Advances in Indexing for Mobile Objects

Pankaj K. Agarwal
Department of Computer Science
Duke University
Durham, NC 27708
pankaj@cs.duke.edu *

Cecilia M. Procopiuc
AT&T Research Labs
180 Park Ave., Box 971
Florham Park, NJ 07932
magda@research.att.com

1 Introduction

With the rapid advances in positioning systems, such as GPS, ad-hoc networks, and wireless communication, it is becoming increasingly feasible to track and record the changing position of continuously moving objects. Several areas such as digital battlefields, air-traffic control, mobile communication, navigation system, and geographic information systems need the capability to index moving objects, so that various queries on them can be answered in real time. The queries might relate either to the current configuration of objects, to a past configuration, or to a future configuration — in the last case, we are asked to predict the behavior based on the current information. However, most existing database systems assume that the data is constant unless it is explicitly modified. Such systems are not suitable for representing, storing, and querying continuously moving objects, since, unless the database is continuously updated (at considerable time and resources expense), a query output will be obsolete. The need for new database technology that can support kinetic applications has given rise to a large number of questions, from defining suitable models and query languages [18, 28] to developing new engines capable of indexing and querying moving objects efficiently.

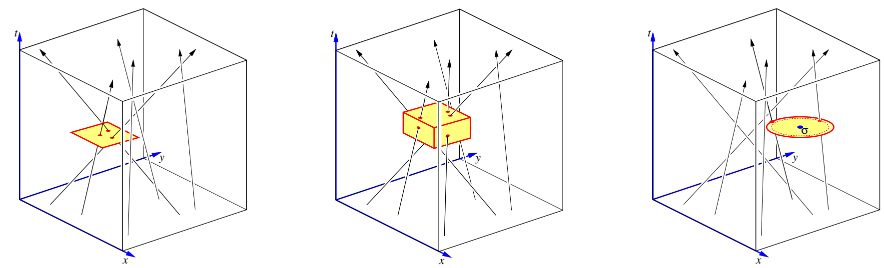
In this paper we discuss the latter line of research, and we review recent advances and remaining challenges. Since moving objects can be viewed as points moving along algebraic curves, the problem has a natural representation in the context of computational geometry. Hence, many results we present here draw heavily on the rich literature of geometric indexing structures, kinetic data structures, and geometric approximation techniques.

Data representation. Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in $\mathbb{R}^2$, each moving continuously. Let $p_i(t) = (x_i(t), y_i(t))$ denote the position of p_i at time t, and let $S(t) = \{p_1(t), \ldots, p_n(t)\}$. Let $\bar{p}_i = \bigcup_t (p_i(t), t)$ denote the trajectory of p_i over time. We say that the motion of S is *linear* if each x_i, y_i is a *linear* function, i.e., each $\bar{p}_i$ is a line in $\mathbb{R}^3$, and the motion of S is *algebraic* if each x_i, y_i is a polynomial of bounded degree. For most of the results discussed in this paper, we assume the motion of S to be *piecewise linear*, i.e., each $\bar{p}_i$ is a polygonal chain in $\mathbb{R}^3$. The trajectories of points can change any time. We assume that the database system is modified whenever these values change. We will use the term *now* to mean the current time.

Queries. Any indexing structure should be able to answer queries based on the current, past, or future positions of the objects. In the case of future positions, the answer is a *prediction* of the query output based on the current

Copyright 2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Research by P.A. is partially supported by NSF under grants CCR-00-86013 EIA-98-70724, EIA-9972879, EIA-01-31905, and CCR-97-32787, and by a grant from the U.S.–Israel Binational Science Foundation.



(a) (b) (c)

Figure 1: Instances of Q1, Q2, and Q3 queries, respectively, with time as the z-axis.

trajectories of the points. In this paper we focus our attention on range-searching and nearest-neighbor queries, which are defined below.

Q1. Given an axis-aligned rectangle R in the xy-plane and a time value t_q, report all points of S that lie inside R at time t_q, i.e., report $S(t_q) \cap R$; see Figure 1 (a).

Q2. Given a rectangle R and two time values $t_1 \leq t_2$, report all points of S that lie inside R at any time between t_1 and t_2, i.e., report $\bigcup_{t=t_1}^{t_2}(S(t) \cap R)$; see Figure 1 (b).

Q3. Given a query point $\sigma \in \mathbb{R}^2$ and a time value t_q, a *nearest-neighbor* query requires reporting the nearest neighbor of σ in $S(t_q)$; Figure 1 (c).

