Indexing Moving Points

(Extended Abstract)

Pankaj K. Agarwal[*] Lars Arge[†] Jeff Erickson[‡]

Abstract

We propose three indexing schemes for storing a set S of N points in the plane, each moving along a linear trajectory, so that a query of the following form can be answered quickly: Given a rectangle R and a real value t_q, report all K points of S that lie inside R at time t_q. We first present an indexing structure that, for any given constant $\varepsilon > 0$, uses $O(N/B)$ disk blocks, where B is the block size, and answers a query in $O((N/B)^{1/2+\varepsilon} + K/B)$ I/Os. It can also report all the points of S that lie inside R during a given time interval. A point can be inserted or deleted, or the trajectory of a point can be changed, in $O(\log_B^2 N)$ I/Os. Next, we present a general approach that improves the query time if the queries arrive in chronological order, by allowing the index to evolve over time. We obtain a tradeoff between the query time and the number of times the index needs to be updated as the points move. We also describe an indexing scheme in which the number of I/Os required to answer a query depends monotonically on the difference between t_q and the current time. Finally, we develop an efficient indexing scheme to answer approximate nearest-neighbor queries among moving points.

[*]Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129; pankaj@cs.duke.edu; http://www.cs.duke.edu/~pankaj. Supported in part by National Science Foundation grants EIA–9870734, EIA–997287, and CCR–9732787, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corporation, and by a grant from the U.S.-Israeli Binational Science Foundation.

[†]Center for Geometric Computing, Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129; large@cs.duke.edu; http://www.cs.duke.edu/~large. Supported in part by National Science Foundation grants EIA–9870734, EIA–997287, and EIA–9984099.

[‡]Department of Computer Science, University of Illinois, Urbana, IL 61801; jeffe@cs.uiuc.edu; http://www.uiuc.edu/~jeffe. Supported in part by National Science Foundation grant DMS–9627683, by U.S. Army Research Office MURI grant DAAH04–96–1–0013, and by a Sloan Fellowship.

1 Introduction

Efficient indexing schemes that support range searching and its variants are central to any large database system. In relational database systems and SQL, for example, one-dimensional range searching is a commonly used operation [17, 27]. Various two-dimensional range-searching problems are crucial for the support of new language features, such as constraint query languages [17] and class hierarchies in object-oriented databases [17]. In spatial databases such as geographic information systems (GIS), range searching obviously plays a pivotal role, and a large number of external data structures (indexing schemes) for answering such queries have been developed (see [15, 22, 35] and references therein).

The need for storing and processing continuously moving data arises in a wide range of applications, including digital battlefields, air-traffic control, and mobile communication systems [10, 4]. Most of the existing database systems, which assume that the data is constant unless it is explicitly modified, are not suitable for representing, storing, and querying continuously moving objects because either the database has to be continuously updated or a query output will be obsolete. A better approach would be to represent the position of a moving object as a function $f(t)$ of time, so that the position changes without any explicit change in the database system. This way the database needs to be updated only when the function $f(t)$ changes (e.g., when the velocity of the object changes). Recently there has been some work on extending the capabilities of existing database systems to handle moving-object databases (MOD) (see e.g. [31, 32, 11]).

In this paper, we focus on developing efficient indexing schemes for storing a set of points, each moving in the xy-plane, so that range queries over their locations in the future can be answered quickly. An example of such a *spatio-temporal* query is: "Report all points that will lie inside a query rectangle R five minutes from now."

1.1 Problem statement

Let $S = \{p_1, p_2, \ldots, p_N\}$ be a set of moving points in the xy-plane. For any time t, let $p_i(t)$ denote the position of p_i at time t, and let $S(t) = \{p_1(t), \ldots, p_N(t)\}$. We will assume that each point p_i is moving along a straight line at some fixed speed, or more formally, that $p_i(t) = \mathbf{a}_i \cdot t + \mathbf{b}_i$ for some $\mathbf{a}_i, \mathbf{b}_i \in \mathbb{R}^2$. The trajectories of the points p_i can change at any time. We assume that the objects are responsible for updating the values $\mathbf{a}_i$ and $\mathbf{b}_i$ and that the database system is notified whenever these values change. We will use the term *now* to mean the current time.

We are interested in answering the queries of the following form:

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 $S(t_q) \cap R$.

Q2. Given a rectangle R and two time values t_1 and t_2, report all points of S that lie inside R at any time between t_1 and t_2, *i.e.*, report $\bigcup_{t=t_1}^{t_2}(S(t) \cap R)$. In many applications, either $t_1 = now$ or $t_2 = now$.

Q3. Given a query point $\sigma \in \mathbb{R}^2$ and a time value t_q, report a *δ-approximate nearest neighbor* of σ in S at time t_q, *i.e.*, a point $p \in S(t_q)$ such that $d(\sigma, p) \leq (1+\delta) \cdot \min_{r \in S(t_q)} d(\sigma, r)$.

As our main interest is minimizing the number of disk accesses needed to answer a range query, we will consider the problem in the standard external memory model. 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 range query. As we are interested in solutions that are *output sensitive*, our query I/O bounds are not only expressed 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 the minimum number of disk blocks we need to store N points is $\lceil N/B \rceil$. Similarly, at least $\lceil K/B \rceil$ I/Os are needed to report K output points. We refer to these bounds as "linear" and introduce the notation $n = \lceil N/B \rceil$ and $k = \lceil K/B \rceil$.

1.2 Previous results

A detailed summary of early work on temporal databases can be found in the survey paper by Salzberg and Tsotras [29]. Most early work concentrated on multiversion and/or time-series data. Recently, however, there has been a flurry of activity on developing data models and query languages for supporting continuously moving objects. Sistla and Wolfson [31] developed a temporal query language called *future temporal logic* (FTL) that supports proximity queries on moving objects. Sistla *et al.* [32] later proposed a data model called *moving objects spatio-temporal* (MOST) for moving objects and refined FTL. These models were later extended in [40, 36, 38, 39, 37] to incorporate several important issues, including uncertainty in the motion and communication cost. Other spatio-temporal models can be found in [11, 13, 14].

Although a number of practical methods have been proposed for accessing and searching moving objects [38, 33, 20], they all require $\Omega(n)$ I/Os in the worst case—even if the query output size is $O(1)$. Kollios *et al.* [18] proposed an efficient indexing scheme, based on partition trees [1, 2], for storing a set of points moving on the real line. It uses $O(n)$ disk blocks, answers a 1-dimensional query of type Q2 using $O(n^{1/2+\varepsilon} + k)$ I/Os, for an arbitrarily small constant $\varepsilon > 0$, and inserts or deletes a point using $O(\log_2^2 n)$ I/Os. They also present another scheme that uses $O(Nn)$ disk blocks and answers a query of type Q1 using $O(\log_B n + k)$ I/Os, assuming that the speed of a point never changes. Furthermore, they propose a practical data structure for points moving in $\mathbb{R}^2$, but it requires $\Omega(n)$ I/Os in the worst case to answer a query. In another paper, Kollios *et al.* [19] proposed a practical data structure for answering nearest-neighbor queries for moving points on the real line, but no bound on the query time was proved. Šaltenis *et al.* [28] propose an R-tree based indexing scheme for answering queries of type Q1 and Q2; see also [25].

