2000 time instants.

To verify that indexing mobile objects as line segments is not efficient, we stored the trajectories in an R*-tree. We fixed the page size to 4096 bytes. To represent a line segment in an R*-tree we used four 4-byte numbers (the two end points) and one more number as a pointer to the real object, resulting in a page capacity of $B = 204$. For the B+-tree we used one 4-byte number to represent the b-coordinate, one number for the speed and another one for the pointer, so the page capacity was $B = 341$. We index each record using only the b-coordinate but using the speed of each object we can identify the objects that correspond to the real answer and report only these objects. The same page capacity used and for the hB^{Π}-tree.However, each hB^{Π}-tree page reserves some space for internal structural data. We consider a simple buffering scheme for the results we present here. For each tree we buffer the path from the root to a leaf node, thus the buffer size is only 3 or 4 pages. For the queries we always clear the buffer pool before we run a query. An update is performed when the motion information of an object changes.

In Figure 6 we present the results for the average number of I/O's per query for 10% queries and in figure 7 for 1% queries. The approximation method used $c = 4, 6$ and 8 B+-trees. As anticipated, the line segments method with R*-trees has the worst performance. Also, the approximation method outperforms the hB^{Π}-tree for small queries and it is slightly better for large queries.

In Figures 8,9 we plot the space consumption and the average number of I/O's per update respectively. We did not report the update performance for the R*-tree method because it was very high (more than 90 I/O's per update). The update and space performance of the hB^{Π}-tree is better than the other methods since its objects are stored only once and better clustered than the R*-tree. The update performance of the hB^{Π}-tree and the approximation approach remain constant for different number of mobile objects. The space of all methods is linear to the number of objects. The approximation approach uses more space due to the use of c observation indices. There is a tradeoff between c and the query/update performance.

6 Related Work

The problem of indexing mobile objects is novel; we are not aware of any other related work except [35] where a method to index mobile objects based on the PMR-quadtree is presented. However as we mentioned earlier, this approach has large space and update overhead.

Mobility in a geographic system is addressed in [32] where the aim is to map close points in space to adjacent disks so as collision detection queries are optimized.

The queries we examine have also a temporal component. There has been a lot of research in temporal indexing [29], however it has focused on queries about the past and not the future as in our case.

Representing the trajectories as line segments in two and three dimensions, also relates to spatial indexing [17]. [22] presents a qualitative comparison of three spatial access methods for a line segments database is presented. In particular they consider the R*-tree, the R+-tree and the PMR-quadtree. The result is that all these methods are comparable and no one seems to be superior than the others.

A method to index line segments on the plane is presented in [23]. A line segment is represented by the slope and the intercept of the line obtained by extending the line segment and by the range of the projection of the segment to one axis. Using this mapping, a line segment on the plane is mapped to a vertical line segment in three dimensions. Then a standard spatial access

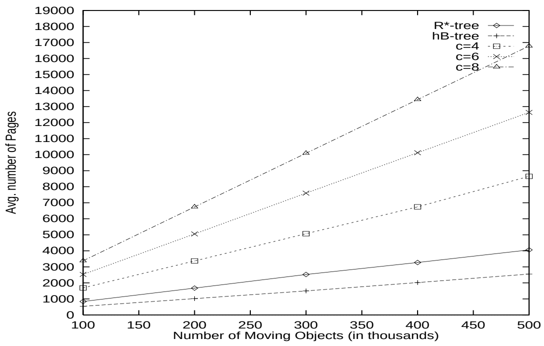


Figure 8: Space Consumption.