Model of computation. In order to be useful in practice, any proposed method must scale well over large data sets. Since the data resides on disk, the focus of these approaches is to minimize the data transfer between disk and main memory, which is likely to be the bottleneck for any system. Most of the results in this paper will be discussed in the *parallel disk model* introduced by Vitter and Shriver [31], which closely models the way computers handle data. This model assumes that each disk access transmits a contiguous block of B units of data in a single *input/output operation* or *I/O*. The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in units of disk blocks) and the number of I/Os required to answer a query. Since we are also interested in solutions that are *output sensitive*, our query I/O bounds are expressed not only in terms of N, the number of points in S, but also in terms of K, the number of points reported by the query. Note that we need at least $\lceil N/B \rceil$ blocks to store all N points, and at least $\lceil K/B \rceil$ blocks to store the output from a range query. We refer to these bounds as "linear" and introduce the notation $n = \lceil N/B \rceil$ and $k = \lceil K/B \rceil$.

General approaches. Two general approaches have been proposed to index moving points. The first approach, which we refer to as a *time-oblivious* approach, regards time as a new dimension and indexes the trajectories $\bar{p}_i$ of input points p_i. One can either index the trajectories themselves using various techniques [24, 29], or one can work in a parametric space [1, 21], map each trajectory to a point in this space, and build an index on these points. An advantage of a time-oblivious scheme is that the index is updated only if the trajectory of a point changes or when a point is inserted into or deleted from the index. However, updating a trajectory can be quite expensive. Since this approach indexes either curves in $\mathbb{R}^3$ or points in higher dimensions, the query time tends to be large.

The second approach, which we refer to as *kinetic data structures*, builds a dynamic index on the moving points. Roughly speaking, at any time it maintains an index on the current configuration of the points. As the points move, the index is updated. The main observation is that although the points are moving continuously, the index is updated only at discrete time instances. This approach leads to fast query time, but the main disadvantage is that it can answer a query only at the current configuration. It can be extended to handle queries arriving in chronological order, i.e., the time stamps of queries are in nondecreasing order. Using the so-called persistence technique [15, 27], the index can answer queries on the near future or the near past configurations of points. Another weakness of this approach is the cost associated with updating the index as the points move.

One can combine the two approaches to take advantage of each one. In many applications, such as air-traffic control, it is more crucial to answer the queries related to the near future configurations (i.e., $|t_q - now|$ is small) efficiently and accurately. The idea is to still use time as an additional dimension, but optimize the index for the time close to now and store only an approximate representation of the point configurations far away from now. The index is updated periodically to maintain the above invariant. This approach, which we call *time-responsive* indexing, has been used in a number of recent papers [1, 4, 25, 32].

The paper is organized as follows. In Section 2 we review a few geometric techniques that will be useful for the indexes discussed here. For simplicity, we first discuss in Section 3 the above approaches for points moving on the x-axis. Section 4 reviews the known techniques for range searching amid points moving in the plane. Section 5 discusses nearest-neighbor queries on moving points, and we conclude in Section 6 by mentioning a few open problems.

2 Geometric Techniques

In this section we discuss a few geometric techniques, which lie at the heart of many of the indexing schemes discussed in subsequent sections.

Duality. Duality is a popular and powerful technique used in geometric algorithms [14] and has been used in indexing moving points [1, 21, 28]; it maps each point in $\mathbb{R}^2$ to a line in $\mathbb{R}^2$ and vice-versa. We use the following duality transform (a few other variants also appear in the literature): The dual of a point $(a, b) \in \mathbb{R}^2$ is the line $x_2 = ax_1 - b$, and the dual of a line $x_2 = \alpha x_1 + \beta$ is the point $(\alpha, -\beta)$. Let σ^* denote the dual of an object (point or line) σ, and for any set of objects Σ, let Σ^* denote the set of dual objects $\{\sigma^* \mid \sigma \in \Sigma\}$. An essential property of this transformation is that a point p is above (resp., below, on) a line h if and only if the dual point h^* is above (resp., below, on) the dual line p^*. The dual of a strip σ is a vertical line segment σ^*, in the sense that a point p lies inside σ if and only if the dual line p^* intersects σ^*. See Figure 2.

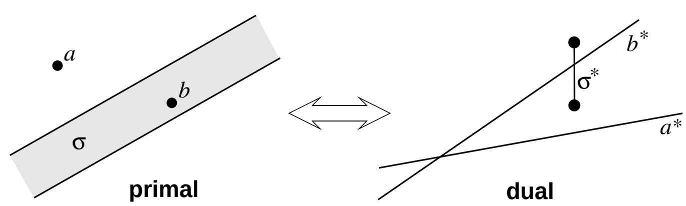


Figure 2: The duals of two points and a strip are two lines and a vertical line segment.