In the computational geometry community, early work on moving points focused on bounding the number of changes in various geometric structures (e.g., convex hull, Delaunay triangulation) as the points move [30]. In their seminal paper, Basch *et al.* [7] introduced the notion of *kinetic data structures*. Their work has lead to several interesting results related to moving objects, including results on kinetic space partition trees (also known as cell trees) [3]. The main idea in the kinetic framework is that as the points move and their *configuration* changes, *kinetic updates* are performed on the data structure when certain events occur. Although the points move continuously, the combinatorial structure of the data structure change only at certain discrete times called *events*, and therefore one does not have to update the data structure continuously. In contrast to fixed-time-step methods, in which the fastest moving object determines the update time step for the entire data structure, a kinetic data structure is based on events, which have a natural interpretation in terms of the underlying structure. However, so far all work on kinetic data structures has been done in an internal memory computational model.

1.3 Our results

This paper contains four main results on indexing moving points in the plane. We use two essentially different approaches. The first approach regards time as the third axis and solves the problem in the xyt-space. The second approach, based on the work on kinetic data structures, regards points in the xy-plane, and the indexing scheme evolves over time. That is, it is modified when certain kinetic events occur, namely when the x- and y-coordinates of two points become equal. (The data structure is not necessarily updated at every one of such $O(N^2)$ events.) To our knowledge, the structures we develop are the first indexing schemes with provable performance bounds for answering range queries on points moving in the xy-plane.

In Section 2, we describe some useful general concepts from computational geometry. Next, in Section 3, we present an indexing scheme based on *partition trees* that uses $O(n)$ disk blocks and answers a query of type Q1 using $O(n^{1/2+\varepsilon} + k)$ I/Os, for any arbitrarily small constant $\varepsilon > 0$. A point can be inserted or deleted using $O(\log_B^2 n)$ I/Os. The index does not change unless the trajectory of a point changes, and therefore we say that the index is *time-oblivious.* It can also answer queries of type Q2 within the same I/O bound, though a point may be reported more than once (at most six times). Finally, the indexing scheme can also handle certain forms of uncertainty in the velocity of points, without affecting the asymptotic performance.

While the partition tree scheme is time-oblivious, the cost of a query is relatively high. In Section 4, we show that by allowing the index to evolve over time, we can answer a Q1 query using $O(\log_B n + k)$ I/Os, provided that the queries arrive in chronological order. This is achieved using the kinetic framework on an *external range tree* [5]. Our structure uses $O(n \log_B n/(\log_B \log_B n))$ disk blocks. Kinetic range trees (with slightly worse performance) were first developed in the internal-memory setting by Basch *et al.* [8]. We also show how one can combine kinetic range trees with partition trees to obtain a tradeoff between the query time and the number of events at which the kinetic index needs to be updated. Given a parameter $NB \le \Delta \le N^2$, we can answer a query in $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os, and provided the trajectories of the points do not change, there are $O(\Delta)$ events.

In many applications, such as air-traffic control, queries in the near future are more critical than queries far away the in future. In such applications, we need an indexing scheme that has fast response time for near-future queries but may take more time for queries that are far away. In Section 5, we propose such an indexing scheme. Using $O(n \log_B n/(\log_B \log_B n))$ disk blocks, it answers a query of type Q1 so that the number of I/Os required is a monotonically increasing function of $(t_q - now)$. The query time never exceeds $O(n^{1/2+\varepsilon} + k)$. If the points and their trajectories are uniformly distributed, then a query takes $O((\Delta_q/n)^{1/2} n^{\varepsilon} + k)$ I/Os, where $0 \le \Delta_q \le \binom{n}{2}$ is the number of kinetic events in the time interval $[now, t_q]$.

Finally, in Section 6, we describe an indexing scheme for answering Q3 queries. Given a parameter $\delta > 0$, we construct an indexing scheme, based on partition trees, that uses $O(n/\sqrt{\delta})$ disk blocks. For a query point $\sigma \in \mathbb{R}^2$ and a time value t_q it returns a δ-approximate nearest neighbor using $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ I/Os. A point can be inserted or deleted in $O((\log_B^2 n)/\sqrt{\delta})$ I/Os.

2 Geometric Preliminaries

In order to state our results we need some concepts and results from computational geometry.

2.1 Duality

Duality is a popular and powerful technique used in geometric algorithms; it maps each point in $\mathbb{R}^2$ to a line in $\mathbb{R}^2$ and vice-versa. We use the following duality transform: The dual of a point $(a_1, a_2) \in \mathbb{R}^2$ is the line $x_2 = -a_1 x_1 + a_2$, and the dual of a line $x_d = b_1 x_1 + b_2$ is the point (b_1, b_2). Let σ^* denote the dual of an object (point or line) σ; for a set of objects Σ, let $\Sigma^* = \{\sigma^* \mid \sigma \in \Sigma\}$. An essential property of duality is that a point p is above (resp., below, on) a line h if and only if the dual line p^* is above (resp., below, on) the dual point h^*. See Figure 1. The dual of a strip σ is a vertical line segment σ^* in the sense that a point p lies in σ if and only if the dual line p^* intersects σ^*.

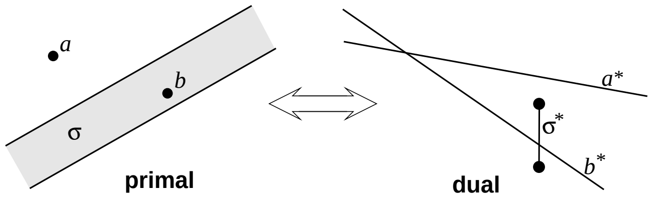

Figure 1. The duality transformation in two dimensions. The duals of two points and a strip are two lines and a vertical line segment.

2.2 External partition trees

Partition trees are one of the most commonly used internal memory data structures for geometric range searching [2, 21]. Our first indexing scheme is based on partition trees, which were originally described in Matoušek [21], and later extended in [1] to the external memory setting. We briefly summarize them here with an emphasis on insertion/deletion operations, as we slightly improve their performance compared to [1]

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 S_i. Note that a point of S may lie in many triangles, but it belongs to only one S_i; see Figure 2. The size of Π, here denoted r, is the number of 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.

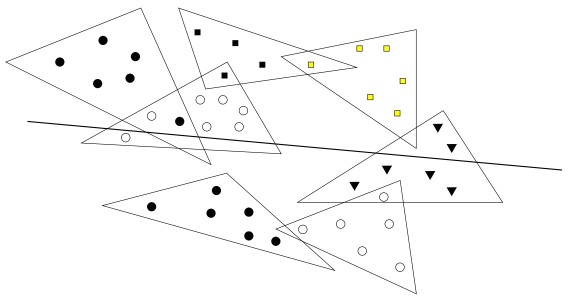

Figure 2. A balanced simplicial partition of size 7.

The following lemma states how fast a simplicial partition can be constructed. The I/O bound mentioned here is smaller than the one in [1]. Because of lack of space, we omit the proof.

Lemma 2.1. *Let S be a set of N points in the plane, and let $r = O(B)$ be a positive integer. We can construct a balanced simplicial partition Π of size r for S in $O(N)$ expected I/Os, such that the crossing number of Π is $O(\sqrt{r})$.*[1]

We construct a partition tree as follows: Each node v in a partition tree T is associated with a subset $S_v \subseteq S$ of points and a simplex $\triangle_v$. For the root u of T, we have $S_u = S$ and $\triangle_u = \mathbb{R}^2$. Let $N_v = |S_v|$ and $n_v = \lceil N_v/B \rceil$. For each node v, we construct the subtree rooted at 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 $r_v = \min\{cB, 2n_v\}$, where $c \geq 1$ is a constant to be specified later. We compute a balanced simplicial partition $\Pi_v = \{(S_1, \triangle_1), \ldots, (S_{r_v}, \triangle_{r_v})\}$ for S_v with small crossing number and then recursively construct a partition tree T_i for each subset S_i. For each i, we store the vertices of $\triangle_i$ and a pointer to T_i; the root of T_i is the ith child of v, and it is associated with S_i and $\triangle_i$. We need $O(c) = O(1)$ blocks to store any node v. Since r_v was chosen to be $\min\{cB, 2n_v\}$, every leaf node contains $\Theta(B)$ points. Thus the total number of nodes in the tree is $O(n)$, so the total size of the partition tree is $O(n)$. If we apply Lemma 2.1 recursively to build the entire partition tree the total construction time is $O(N \log_B n)$ expected I/Os.