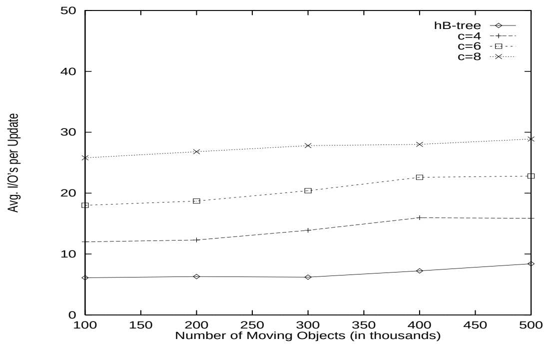


Figure 9: Update Performance.

method can be used to index the new segments. It is shown analytically and experimentally that the queries in the transformation space have better selectivity that in the original space.

An interesting approach to index constraint databases is presented by Bertino et al. in [9]. In particular they address the problem of indexing conjunction of linear constraints with two variables, in order to answer ALL and EXIST queries (variations of the half-plane query). They use the dual transformation and they reduce the problem to a point location problem. Then, if the line that defines the query has slope from a predefined set of slopes, an optimal solution can be derived using the external memory Interval tree[5]. Other works on indexing constraint databases include [5, 25]. These approaches reduce the problem of indexing constraint to a dynamic interval management problem or to a special case of two-dimensional range searching, and therefore are not applicable to our problem.

The issue of mobility and maintenance of a number of configuration functions among continuously moving objects has been addressed by Basch et al. in [6]. Such functions are the convex hull, the closest pair and the minimum spanning tree. They propose a framework to transform a static data structure into a *kinetic* data structure (KDS) that maintains an attribute of interest for a set of mobile objects and they give a number of criteria for the quality of such structures. The key structure is an event queue that contains events corresponding to times where the value of the configuration function (may) change. This event queue is the interface between the data structure and the mobile objects. All these structures are main memory data structures.

7 Conclusions and Future Work

Indexing mobile objects is a novel problem motivated by real life applications. We study the one and two dimensional versions of the problem. For the one dimensional case, we give a dynamic, external memory algorithm with guaranteed worst case performance and linear space. We also give a practical approximation algorithm also in the dynamic, external memory setting, which has linear space and expected logarithmic query time. Finally we give an algorithm with guaranteed logarithmic query time for a restricted version of the problem. We also extend some of our results into two dimensions. First we consider the case where objects move in 2–dimensional networks of 1–dimensional routes. In this case we can effectively apply our 1–dimensional algorithms. We also consider objects that move on a plane, and we discuss extensions of our techniques to two dimensions.

Future work includes a variety of interesting problems. In addition to performing a more complete performance study (using various data distributions) we plan to address restricted versions of the 2–dimensional problem using realistic assumptions. One idea is to cluster similarly moving objects into representative clusters. If query response is time critical, main–memory database techniques need to be involved. We are currently studying the problem of indexing mobile objects with probabilistic route choices. A generalization of the 1.5–dimensional problem is when the terrain is subdivided into areas with various speed limits (or terrain abnormalities that limit movement according to direction). Other interesting queries are near-neighbor queries and joins among relations of mobile objects. Some applications may require keeping the history of mobile objects (for traffic analysis etc.); then the indices presented need to support historical queries. This probably requires making the presented structures partially persistent [7, 26]. While in this paper we restricted the object movement to simple (linear) functions, it is a first step at examining ways to index more complex functions.

8 Acknowledgement

The problem of indexing moving objects was proposed to us by Ouri Wolfson at the 1997 Dagstuhl Seminar on

Temporal Databases. The authors would like to thank O. Wolfson, C. Faloutsos and V.S. Subrahmanian for many helpful discussions.

References

[1] P.K. Agarwal, L. Arge, J. Erickson, P. Franciosa and J.S. Vitter. Efficient Searching with Linear Constraints In *Proc. 17th ACM PODS Symposium on Principles of Database Systems*,pp. 169-178 1998.

[2] P.K. Agarwal, and J. Erickson. Geometric range searching and its relatives. In *Discrete and Computational Geometry: Ten Years Latter.*, (B. Chazelle, E. Goodman, and R. Pollack eds.), American Math. Society, Providence, 1998.