Partition trees. Partition trees, originally proposed by Willard [33], are one of the most commonly used internal memory data structures for geometric-searching problems [5, 23]. Recently, partition trees have been extended to the external memory model [2]. Let S be a set of N points in $\mathbb{R}^2$. A *simplicial partition* of S is a set of pairs $\Pi = \{(S_1, \triangle_1), (S_2, \triangle_2), \ldots, (S_r, \triangle_r)\}$, where the S_i's are disjoint subsets of S, and each $\triangle_i$ is a triangle containing the points in the corresponding subset S_i. A point of S may lie in many triangles, but it belongs to only one subset S_i. The size of Π, here denoted r, is the number of subset-triangle pairs. A simplicial partition is *balanced* if each subset S_i contains between N/r and $2N/r$ points. The *crossing number* of a simplicial partition is the maximum number of triangles crossed by a single line. By extending Matoušek's algorithm to the external memory model, Agarwal *et al.* [2] showed that for $r = O(B)$, a balanced simplicial partition Π for S of size r and crossing number $O(\sqrt{r})$ can be constructed in $O(nr)$ expected I/Os.

Using simplicial partitions, a partition tree T on S can be constructed as follows: Each node v in T is associated with a subset $S_v \subseteq S$ of points and a triangle $\triangle_v$. For the root of T, we have $S_{\text{root}} = S$ and $\triangle_{\text{root}} = \mathbb{R}^2$. Let $N_v = |S_v|$ and $n_v = \lceil N_v/B \rceil$. We construct the subtree rooted at node v as follows. If $N_v \leq B$, then v is a leaf and we store all points of S_v in a single block. Otherwise, v is an internal node of degree $r_v = O(B)$. We compute a balanced simplicial partition $\Pi_v = \{(S_1, \triangle_1), \ldots, (S_{r_v}, \triangle_{r_v})\}$ for S_v with crossing number $O(\sqrt{r_v})$ and then recursively construct a partition tree T_i for each subset S_i. T uses $O(n)$ disk blocks and can be constructed in $O(N \log_B n)$ expected I/Os. For a query triangle σ, all points in $S \cap \sigma$ can be reported as follows: we visit T in a top down fashion. Suppose we are at a node v. If v is a leaf, we report all points of S_v that lie inside σ. Otherwise, we test each triangle $\triangle_i$ of Π_v. If $\triangle_i$ lies completely outside σ, we ignore it; if $\triangle_i$ lies completely inside σ, we report all points in S_i by traversing the ith subtree of v; finally, if σ crosses $\triangle_i$, we recursively visit the ith child of v. As shown in [1, 2], the query takes $O(n^{1/2+\varepsilon} + k)$ I/Os. Moreover, a point can be inserted into or deleted from the tree in $O(\log_B^2 n)$ expected I/Os.

Kinetic data structures. The *kinetic data structure* (KDS) framework, originally proposed by Basch *et al.* [12], stores only a "combinatorial snapshot" of the moving points at any time. Although the points are moving continuously, the data structure itself only depends on certain combinatorial properties (such as two points in 1D coincide) and changes only at discrete instants, called *events*. When an event occurs, we perform a *kinetic update* on the data structure. Since we know how the points move, we can predict when any event will occur. The evolution of the data structure is driven by a global *event queue*, which is a priority queue containing all future events. A good KDS is one that strikes a proper balance in maintaining a certificate set. On the one hand, the certificate set should be as stable as possible and not undergo drastic or unnecessarily frequent changes as the objects move; furthermore, it should be repairable by local operations when certificates fail. At the same time, the certificate set has to enable a fast computation of the attribute of interest in order to be useful. See [16] for an overview of KDS.

Persistence. Many database applications ask for updating the current index while querying both the current and earlier versions of the index. The *persistence* technique, proposed by Sarnak and Tarjan [27] and generalized by Driscoll *et al.* [15], provides a way to adapt KDS so that earlier versions of the index can be queried. The persistence technique basically stores the "difference" between two consecutive version of the index, by keeping track of the time interval during which an element is really present in the index. A B-tree can be made persistent as follows [13, 30]. Roughly speaking, each data element is augmented with a *life span* consisting of the time at which the element was inserted and (possibly) the time at which it was deleted. Similarly, each node in the B-tree is also augmented with a life span. We say that an element or a node is *alive* during its life span. Apart from the normal B-tree constraint on the number of elements in a node, we also maintain that a node contains $\Theta(B)$ alive elements (or children) in its life span. This means that for a given time t, the nodes with life span containing t make up a B-tree on the elements alive at that time. An insertion or deletion in a persistent B-tree is performed almost like a normal insertion; we omit the details from here. Becker *et al.* [13] show that each update operation takes $O(\log_B n)$ I/Os, and that ν update operations require $O(\nu/B)$ additional disk blocks.