[1]Matoušek [21] describes a more complicated deterministic algorithm that constructs a balanced simplicial partition with crossing number $O(\sqrt{r})$, but his algorithm requires more I/Os, and our simpler construction leads to the same asymptotic query bounds.

We want to be able to answer a query of the following type: Find all points inside a query strip σ. In order to do so, 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. Note that each point in σ is reported exactly once. As shown in [1], a query takes $O(n^{1/2+\varepsilon} + k)$ I/O, for any arbitrarily small constant $\varepsilon > 0$.

We can make our external partition tree dynamic using the *partial rebuilding* technique of Overmars [23]. At each node v in the tree, we store N_v, the number of points stored in its subtree. To insert a new point p into the subtree rooted at v, we first increment N_v. Then if v is a leaf, we add p to the subset S_v; otherwise, we find a triangle in the simplicial partition Π_v that contains p, and recursively insert p into the corresponding subtree. If more than one triangle in Π_v contains p, we choose the one whose subtree is smallest. The deletion process is similar.

In order to guarantee the same query time as in the static case, we occasionally need to rebuild parts of the partition tree after an update. A node u is *unbalanced* if it has a child v such that either $N_v < N_u/2r_u$ or $N_v > 4N_u/r_u$; in particular, the parent of a leaf v is unbalanced if either $N_v < B/4$ or $N_v > 2B$. (There is nothing special about the constants 2 and 4 in this definition.) To ensure that every node in the tree is balanced after inserting or deleting a point, we rebuild the subtree rooted at the unbalanced node closest to the root.

Rebuilding the subtree rooted at any node v takes $O(N_v \log_B n_v)$ expected I/Os, and N_v must be incremented or decremented $\Omega(N_v)$ times between rebuilds. Thus, the amortized cost of modifying N_v is $O(\log_B n_v)$ expected I/Os. Since each insertion or deletion changes $O(\log_B n)$ counters, the overall amortized cost of an insertion or deletion is $O(\log_B^2 n)$ I/Os. Using these bounds in the indexing scheme of Kollios *et al.* [18], we obtain their result on 1-dimensional range searching, but with improved update performance.

Theorem 2.2. *Given a set S of N linearly moving points in $\mathbb{R}$ and a parameter $\varepsilon > 0$, we can preprocess S into an index of size $O(n)$ blocks so that a Q2 query can be answered in $O(n^{1/2+\varepsilon} + k)$ I/Os. The data structure can be constructed in $O(N \log_B n)$ expected I/Os, and points can be inserted into or deleted at an amortized cost of $O(\log_B^2 n)$ expected I/Os each.*

3 Time-Oblivious Indexing

We now describe our first indexing scheme to answer Q1 queries for points in $\mathbb{R}^2$. The indexing scheme can easily

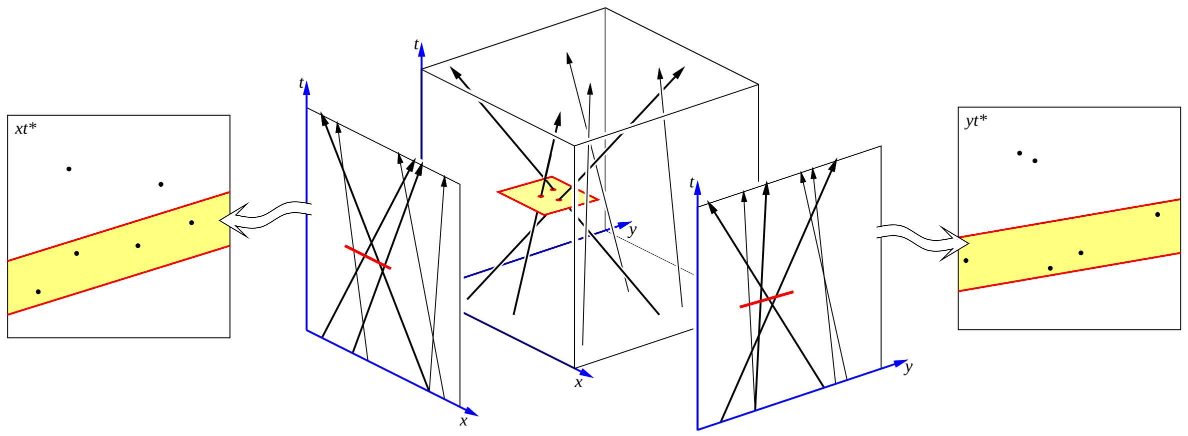

Figure 3. Decomposing a rectangle query among moving two-dimensional points into two strip queries among static two-dimensional points, by dualizing the xt- and yt-projections. A line intersects the rectangle if and only if both corresponding points lie inside their strips.

be generalized to higher dimensions, but for brevity we will describe only the two-dimensional case.

Let S be a set of N linearly-moving points in the xy-plane. These points trace out N lines in three-dimensional space-time; in this setting, a Q1 query asks which lines intersect a rectangle R on the plane $t = t_q$, parallel to the xy-plane. We can map each line to a point in $\mathbb{R}^4$ and use a four-dimensional partition tree to answer a Q1 query as in [18], but that would be very expensive. Instead, we use a *multi-level* partition tree. We observe that a line ℓ intersects R 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. Each moving point p in the xy-plane now 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 induces two query strips σ^x and σ^y, and the result of a query is the set of points $p \in S$ such that $p^x \in \sigma^x$ and $p^y \in \sigma^y$. See Figure 3.

We construct our data structure $\mathcal{T}$ as follows. First, we build a *primary* partition tree T^x for the points S^x. Then at certain nodes v of T^x, we attach a *secondary* partition tree T_v^y for the points S_v^y. Specifically, we attach secondary trees to every node whose depth is a multiple of $\alpha \log_B n$, where $0 < \alpha < 1$ is a constant to be determined later. Since the size of a single secondary tree T_v^y is $O(n_v)$ blocks, the total size of all the secondary trees is $O(n/\alpha) = O(n)$ blocks, we can construct them all in $O(N \log_B n)$ expected I/Os, and we can still insert or delete a point in $O(\log^2 B)$ (amortized) expected I/Os.[2]

The algorithm for answering a query is nearly the same as for the unadorned partition tree. Given two query strips σ^x and σ^y, we search through the primary partition tree for the points in $P^x \cap \sigma^x$. If we find a triangle $\triangle_i$ that lies completely inside σ^x, we do not perform a complete depth-first search of the corresponding subtree. Instead, we search only to the next level where secondary structures are available, and for each node v at that level, we use the secondary tree T_v^y to report all points of $P_v^y \cap \sigma^y$.