[3] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. In *Communications of the ACM*, 31(9):1116-1127, 1988.

[4] ArcView GIS. ArcView Tracking Analyst. 1998.

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

[6] J. Basch, L. Guibas and J. Hershberger. Data Structures for Mobile Data. In Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, Louisiana, 1997.

[7] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, 5(4):264-275,1996.

[8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method For Points and Rectangles. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, May 1990.

[9] E. Bertino, B. Catania and B. Shidlovsky. Towards optimal two-dimensional indexing for constraint databases. *Information Processing Letters*, 64(1997):1-8.

[10] B. Chazelle and B. Rosenberg. Lower Bounds on the Complexity of Simplex Range Reporting on a Pointer Machine. *Proc. 19th Intern. Colloquium on Automata, Languages and Programming. LNCS,* Vol. 693, Springer-Verlang, Berlin, 1992.

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

[12] R. Cole. Searching and Storing Similar Lists. *Journal of Algorithms*, 7(2):202-220, 1986.

[13] D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.

[14] J. Driscoll, N. Sarnak, D. Sleator and R.E. Tarjan. Making Data Structures Persistent. In *Proc. of the 18th Annual ACM Symp. on Theory of Computing*, Berkeley, CA, 1986.

[15] M. Erwig, R.H. Guting, M. Schneider and M. Vazirgianis. Spatio-temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. In *Proc. of ACM GIS Symposium '98.*

[16] G. Evangelidis, D. Lomet, and B. Salzberg. The hB^{Π}-tree: A Modified hB-tree Supporting Concurrency, Recovery and Node Consolidation. In *Proc. 21st Intern. Conf. on Very Large Data Bases*, Zurich, September 1995.

[17] V. Gaede and O. Gunther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170-231, 1998.

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

[19] O. Gunther. The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases. In *Proc. Fifth IEEE International Conference on Data Engineering*, Los Angeles, CA, USA, February 1989.

[20] A. Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD Intern. Conf. on Management of Data*, pages 47–57, June 1984.

[21] A. Henrich, H.-W. Six, P. Widmayer. The LSD–tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In *Proc. 15th Intern. Conf. on Very Large Data Bases*, pages 45–53, Amsterdam, August 1989.

[22] E.G. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Linear Segment Databases. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 205-214, San Diego, June 1992.

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

[24] I. Kamel and C. Faloutsos. On Packing R–trees. In *Proc. Second Int. Conference on Information and Knowledge Management (CIKM)*, Washington, DC, Nov. 1-5, 1993.

[25] P. Kanellakis, S. Ramaswamy, D. Vengroff and J. Vitter. Indexing for Data Models with Constraint and Classes. In *Proc. 12th ACM SIFACT-SIGMOD-SIGART Symposium on Principles of Database Systems*,pages 233-243 , Washington, D.C, 1993.

[26] A. Kumar, V.J. Tsotras and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Trans. on Knowledge and Data Engineering*, 10(1):1-20, 1998.

[27] J. Matousek. Efficient Partition Trees. *Discrete and Computational Geometry*, 8 (1992), 432-448.

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

[29] B. Salzberg and V.J. Tsotras. A Comparison of Access Methods for Time-Evolving Data. *ACM Computing Surveys*, March 1999.

[30] H. Samet. *The Design and Analysis of Spatial Data Structures.*, Addison-Wesley, Reading, MA, 1990.

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

[32] S. Shekhar and T.A. Yang. Motion in a Geographical Database System In *2nd International Symposium on Advances in Spatial Databases*, pages 339–357, Zürich, Switzerland, August 1991.

[33] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Moving Objects. In *Proc. 13th IEEE International Conference on Data Engineering*, pages 422-432, Birmingham, U.K, April 1997.

[34] S. Subramanian and S. Ramaswamy. 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, NY, USA, 1995.

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

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

[37] O. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving Objects Databases: Issues and Solutions In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management.* Capri, Italy, July 1998.