3 One-Dimensional Indexing

In this section we describe the three general techniques discussed in the introduction for indexing a set S of points moving in $\mathbb{R}$. For simplicity, we assume that the motion of S is linear, i.e., $p_i(t) = a_i t + b_i$, where $a_i, b_i \in \mathbb{R}$, and $\bar{p}_i$ is a line in the xt-plane. Let $L = \{\bar{p}_i \mid p_i \in S\}$.

Time-oblivious indexing. Tayeb *et al.* [29] proposed an index based on the so-called PMR-quadtrees [26], in which a square (associated with a node of the quadtree) is split into four subsquares if at least B line segments intersect it, and a line segment is associated with every subsquare that it intersects. This approach introduces substantial data replication and is thus not space efficient. To reduce the data replication problem, they use a time parameter U and rebuild the entire index at time U. Although this approach does not guarantee any nontrivial bound on the query performance, it extends to algebraic or piecewise-linear motion of S without any modification.

An index with provable query time can be developed using partition trees, described in Section 2. A 1-dimensional Q1 query on S asks which of the lines in L intersect a line segment σ parallel to the x-axis. Let P be the set of points dual to the lines in L; see Section 2. Then the above query is equivalent to reporting the points in P that lie inside the strip σ^* dual to σ. Since σ^* can be regarded as an unbounded triangle, a Q1 query can be answered in $O(n^{1/2+\varepsilon} + k)$ I/Os, and a point can be inserted into or deleted from the index in $O(\log_B^2 n)$ expected I/Os [1, 21]. Agarwal *et al.* [1] also showed that partition trees can also be used to answer Q2 queries with the same performance bounds. The partition-tree approach can be generalized to handle algebraic motion, at the cost of increasing the query time, by using the so-called linearization technique [10, 34]. Suppose the trajectory of each point $p \in S$ is given by a polynomial of degree D in the time parameter t. Using linearization, S is mapped to a set P of points in $\mathbb{R}^{D+1}$ and a Q1 query in S is reduced to reporting the points of P that lie inside a $(D+1)$-dimensional slab. By preprocessing P into a higher dimensional partition tree, a query can be answered using $O(n^{1-/(D+1)+\varepsilon} + k)$ I/Os; see [1] for details.

Chronological queries. The query time can be significantly improved if the index is allowed to change over time and if queries arrive in *chronological order*, i.e., if the time stamp of the current query is t_q, then the time stamp of the future queries is at least t_q. It is well known that a one-dimensional range query among a set of static points in $\mathbb{R}^1$ can be answered in $O(\log n + k)$ I/Os using B-trees. By maintaining B-trees for moving points using the KDS framework outlined in Section 2, a range query can be answered in $O(\log_B n + k)$ I/Os. If the motion of S is algebraic, the index processes $O(N^2)$ events and the index can be updated in $O(\log_B n)$ expected I/Os at each event. Agarwal *et al.* [1] show that the kinetic B-trees can be combined with partition trees to obtain a tradeoff between query time and the total number of kinetic events. Specifically, for a parameter Δ, where $N \le \Delta \le N^2$, range queries arriving in chronological order can be answered using $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os, and an event can be processed in $O(\log_B(\Delta/N))$ I/Os, provided the motion of S is linear. If the trajectories of points do not change, the index processes at most Δ events.

Time-responsive indexing. As observed in the previous section, the combinatorial structure of a kinetic data structure is fixed until there is an *event*. For kinetic B-trees, an event occurs when two points have the same value. Using the persistence technique described in Section 2, one can maintain all versions of kinetic B-trees in a total of $O(n + \sigma/B)$ blocks, where $\sigma = O(N^2)$ is the number of events processed by the KDS. Kollios *et al.* [21] used this approach to answer Q1 queries at any time in $O(\log_B n + k)$ I/Os. Agarwal *et al.* [1] showed that instead of maintaining all versions, one can maintain only a few past and future versions of the kinetic B-tree so that a query in the near past or future can be answered efficiently. Specifically, they showed that S can be stored in an index of size $O(n)$ so that a Q1 query can be answered in $O(\log_B n + k)$ I/Os, provided there are at most N events between t_q and now. The amortized cost of an event is $O(\log_B n)$ I/Os. Kollios *et al.* [22] have also used persistence to store multiple versions of other indexing schemes, such as R-trees, on moving objects.