Let $\Sigma_1(N_v)$ denote the number of I/Os required to answer a query in some secondary partition tree T_v^y over N_v points, excluding the $O(K_v/B)$ I/Os used to report the K_v points inside the query range. In [1], it was shown that $\Sigma_1(N)$ obeys the recurrence

$$\Sigma_1(N_v) = 1 + O(\sqrt{r_v}) \cdot \Sigma_1(2N_v/r_v),$$

which has the solution $\Sigma_1(N) = O(n^{1/2+\varepsilon})$ for arbitrary small $\varepsilon > 0$. Similarly, let $\Sigma_2(N)$ denote the number of I/Os to answer a query in the entire two-level data structure, excluding the $O(k)$ I/Os required to report the output. Among the descendants of any node in T^x that we visit recursively, we perform $O((cB)^{\alpha \log_B n}) = O(n^{\alpha \log_B (cB)}) = o(n^{2\alpha})$ secondary queries, each on at most n_v points. Thus, $\Sigma_2(N)$ obeys the following recurrence:

$$\begin{aligned}\Sigma_2(N_v) &= o(n^{2\alpha}) \cdot \Sigma_1(N_v) + O(\sqrt{r_v}) \cdot \Sigma_2(2N_v/r_v) \\ &= o(n^{2\alpha}) \cdot O(n_v^{1/2+\varepsilon}) + O(\sqrt{r_v}) \cdot \Sigma_2(2N_v/r_v).\end{aligned}$$

The solution to this recurrence is $\Sigma(N) = O(n^{1/2+\varepsilon'})$, where $\varepsilon' < 2(\varepsilon + \alpha)$. We can make ε' arbitrarily close to ε by choosing α appropriately.

Theorem 3.1. *Given a set S of N linearly moving points in $\mathbb{R}^2$ and a parameter $\varepsilon > 0$, we can preprocess S into an index of size $O(n)$ blocks so that a Q1 query can be answered in $O(n^{1/2+\varepsilon} + k)$ I/Os. The index*

[2]We could attach secondary partition trees at every node in T^x, but the resulting index would require $O(n \log_B n)$ block and would have the same asymptotic query bound as this structure.

can be constructed in $O(N \log_B n)$ expected I/Os, and points can be inserted or deleted at an amortized cost of $O(\log_B^2 n)$ expected I/Os each.

Remarks.

(i) We can generalize our construction to any (constant) number of dimensions by building more levels of partition trees. The space, preprocessing time, query time, and update time of this multilevel partition tree only increase by a constant factor with each additional dimension.

(ii) If the points in S are distributed uniformly in some rectangular domain, then we can simplify our construction considerably by using a a regular $\sqrt{r} \times \sqrt{r}$ square grid instead of a simplicial partition. Details will appear in the full paper.

(iii) The external partition tree can also handle certain forms of uncertainty in the velocity of the location of points with the same query performance.

A query of type Q2—report all points lying in a rectangle R in the xy-plane at any time during the interval $[t_1, t_2]$—is equivalent to reporting all lines that intersect the box $\mathcal{B} = R \times [t_1, t_2]$. We can report all such lines by separately reporting the lines intersecting the top facet of $\mathcal{B}$, the left facet of $\mathcal{B}$, and the front facet of $\mathcal{B}$, using three separate copies of our earlier index.

Theorem 3.2. *Given a set S of N linearly moving points in $\mathbb{R}^2$ and a parameter $\varepsilon > 0$, we can preprocess S into an index of size $O(n)$ blocks so that a Q2 query can be answered in $O(n^{1/2+\varepsilon} + k)$ I/Os. The index can be constructed in $O(N \log_B n)$ expected I/Os, and points can be inserted or deleted at an amortized cost of $O(\log_B^2 n)$ expected I/Os each.*

4 Chronological Queries

In this section, we discuss how we can improve on the $O(n^{1/2+\varepsilon} + k)$ query bound presented in the previous section if we allow the data structure to change over time, and if we allow queries only at the current time. The approach can be extended to handle queries in future time as long as they arrive in *chronological order*. Our so-called *kinetic range tree* is based on an internal-memory data structure with suboptimal query time developed by Basch *et al.* [8] and a query-optimal external range tree developed in [5]. We will present our range tree in the primal setting. We give a brief overview of the external range tree in Section 4.1, and then discuss how to *kinetize* it in Section 4.2. The main idea will be to store only a snapshot of the moving points at any time; whenever the "sorted order" of the points changes, we perform a *kinetic update* on the data structure. Finally, in Section 4.3, we discuss a tradeoff between the query cost and the cost of updating the data structure.

4.1 External range trees

The external range tree $\mathcal{K}$ presented in [5] is a three-level structure. The primary structure is a tree over the x-coordinates of the points in S, similar to a B-tree with fan-out $\log_B n$. This tree has height $O(\log_{\log_B n} n) = O(\log_B n / (\log_B \log_B n))$, and a point p is stored in secondary structures of all the nodes on the path from the root to the leaf storing p. These secondary structures use linear space, so the structure use $O(n \log_B n / (\log_B \log_B n))$ blocks in total. The secondary structures are external versions of a priority search tree, which can be used to answer three-sided range queries, i.e., reporting the points lying in a rectangle of the form $[a, b] \times [c, \infty)$. The basic structure of the external priority search tree is that of a B-tree, in which each node v has an auxiliary structure $\mathcal{A}_v$ containing B^2 points. $\mathcal{A}_v$ uses linear space, and supports three-sided queries and updates in $O(1 + k)$ I/Os and $O(1)$ I/Os, respectively. For brevity we only discuss the B^2 structure below. The full description of the external range tree appears in [5], where the following is proved.

Lemma 4.1 ([5]). *A set of N points in $\mathbb{R}^2$ can be stored in an index using $O(n \log_B n / (\log_B \log_B n))$ disk blocks, so that a range query can be answered in $O(\log_B n + k)$ I/Os. Points can be inserted or deleted at an amortized cost of $O(\log_B^2 n)$ I/Os each.*

Consider a set S of B^2 points (x_i, y_i) ordered in increasing order of their x-coordinates. The structure $\mathcal{A}$ for answering 3-sided range queries on S consists of $2B - 1$ blocks $b_1, b_2, \ldots, b_{2B-1}$ storing the points, plus a constant number of *catalog blocks*. An x-interval $[x_{l_i}, x_{r_i}]$ and a y-interval $[y_{u_i}, y_{v_i}]$ are associated with each block b_i. The catalog blocks store these $2B - 1$ x- and y-intervals. The block b_i contains a point (x_j, y_j) of S whenever the following condition holds:

$$x_{l_i} \le x_j \le x_{r_i} \quad \text{and} \quad y_{u_i} \le y_j \qquad (\clubsuit)$$

The blocks are constructed as follows. Initially, we create B blocks $b_1, b_2, \ldots, b_B$. For each $i \le B$, the x-interval of b_i is $[x_{(i-1)B+1}, x_{iB}]$, and the lower endpoint y_{u_i} of b_i is $-\infty$. Hence b_i contains the points $(x_{(i-1)B+1}, y_{(i-1)B+1}), \ldots, (x_{iB}, y_{iB})$. Next, we sweep a horizontal line upwards from $y = -\infty$, and for each block b_i, we keep track of the number of points in b_i lying above the sweep line. When the sweep line reaches a point (x_j, y_j) such that two consecutive blocks b_i and b_{i+1} both have fewer than $B/2$ points lying above the line $y = y_j$, a new block b_r is created with y_j as the lower endpoint of its y-interval and with $[x_{l_i}, x_{r_{i+1}}]$ as its x-interval. That is, b_r contains the (at most B) points of b_i and b_{i+1} that lie above the line $y = y_j$. The upper endpoints of the y-intervals of b_i and b_{i+1} are set to y_j, and we no longer keep track of b_i and b_{i+1} during the sweep. The sweep continues in this manner until the line reaches $+\infty$. At that point at most $B+(B-1) = 2B-1$ blocks have been created. In [5] it is shown how the above structure can be constructed in $O(B)$ I/Os.

To answer a 3-sided query $[a, b] \times [c, \infty)$, we first load the catalog blocks into main memory using $O(1)$ I/Os. We then identify all blocks whose y-interval includes c (*i.e.*, the blocks under consideration when the sweep line was at $y = c$) and whose x-range intersects the interval $[a, b]$. We load these blocks into main memory one at a time and report the relevant K points. It is shown in [5] that the query takes $O(1 + K/B)$ I/Os. See Figure 4.

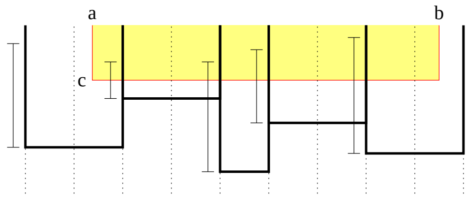

Figure 4. Answering query $q = (a, b, c)$. Dotted lines indicate the B initial blocks (before the vertical sweep). Solid blocks are active when the sweep line is at $y = c$; the y-interval of each active block is indicated to its left. All active blocks are loaded to answer q, and except for the blocks containing a and b, every consecutive pair of blocks contributes at least $B/2$ points to the output.

The structure can easily be made dynamic using global rebuilding [23]. Updates are simply logged in an extra block $\mathcal{B}$. After B updates, when $\mathcal{B}$ becomes full, the entire structure is rebuilt from scratch in $O(B)$ I/Os. Thus, the amortized cost of any insertion or deletion is $O(1)$ I/Os. At every query, we spend one extra I/O examining $\mathcal{B}$ to ensure that query results are consistent with recorded updates.

Lemma 4.2. *A set of B^2 points can be stored in an index of size $O(B)$ blocks, so that a 3-sided range query can be answered in $O(1 + k)$ I/Os. The index can be constructed in $O(B)$ I/Os, and points can be inserted or deleted at an amortized cost of $O(1)$ I/Os each.*

4.2 Kinetic range trees

We now discuss how the external range tree [5] can be modified to work on moving points such that range queries at the current time can be answered quickly. We will only explain the modifications needed on the structure $\mathcal{A}$ for answering 3-sided queries on B^2 points. Further details will appear in the full paper.

Recall that the endpoints of the x- and y-intervals of each block b_i in $\mathcal{A}$ were defined by four points of S. If $[x_{l_i}, x_{r_i}]$ and $[y_{u_i}, y_{v_i}]$ are the x-and y-intervals of b_i, then we call the 4-tuple (l_i, r_i, u_i, v_i) the *signature* of b_i. In the case of moving points, we define the x- and y-intervals of b_i at time t to be $[x_{l_i}(t), x_{r_i}(t)]$ and $[y_{u_i}(t), y_{v_i}(t)]$. Thus the x- and y- intervals of each block change continuously with time. However, a point $p_j = (x_j, y_j)$ in b_i continues to satisfy condition (♣) until some time t when either $x_j(t) = x_{l_i}(t)$, $x_j(t) = x_{r_i}(t)$, or $y_j(t) = y_{u_i}(t)$. Consequently, the *content* of blocks change only at discrete instants. The time values t at which one of the above three equalities holds are called *kinetic events.*

We handle kinetic events exactly as we handle dynamic updates, by global rebuilding. We maintain an extra update block $\mathcal{B}$, and whenever a point p leaves a block b_i, we delete p from b_i and add it to $\mathcal{B}$. When $\mathcal{B}$ becomes full (after B events), we reconstruct the entire data structure from scratch. The amortized cost of each event is thus $O(1)$ I/Os. For each point p_j in b_i, we compute the first time t_{ij} when it will leave b_i. We store these time values in a priority queue, called the *event queue*, and keep track of the next event $t^* = \min_{i,j} t_{ij}$. At *now* $= t^*$, we update the data structure as described above and update the priority queue. Whenever a point leaves a block, it enters another one, and this requires computing another failure time t_{ij} and storing that in the event queue. The event queue can be updated using $O(1)$ I/Os at each event. By modifying the above scheme slightly, we can also change the trajectory of a point in $O(1)$ I/Os. We omit the details.

To answer a query $q = [a, b] \times [c, \infty)$ at time t_q, we simply load the catalog blocks and calculate the x- and y-intervals of the blocks at time t_q. We then load $\mathcal{B}$ and those blocks whose x-intervals intersect the interval $[a, b]$ at time t_q and whose y-intervals intersect c at time t_q. We then report the points that lie in the query rectangle. As previously, the query procedure takes $O(1 + K/B)$ I/Os to report K points.

Lemma 4.3. *Let S be a set of B^2 points in $\mathbb{R}^2$, each moving with a fixed velocity. We can store S in an index of size $O(B)$ blocks, so that a current 3-sided query can be answered in $O(1 + k)$ I/Os. Amortized cost of each event is $O(1)$ I/Os. If the trajectories of points do not change over time, then there are a total of $O(B^2)$ events.*

Using similar ideas, we can evolve the primary and secondary structures of the external range tree. Events are now defined to be the time instances at which the x- or y-coordinates of two points become equal. These events can be calculated efficiently and stored in a global priority queue so that the next event can be computed in $O(\log_B n)$ I/Os. The data structure can be updated in $O(\log_B^2 n)$ I/Os at each such event. Our approach is similar to the techniques of Basch *et al.* [8] for kinetizing internal-memory range trees. Details will appear in the full paper.

Theorem 4.4. *Given a set S of N linearly moving points in $\mathbb{R}^2$, we can preprocess S into a kinetic range* *tree $\mathcal{K}$ of size $O(n \log_B n/(\log_B \log_B n))$ blocks so that a current Q1 query can be answered in $O(\log_B n+k)$ I/Os. The amortized cost of a kinetic event is $O(\log_B^2 n)$ I/Os. if the trajectories of points do not change over time, the total number of events is $O(N^2)$.*

Remark. The kinetic range tree works within the same I/O bounds even if the points move along more complex trajectories, provided we can quickly determine when a moving point crosses a given horizontal or vertical line.

4.3 Query/update tradeoffs

If we are going to perform only a few queries, it is not efficient to spend a total of $O(N^2 \log_B^2 N)$ time evolving the kinetic range tree. We can combine the kinetic range tree $\mathcal{K}$ with our external partition tree $\mathcal{T}$ to obtain a tradeoff between the cost of answering queries and the total number of events. Our construction is similar to a general technique used to establish tradeoffs between data structure size and query time [2]. We will first describe the tradeoff structure for moving one-dimensional points, and then briefly describe how to generalize it to two dimensions. Many details will be deferred to the full paper.

Let Δ be a parameter between B^2N and N^2. Our structure for moving one-dimensional points consists of the top ℓ levels of an external partition tree, where $\ell = \log_{r/4}(N^2/\Delta)$ and $r = cB$ is the fanout of the tree. This tree clearly has r^ℓ leaves. At each leaf v, we store the corresponding subset of $N_v \leq N/(r/2)^\ell$ points in a kinetic one-dimensional range tree. See Figure 5.

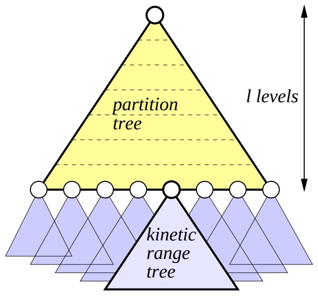

Figure 5. Schematic picture of our query/update tradeoff structure.