Agarwal *et al.* [1] also described an index that combines partition trees with multiversion kinetic data structures to obtain an index whose query cost is a monotone function of $|t_q - now|$. However, they were not able to prove any bounds on the performance of their index in the worst-case. Subsequently, Agarwal *et al.* [4] proposed another index that provides a bound on the query performance, which is a monotone function of $|t - now|$. Since the index is of combinatorial nature, time is measured in terms of kinetic events. Let $\varphi(t)$ be the number of kinetic events that occur (or have occurred) between now and t. The query bounds then depend on $\varphi(t_q)$, the number of kinetic events between now and t_q. Their index answers a Q1 query at time t_q, with $NB^{i-1} \le \varphi(t_q) \le NB^i$, using $O(B^{i-1} + \log_B n + k)$ I/Os, and spends $O(\log_B^3 n)$ I/Os at each kinetic event. Recently Agarwal *et al.* [3] improved the query time to $O(\sqrt{B^{i-1}} + \log_B n + k)$ without affecting the update time.

4 Two-Dimensional Indexing

We now discuss how the techniques described in the previous section extend to points moving in $\mathbb{R}^2$. Again, we first discuss the time-oblivious approach, then the KDS framework, and finally the approaches that combine them.

Time-oblivious indexing. Pfoser *et al.* [24] propose two R-tree based schemes for indexing the trajectories of S, assuming that the motion of S is piecewise linear. For a point $p_i \in S$, let Γ_i denote the set of line segments in the trajectory $\bar{p}_i$, and set $\Gamma = \bigcup_i \Gamma_i$. Their first index, called the STR-tree, regards each segment of Γ independently and builds an R-tree on them. They propose new heuristics to split a node, which take the trajectories of S 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 circumvent this problem by storing all line segments of the same trajectory at the same leaf of the tree.

If the motion of S is linear, the trajectories of points in S are a set of lines in $\mathbb{R}^3$. Using the fact that a line in $\mathbb{R}^3$ can be represented by four real parameters, Kollios *et al.* [21] proposed mapping each line to a point in $\mathbb{R}^4$ and using four-dimensional partition trees to answer Q1 queries. Agarwal *et al.* [1] developed a considerably faster index by reducing the problem to 1D, using the following observation: a line ℓ in $\mathbb{R}^3$ (xyt-space) intersects a horizontal rectangle R, parallel to the xy-plane, if and only if their projections onto the xt- and yt-planes both intersect. We apply a duality transformation to the xt- and yt-planes, as described in Section 2. Thus, each moving point p in the xy-plane induces two static points p^x and p^y in the dual xt-plane and the dual yt-plane, respectively. For any subset $P \subseteq S$, let P^x and P^y respectively denote the corresponding points in the dual xt-plane and the dual yt-plane. Any query rectangle (query segments in the xt- and yt-planes) induces two query strips σ^x and σ^y, and the result of a query is the set of points $p \in S$ for which $p^x \in \sigma^x$ and $p^y \in \sigma^y$. See Figure 3.

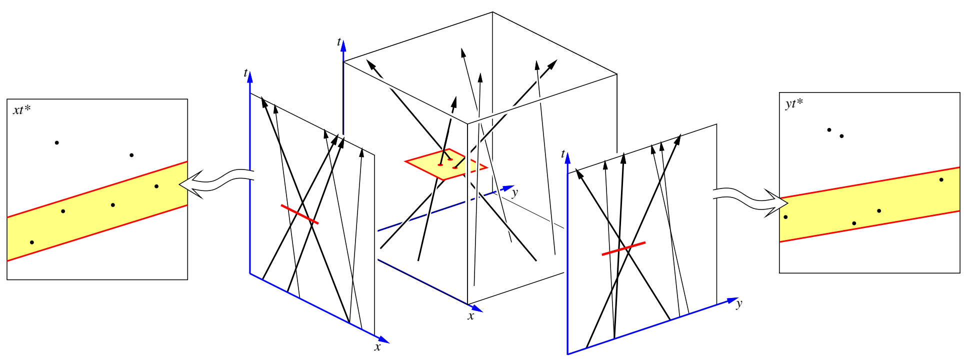


Figure 3: Decomposing a rectangle query among moving two-dimensional points into two strip queries among static two-dimensional points, by dualizing the xt- and yt-projections. A line intersects the rectangle if and only if both corresponding points lie inside the strips.

Agarwal *et al.* [1] propose a multi-level partition tree, a general technique that allows one to answer complex queries by decomposing them into several simpler components and by designing a separate data structure for each component, to answer Q1 and Q2 queries. Roughly speaking, they construct a *primary* partition tree T^x for the points P^x. Then at every node v of T^x, they attach a *secondary* partition tree T_v^y for the points S_v^y, where S_v is the set of points stored in the primary subtree rooted at v. The total space used by the index is $O(n \log_B n)$. A Q1 query is answered in almost the same way as in the basic partition tree. Given two query strips σ^x and σ^y, it first searches through the primary partition tree T^x for the points in $P^x \cap \sigma^x$. If it finds a triangle $\triangle_i$ associated with a node v of the partition tree T^x that lies completely inside σ^x, it searches in the secondary tree T_v^y to report all points of $P_v^y \cap \sigma^y$. Agarwal *et al.* [1] showed that the query takes $O(n^{1/2+\varepsilon} + k)$ I/Os and that the size can be reduced to $O(n)$ without affecting the asymptotic query time. As in the one-dimensional case, a

Q2 query can also be answered using within the same bound using multilevel partition trees, and the index can be generalized to handle algebraic motion of S using the linearization technique.

Chronological queries. Agarwal *et al.* [1] use the KDS framework on external range trees developed by Arge *et al.* [11] to answer Q1 queries on points moving in $\mathbb{R}^2$. The external range tree presented in [11] is a three-level structure. Omitting the details, which can be found in the original paper, we mention that the kinetic range tree uses $O(n \log_B n/(\log_B \log_B n))$ blocks, and a Q1 query with time stamp $t_q = now$ can be answered in $O(\log_B n + k)$ I/Os. The amortized cost of a kinetic event or trajectory change is $O(\log_B^2 n/\log_B \log_B n)$ I/Os. If the trajectories of the points do not change, the total number of events is $O(N^2)$. As in the one-dimensional case, by combining partition trees with kinetic range trees, a tradeoff between query time and the number of events can be obtained.

Since external range trees are too complicated, a more practical approach is to use the KDS framework on kd-trees, as proposed in [7]. However, the invariants maintained by a kd-tree — at each node points are partitioned into two halves by a line and the orientation of the line alternates between being horizontal and vertical along a path — are too expensive to maintain for moving points. Agarwal *et al.* [7] propose two variants of kinetic kd-trees in internal memory model: the *pseudo kd-tree*, which allows the number of points stored in the two children of a node to differ by a constant fraction, and the *overlapping kd-tree*, which allows the bounding boxes of two children of a node to overlap. Both variants answer Q1 queries that arrive in chronological order in $O(N^{1/2+\varepsilon})$ time, for any constant $\varepsilon > 0$, process $O(N^2)$ kinetic events, and spend only polylogarithmic time at each event. In contrast to other traditional approaches, which require subtrees to be rebuilt once in a while and thus generate some expensive updates, the update time in their structure is worst-case as long as the trajectories of the points do not change. Efficient KDS for maintaining binary space partition trees (also known as cell trees [17]), which are generalizations of kd-trees have also been developed [6, 8].

Time-responsive indexing. Similar to the one-dimensional case, one can maintain multiversion external kinetic range trees in order to answer queries both in the past and the future. Agarwal *et al.* [1] describe a method for maintaining such a structure and show that its size is $O(n \log_B n/(\log_B \log_B n))$ blocks, and the amortized cost per event is $O(\log_B^2 n)$ I/Os. A Q1 query can be answered in $O(\log_B n + k)$ I/Os, provided there are at most $n/\log_B n$ events between t and the current time. They also combine partition trees with multiversion kinetic data structures to obtain an index whose query cost is a monotone function of $|t_q - now|$. A different structure is proposed in [4]. It answers a Q1 query at time t_q in $O(\sqrt{N/B^i}(B^{i-1} + \log_B N) + k)$ I/Os, provided that the number of kinetic events that occur up to time t_q is in the interval $[NB^{i-1}, NB^i]$. The bounds were later improved in [3].

Since the above time-responsive indexing schemes are not practical, it is desirable to kinetize one of the simpler structures such as an R-tree. Recall that each node v of an R-tree is associated with a subset S_v of points and the smallest rectangle R_v containing S_v. Although the rectangles associated with the children of a node can overlap, the areas of overlap and the areas of the bounding boxes must be small to expect good query performance. Maintaining these properties over time is likely to be significantly less expensive than maintaining the stronger invariants that other indexes (e.g. kd-trees) require. Moreover, an R-tree works correctly even if the overlap areas are too large, though the query performance deteriorates.