Each kinetic range tree undergoes $O(N^2(2/r)^{2\ell})$ kinetic events, so the total number of kinetic events for the entire structure is $O(N^2(4/r)^\ell) = O(\Delta)$. To compute the number of I/Os required to answer a query, we modify our earlier recurrence for $\Sigma_1(N_v)$ by adding a new base case $\Sigma_1(N_v) = O(\log_B n)$ for all $N_v \leq N/(r/2)^\ell$. The solution for the modified recurrence is $\Sigma_1(N) = O((r^\ell)^{1/2+\varepsilon})$. Note that $r^\ell = (N^2/\Delta)^{1+\lambda}$, where $\lambda = 1/(\log_2 r - 2)$; by setting the constant c appropriately, we can make λ arbitrarily small. Thus, the overall query time for this combined structure is $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os, where $\varepsilon' < \varepsilon + \lambda$.

Theorem 4.5. *Given a set S of N linearly moving points in $\mathbb{R}^1$ and a parameter $BN \leq \Delta \leq N^2$, we can preprocess S into an index of size $O(n)$ blocks so that a current Q1 query can be answered in $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os. If the trajectories of the points do not change, there are at most $O(\Delta)$ events, and the amortized cost per event is $O(\log_B n)$ I/Os.*

To generalize these results to moving points in the plane, we use the multi-level partition tree from Section 3. We modify the primary structure T^x by replacing all subtrees at level ℓ (including their secondary structures) with kinetic two-dimensional range trees. For each node $v \in T^x$ that has a secondary structure T_v^y, we modify T_v^y by replacing all subtrees at level $\ell - d_v$ with kinetic one-dimensional range trees, where d_v is the depth of v in T^x. Omitting the analysis from this extended abstract, we conclude:

Theorem 4.6. *Given a set S of N linearly moving points in $\mathbb{R}^2$ and a parameter $BN \leq \Delta \leq N^2$, we can preprocess S into an index of size $O(n \log_B n/(\log_B \log_B n))$ blocks so that a current Q1 query can be answered in $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os. If the trajectories of the points do not change, there are at most $O(\Delta)$ events, and the amortized cost per event is $O(\log_B^2 n)$ I/Os.*

5 Time-Responsive Indexing

The partition tree $\mathcal{T}$ has the advantage that it can answer Q1 queries in any order, but the cost of answering a query is high. On the other hand, the kinetic range tree $\mathcal{K}$ answers a query quite fast, but it can handle queries only in chronological order. In this section, we present an indexing scheme that combines the advantages of both schemes—it can answer queries in any order, and the number of I/Os need to answer a query is small if the query time t_q is close to the current time. We first describe how to answer a Q1 query in near future or in near past in $O(\log_B n + k)$ I/Os, and then extend our approach to arbitrary query times.

5.1 Recent-past and near-future queries

As observed in the previous section, the combinatorial structure of the kinetic range tree $\mathcal{K}$ does not change until the x- and y-coordinates of two points become equal. At some of these events, we update the data structure using $O(\log_B^2 n)$ I/Os. Let $e_1, e_2, \ldots$ be the sequence of events at which $\mathcal{K}$ is updated, let t_i be the time at which the event e_i occurs, and let $\mathcal{K}_i$ be the version of $\mathcal{K}$ between the events e_{i-1} and e_i. (For notational convenience, we define $t_i = -\infty$ for

$i = 0$.) $\mathcal{K}_i$ is valid for the time interval $[t_{i-1}, t_i)$. Let $\Delta = n/\log_B n$. If the current time *now* $\in [t_{i-1}, t_i)$, we maintain the versions $\mathcal{K}_{i-\Delta}, \mathcal{K}_{i-\Delta+1}, \ldots, \mathcal{K}_{i+\Delta}$ of $\mathcal{K}$, that is, we maintain Δ past versions and Δ future versions of $\mathcal{K}$. Note that the future versions $\mathcal{K}_i, \ldots \mathcal{K}_{i+\Delta}$ are tentative since they are built by anticipating future events, based on the current trajectories of the points. If the trajectory of a point changes, the future versions of the structure might also change.

Now, instead of storing each $\mathcal{K}_i$ explicitly, we store the "differences" between $\mathcal{K}_{i-1}$ and $\mathcal{K}_i$, using the ideas of persistent data structures (see, for example, [12, 9, 34]). We call our data structure an *interim-persistent* data structure. Recall that $\mathcal{K}$ has three levels. The primary and secondary structures are variants of B-trees, and we can store the different version of them by adapting the standard techniques [12, 9, 34]. Because of lack of space we will not discuss the modifications needed to tailor these techniques for our purposes. We will instead focus on the auxiliary structure $\mathcal{A}_v$ stored at each node v of the secondary B-tree, which answers a 3-sided range query on a set A of $O(B^2)$ points.

Recall that $\mathcal{A}$ maintains an extra update block $\mathcal{B}$ that records kinetic events; after B kinetic events have occurred, $\mathcal{A}$ is reconstructed and the old version of $\mathcal{A}$ is discarded. To maintain multiple versions of $\mathcal{A}$, we modify the update and query procedures as follows. First, whenever a point enters or leaves a block, we record the time of the event in the update block $\mathcal{B}$ in addition to the identity of the point. When $\mathcal{A}$ is reconstructed, we do not discard the old version, but instead declare it *inactive*. During T kinetic events we thus maintain a sequence of versions $\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^{T/B}$ of $\mathcal{A}$, only the last of which is active. For each inactive version $\mathcal{A}^j$, we store its *death* time d_j, the time at which $\mathcal{A}^j$ became inactive; for notational convenience we set $d_0 = -\infty$ and $d_s = \infty$, where $\mathcal{A}_s$ is the active version. To maintain the T versions of $\mathcal{A}$, we need $O(T)$ disk blocks.

To report all points of $\mathcal{A}$ that lie in a 3-sided rectangle R at time t_q, we first find the version $\mathcal{A}^j$ that is active at time t_d, *i.e.*, such that $d_{j-1} \le t_q < d_j$. Since we also maintain multiple versions of the primary and secondary structures of the kinetic range tree, $\mathcal{A}^j$ can be found using $O(1)$ I/Os. Then we simply query $\mathcal{A}^j$ as in Section 4, except that we report only those points from the update block that were inserted before t_q, and that we do not report the points that were deleted before t_q. In total we use $O(1 + K_v/B)$ I/Os, where $K_v = |A(t_q) \cap R|$.

Finally, we store the death times of all versions of all auxiliary structures in a global priority queue. When the ith kinetic event occurs, at time t_i, we delete all the versions of auxiliary structures whose death times lie in the interval $[t_{i-\Delta}, t_{i-\Delta+1})$. We maintain $\Delta = n/\log_B n$ versions of $\mathcal{K}$ and at each event we perform at most four update operations on $\mathcal{K}$, which in turn implies performing $O(\log_B^2 n/(\log_B \log_B n))$ update operations on auxiliary structures. Thus, the total size of the data structure is $O(n \log_B n/(\log_B \log_B n))$. We still spend $O(\log_B^2 n)$ amortized I/Os at each event to update $\mathcal{K}$.

To change the trajectory of a point p, we must update it in every version of every auxiliary structure that stores p. If p is stored in T different auxiliary structures, then $O(T \log_B n)$ I/Os are needed to update the data structure. In the worst case $T = \Omega(n)$, but in practice it will be much smaller. Omitting further details, we conclude:

Theorem 5.1. *Given a set S of N linearly moving points in $\mathbb{R}^2$, we can preprocess S into an index of size $O(n \log_B n/(\log_B \log_B n))$ blocks, so that 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_q and the current time. The amortized cost per event is $O(\log_B^2 n)$ I/Os. If a point p is stored at T places in the index, we can update the trajectory of p using $O((1+T)\log_B n)$ I/Os.*

A significantly simpler data structure can be developed if S is a set of points in $\mathbb{R}^1$. In this case we can directly apply the ideas of [12, 9, 34] and obtain the following result. Details will appear in the full paper.