There are two main challenges in extending an R-tree to moving points. For stationary points, a rectangle R_v can be represented by four real parameters, specifying the x- and y-coordinates of its bottom-left and top-right corners. But the smallest enclosing rectangle of moving points changes with time, and its shape can be quite complex. Figure 4 shows how the shape of the smallest interval containing a set of moving points changes with time. As proposed in [25, 32], this problem is circumvented by maintaining a rectangle $\tilde{R}_v$ at each node v such that $R_v(t) \subseteq \tilde{R}_v(t)$ for all values of t; see Figure 4. Another challenge in extending R-trees is partitioning the points of S_v among the children of v. Unlike the static case, the points have to be partitioned so that the overlap among the bounding rectangles of the children of v is small at all times. A trade-off between the query performance and the time spent in updating the tree can be obtained.

The first kinetic R-tree, called the TPR-tree, was proposed by Saltenis *et al.* [32]. It uses the notion of *time horizon* H in order to decide how to build the index at the current time. For example, in order to determine how the points of S_v should be partitioned among the children of v, it only considers the overlap among the bounding boxes for the time interval $[now, now + H]$. Similarly, it uses H to compute $\tilde{R}_v$. No kinetic events are defined, but the structure is updated due to external events such as changes in the trajectory, which is handled as a point deletion, followed by a point insertion. If the index is updated frequently, $\tilde{R}_v$ is a good approximation of R_v, but otherwise $\tilde{R}_v$ could be quite large, as evident in Figure 4 (i).

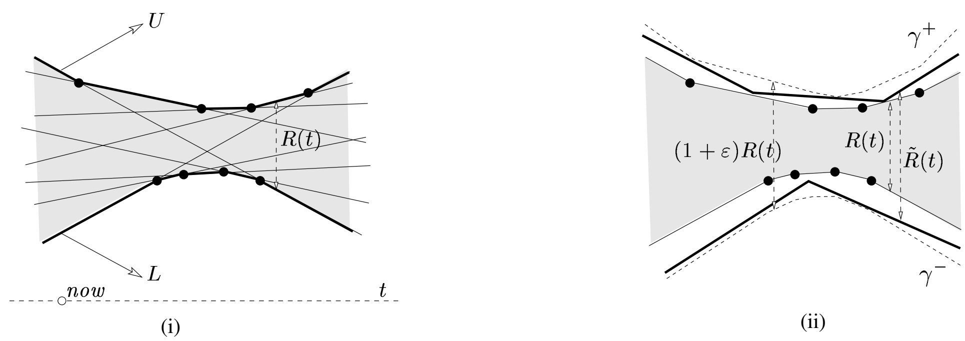


Figure 4: (i) $R(t)$ is the smallest interval containing $S(t)$. The rays U, L denote the approximation of $R(t)$ as computed in [32]. (ii) An ε-approximation of R by polygonal chains γ^-, γ^+ as computed in [25].

The second kinetic R-tree, called STAR-tree, was subsequently proposed by Procopiuc *et al.* [25]. Unlike the previous structure, they do not use the notion of time horizon. Instead they use kinetic events to update the index when the bounding boxes start overlapping a lot. Roughly speaking, if the bounding boxes of the children of a node v overlap considerably, it re-organizes the grand-children of v among the children of v. Using geometric approximation techniques developed in [9], it maintains a rectangle $\tilde{R}_v$ at each node, which is a close approximation of R_v. It provides a trade-off between the quality of $\tilde{R}_v$ and the complexity of the shape of $\tilde{R}_v$. For linear motion of S, the trajectories of the vertices of $\tilde{R}_v$ can be represented as polygonal chains. In order to guarantee that $\tilde{R}_v \subseteq (1+\varepsilon)R_v$, trajectories of the corners of $\tilde{R}_v$ need $O(1/\sqrt{\varepsilon})$ vertices. (Saltenis [32] represented the trajectory of each corner of $\tilde{R}_v$ by a ray.)

Hadjieleftheriou *et al.* [19] have proposed another kinetic R-tree, called PPR-tree, which also relies on storing an approximation of R_v at each node of the tree.

5 Nearest-Neighbor Queries

Given a set S of points in $\mathbb{R}^2$, the *Voronoi diagram* of S is the (maximal) planar subdivision in which the same point of S is nearest to all the points within a region. By preprocessing the Voronoi diagram of S for point-location queries [14], the point in S nearest to a query point can be computed efficiently. Although the Voronoi diagram of a set of moving points can be maintained efficiently, no efficient KDS for point-location structure is known.

Kollios et al. [20] describe a time-oblivious approach to answer nearest-neighbor queries (Q3 queries) for points moving along the x-axis. Although their index does not provide any bound on the worst-case query time, for linear motion of S, partition trees can be used to answer a nearest-neighbor query in $O(n^{1/2+\varepsilon})$ I/Os. Using the STAR-tree structure, Procopiuc *et al.* [25] describe branch-and-bound procedures that compute both exact and approximate k-nearest neighbors, for any $k \geq 1$. Although there are no theoretical bounds on the efficiency of the index, their approach has important practical value. They show through extensive experiments that the index can be expected to perform well for various data and velocity distributions.

Observing that the Euclidean metric can be approximated by a polygonal metric whose unit ball is a regular polygon with few edges, Agarwal *et al.* [1] developed an index for answering a δ-approximate nearest-neighbor query in $\mathbb{R}^2$, which returns a point of S whose distance from the query point at time t_q is at most $(1+\delta)$ times that to the nearest neighbor. Using $O(n/\sqrt{\delta})$ disk blocks, it answers a query in $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ I/Os.

6 Discussion and Open Problems

Uncertainty. One important direction for further research is to incorporate uncertainty in the position and velocity of the input points. Even though the accuracy of high-end positioning systems is sufficient for most kinetic applications, commercially viable applications will have to contend with much lower accuracy systems. In addition, it is currently impractical to sample with high enough frequency so that all trajectory changes are noticed and transmitted to the indexing system. Instead, one has to sample the motion and contend with the accumulation of errors in the data.

Complex queries. Besides range and nearest-neighbor queries, many other types of queries are of practical interest. For example, report the areas of the highest density at some given time stamp, or detect when the number of data points in the neighborhood of a query point exceeds a certain threshold. The first type of query is important for traffic observation and control, while the second type has applications in location-based services.

Engineering issues. The techniques presented in this paper assume a unique database server that maintains the index on the kinetic objects. Clearly, the ideas should be extended to distributed databases and also address issues such as concurrency and data recovery. Although the research on both the front-end and back-end aspects of kinetic databases is still in an early stage, the pace of its evolution suggests that integrated solutions may not be too far in the future.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, 2000. 175–186.

[2] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61:194–216, 2000.

[3] P. K. Agarwal, L. Arge, J. Erickson, and J. Vahrenhold. A time responsive indexing scheme for moving points. in preparation, 2002.

[4] P. K. Agarwal, L. Arge, and J. Vahrenhold. A time responsive indexing scheme for moving points. In *Proc. Workshop on Algorithms and Data Structures*, 2001.

[5] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.

[6] P. K. Agarwal, J. Erickson, and L. J. Guibas. Kinetic BSPs for intersecting segments and disjoint triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, 1998.

[7] P. K. Agarwal, J. Gao, and L. Guibas. Kinetic medians and kd-trees. manuscript, 2002.

[8] P. K. Agarwal, L. J. Guibas, T. M. Murali, and J. S. Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[9] P. K. Agarwal and S. Har-Peled. Maintaining approximate extent measures of moving points. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001.

[10] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[11] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.

[12] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.

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

[14] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[16] L. J. Guibas. Kinetic data structures — a state of the art report. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209. A. K. Peters, Wellesley, MA, 1998.

[17] O. Günther. The design of the cell tree: An object oriented index structure for geometric data bases. In *Proc. 5th IEEE Internat. Conf. Data Eng.*, pages 598–605, 1989.

[18] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Systems*, 25(1):1–42, 2000.

[19] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. VIII Conf. Extending Database Technology*, pages 251–268, 2002.

[20] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Spatiotemporal Database Management*, pages 119–134, 1999.

[21] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, pages 261–272, 1999.

[22] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. Knowledge and Data Engineering*, 13:758–776, 2001.

[23] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[24] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proc. International Conf. on Very Large Databases*, 2000.

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

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

[27] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[28] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. Intl Conf. Data Engineering*, pages 422–432, 1997.

[29] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, pages 185–200, 1998.

[30] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[32] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 331–342, 2000.

[33] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.

[34] A. C. Yao and F. F. Yao. A general approach to D-dimensional geometric queries. In *Proc. 17th Annu. ACM Sympos. Theory Comput.*, pages 163–168, 1985.