Theorem 5.2. *Given a set S of N linearly moving points in $\mathbb{R}^1$, we can preprocess S into an index $\mathcal{K}$ of size $O(n)$ blocks, 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 the current time. The amortized cost per event is $O(\log_B^2 n)$ I/Os. If a point p is stored at T places in the index, we can update the trajectory of p using $O((1 + T)\log_B n)$ I/Os.*

5.2 Answering distant-future queries

We now combine the partition tree with the interim persistent index to obtain an index whose query cost is a function of $|t_q - \mathit{now}|$. The combination is similar to the multilevel partition trees discussed in Section 3. For simplicity, we describe the indexing scheme in $\mathbb{R}^1$ only; a similar approach works in $\mathbb{R}^2$.

Let S be a set of N points in $\mathbb{R}^1$, each moving with fixed velocity. We convert S into a set of lines in the xt-plane, and let P denote the set of points dual to these lines. We construct a partition tree $\mathcal{T}$ on P as described in Section 2.2. Each node v of $\mathcal{T}$ is associated with a subset $P_v \subseteq P$ of points. Let $S_v \subseteq S$ be the set of input points corresponding to P_v.

For each node v whose depth is a multiple of $\alpha \log_B n$, where $\alpha > 0$ is a small constant, we construct an interim persistent index $\mathcal{K}_v$ on S_v (Theorem 5.2). We also maintain the time interval $[t_v^-, t_v^+]$ in which

$\mathcal{K}_v$ is valid. The total size of the resulting structure is $O(n)$ disk blocks. An event is now defined to be a time instance at which two points stored in some secondary structure collide. At each such event point, the data structure can be updated using $O(\log_B n)$ I/Os.

A 1-dimensional Q1 query—report all points of $S(t_q)$ that lie in an interval I—can be answered as follows. We traverse $\mathcal{T}$, starting from the root, as in Section 2.2. Suppose we are at a node v. If v has a secondary structure $\mathcal{K}_v$ and $t_q \in [t_v^-, t_v^+]$, then we report all points of $S_v(t_q) \cap I$ using $\mathcal{K}_v$. This requires $O(\log_B n + K_v/B)$ I/Os, where $K_v = |S_v(t_q) \cap I|$. Otherwise, we visit the children of v exactly as in Section 2.2.

The maximum number of I/Os needed to answer a query, independent of t_q, is $O(n^{1/2+\varepsilon} + k)$; in the worst case, we never use any secondary structure $\mathcal{K}_v$. However, if t_q is close to the current time, the query procedure will visit only a few levels of the partition tree and then it will switch to the secondary kinetic structures. It is difficult to bound the query time in the worst-case without assuming any distribution on the trajectories of points, since several events can occur in a short period of time. If we assume that the points and their trajectories are uniformly distributed (*i.e.*, the points in P are uniformly distributed in a unit square), then we can simplify the partition tree and prove the following.

Theorem 5.3. *Given a set S of N linearly moving points in $\mathbb{R}^1$, we can preprocess S into an index of size $O(n)$ blocks so that the cost of a Q1 query at time t_q is a monotonically increasing function of $|t_q - now|$. If the points and their trajectories in S are uniformly distributed, then a query takes $O((\Delta/n)^{1/2}n^{\varepsilon} + k)$ expected I/Os, where $0 \le \Delta \le \binom{n}{2}$ is the number of events between t_q and now.*

In $\mathbb{R}^2$ we can prove a similar result, with a slightly worse bound on the size of the data structure and update time. Details will appear in the full version of this paper.

Theorem 5.4. *Given a set S of N linearly moving points in $\mathbb{R}^2$, we can preprocess S into an index of size $O(n \log_B n/(\log_B \log_B n))$ blocks so that the cost of a Q1 query at time t_q is a monotonically increasing function of $|t_q - now|$. If the points and their trajectories in S are uniformly distributed, then a query takes $O((\Delta/n)^{1/2}n^{\varepsilon} + k)$ expected I/Os, where $0 \le \Delta \le \binom{n}{2}$ is the number of events between t_q and now.*

6 Almost-Nearest Neighbor Searching

In this section we briefly sketch an indexing scheme for answering Q3 queries. The main idea is to approximate the Euclidean metric with a metric whose unit ball is a regular polygon with few edges.

For any polygon P that contains the origin, we define the distance function $d_P : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^+$ as $d_P(a,b) = \inf\{\lambda \mid b \in \lambda P + a\}$. We can easily verify that this function is a metric if and only if the polygon P is centrally symmetric about the origin.

Let $m = 2\left\lceil 2/\sqrt{\delta} \right\rceil$, and let P be the regular m-gon of circumscribing radius 1, centered at the origin and with a vertex on the x-axis; see Figure 6(i). Since m is even, d_P is a metric, and an easy trigonometric calculation shows that $d_P(a,b) \le (1+\delta)d(a,b)$; see Figure 6(ii).

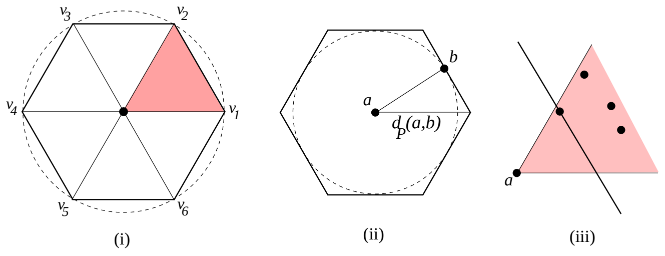

Figure 6. (i) A regular m-gon P centered at the origin. (ii) The difference between the Euclidean and d_P metrics. (iii) The wedge Q_a and the nearest neighbor to a under the distance function $d_{\triangle_1}$.

Decompose P into m triangles $\triangle_1, \dots, \triangle_m$, by connecting every vertex of P to the origin. Obviously, $d_P(a,b) = \min_{1\le i\le k} d_{\triangle_i}(a,b)$. Thus, to find a nearest neighbor in the d_P metric, it suffices to compute the nearest neighbor of a query point under the distance function $d_{\triangle_i}$. Our indexing scheme for approximate nearest neighbor queries consists of a separate data structure for each $\triangle_i$. Without loss of generality, consider $\triangle_1$. For a point a, let $Q(a)$ be the wedge formed by the rays emanating from a in directions ov_1 and ov_2; see Figure 6 (iii). The nearest neighbor of a point σ under the distance function $d_{\triangle_1}$ is the point in $Q(\sigma) \cap S$ nearest to σ in the direction normal to v_1v_2; see Figure 6(iii). For any point $p \in S$, let $f(p)$ be the dot product of p with the vector normal to v_1v_2. It is easy to prove that the nearest $d_{\triangle_1}$-neighbor in S of a point $a \in \mathbb{R}^2$ is the point $p \in S$ minimizing $f(p)$.

We thus have the following problem at hand. We want to preprocess a set S of N moving points in the plane to answer queries of the following form:

Q3'. Given a point σ and a time t_q, compute the point in $p(t_q) \in Q(\sigma)$ minimizing $f(p(t_q))$.

To solve this problem, we use a modification of the two-level partition tree $\mathcal{T}$ described in Section 3. First we construct a partition tree T over the projections of S onto the line ov_1. At certain nodes v of T^1, we construct a secondary partition tree T_v^2 over the projections of the subset S_v onto the line ov_2. Finally, we attach tertiary structures to certain nodes in T_y^2.

Specifically, let v be a node in some secondary partition tree whose depth in that tree is a multiple of $\alpha \log_B n$, for some small constant $\alpha > 0$, and let S_v be the subset of N_v points stored in its subtree. Define $F_v(t) = \min_{p \in S_v} f(p(t))$ to be the *lower envelope* of the functions $f(p(t))$. The graph of F_v is a convex chain with at most N_v vertices, and it can be computed in $O(n_v \log_B n_v)$ I/Os [16]. We store this chain at v. For a given value of t, we can compute $F_v(t)$ in $O(\log_B n)$ I/Os. We can also insert or delete a point in S_v and update the graph of F_v, at an amortized cost of $O(\log_2 n \log_B n)$ I/Os [1, 24]. The total size of our three-tier data structure is $O(n)$ blocks.

Given a query point σ and time value t_q, we answer a Q3' query as follows. We first compute $O(n^{1/2+\varepsilon})$ nodes $v_1, v_2, \ldots, v_m$ in the secondary trees so that $S(t_q) \cap Q(\sigma) = \bigcup_{i=1}^{r} S_{v_i}$, using the query algorithm described in Section 3. Then for each v_i, we compute $F_{v_i}(t_q)$, and finally we return the point p of $S(t)$ that attains $\min_{1 \le i \le m} F_{v_i}(t_q)$.

Omitting further details, we obtain the following.

Lemma 6.1. *Given a set S of N linearly moving points in $\mathbb{R}^2$ and two parameters $\varepsilon > 0$, we can preprocess S into an index of size $O(n)$ blocks so that a Q3' query can be answered in $O(n^{1/2+\varepsilon})$ I/Os. The index can be constructed in $O(N \log_B N)$ expected I/Os, and points can be inserted or deleted at an amortized cost of $O(\log_2 n \log_B n)$ expected I/Os each.*

Since our data structure for Q3 queries consists of $\Theta(1/\sqrt{\delta})$ separate copies of the Q3' index, one for each triangle $\triangle_i$, we conclude:

Theorem 6.2. *Given a set S of N linearly moving points in $\mathbb{R}^2$ and two parameters $\varepsilon, \delta > 0$, we can preprocess S into an index of size $O(n/\sqrt{\delta})$ blocks so that a Q3 query can be answered in $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ I/Os. The index can be constructed in $O(N \log_B N/\sqrt{\delta})$ expected I/Os, and points can be inserted or deleted at an amortized cost of $O(\log_2 n \log_B n)$ expected I/Os each.*

This approach can be extended to higher dimensions. Each new dimension requires a new level of partition tree, and in order to maintain the same approximation error, we must also increase the number of facets in the polyhedron used to approximate the unit sphere. Specifically, we can compute δ-approximate nearest neighbors among moving points in $\mathbb{R}^d$, using an index of size $O(n/\delta^{(d-1)/2})$ blocks, in $O(n^{1/2+\varepsilon}/\delta^{(d-1)/2})$ I/Os per query point. Details will appear in the full paper.

7 Conclusions

In this paper we presented various efficient schemes for indexing moving points in the plane so that queries of type Q1 and Q2 can be answered efficiently. We proposed tradeoffs between the query time and the time spent in updating the indexing scheme as the points move. We also presented an efficient indexing scheme for answering Q3 queries. We conclude by mentioning a few open problems:

1. Develop efficient indexing schemes for points moving along nonlinear trajectories.
2. Develop an efficient indexing scheme for answering exact nearest-neighbor queries.
3. Can the indexing scheme described in Section 5.2 be extended so that Theorem 5.4 hold for any distribution of points and their trajectories.

References

[1] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter, Efficient searching with linear constraints, *Proc. 17th Annu. ACM Sympos. Principles Database Syst.*, 169–178, 1998.

[2] P. K. Agarwal and J. Erickson, Geometric range searching and its relatives, *Advances in Discrete and Computational Geometry* (B. Chazelle, J. E. Goodman, and R. Pollack, eds.), AMS Press, 1999, pp. 1–56.

[3] P. K. Agarwal, J. Erickson, and L. Guibas., Kinetic binary space partitions for intersecting segments and disjoint triangles, *Proc. 9th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 107–116, 1998.

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

[5] L. Arge, V. Samoladas, J. S. Vitter, On two-dimensional indexability and optimal range search indexing, *Proc. 18th Annu. ACM Sympos. Principles Database Syst.*, 346–357, 1999.

[6] L. Arge and J. S. Vitter, Optimal dynamic interval management in external memory, *Proc. 37th Annu. IEEE Sympos. Found. Comput. Sci.*, 560–569, 1996.

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

[8] J. Basch, L. Guibas, and L. Zhang, Proximity problems on moving points, *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, 344–351, 1997.

[9] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, An asymptotic optimal multiversion B-tree, *The VLDB Journal*, 5:264-275, 1996.

[10] S. Chamberlain, Model-based battle command: A paradigm whose time has come, *Proc. 1st Intl. Sympos. Command and Control Research and Technology*, 31–38, 1995.

[11] J. Chomicki and P. Z. Revesz, A geometric framework for specifying spatiotemporal objects, *Proc. 6th Intl. Workshop on Time Representation and Reasoning*, 41–46, 1999.

[12] J. R. Driscoll, N. Sarnak, D.D. Sleator, R. Tarjan, Making Data Structures Persistant, *Journal of Computer and System Sciences*, 38:86–124, 1989.

[13] M. Erwig and M. Schneider, Developments in spatio-temporal query languages, DEXA Workshop, 441–449, 1999.

[14] M. Erwig, R. H. Güting, and M. Schneider, M. Vazirgiannis, Abstract and discrete modeling of spatiotemporal data types, *ACM GIS Symposium*, 131–136, 1998.

[15] V. Gaede and O. Günther, Multidimensional Access Methods, *Computing Surveys*, 30:170–231, 1998.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, External-memory computational geometry, *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, 714–723, 1993.

[17] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, Indexing for data models with constraints and classes, *Journal of Computer and System Sciences*, 52:589–612, 1996.

[18] G. Kollios, D. Gunopulos, and V. J. Tsotras, On indexing mobile objects, *Proc. 18th Annu. ACM Sympos. Principles Database Syst.*, 261–272, 1999.

[19] G. Kollios, D. Gunopulos, and V. J. Tsotras, Nearest neighbor queries in a mobile environment, *Spatiotemporal database management*, 119–134, 1999.

[20] M. Koubarakis and S. Skiadopoulos, Tractable query answering in indefinite constraint databases: Basic Results and applications to querying spatiotemporal information, *Spatio-Temporal Database Management*, 204–223, 1999.

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

[22] J. Nievergelt and P. Widmayer, Spatial data structures: Concepts and design choices, in: *Algorithmic Foundations of GIS* (M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds.), Springer-Verlag, Lecture Notes in Computer Science 1340, 1997.

[23] M.H. Overmars, *The Design of Dynamic Data Structures*, Springer-Verlag, LNCS 156, 1983.

[24] M.H. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. Syst. Sci.* 23:166–204, 1981.

[25] D. Pfoser, Y. Theodoidis, and C. S. Jensen, Indexing trajectories of moving point objects, Chorochronos Tech. Rept. CH-99-3, 1999.

[26] S. Ramaswamy and P. Kanellakis, OODB indexing by class division, *A.P.I.C. Series, Academic Press, New York*, 1995.

[27] S. Ramaswamy and S. Subramanian, Path Caching: A Technique for Optimal External Searching, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 25-35, 1994.

[28] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, Indexing the positions of continuously moving objects, to appear in *Proc. SIGMOD Int. Conf. Management of Data*, 2000.

[29] B. Salzberg and V. J. Tsotras, A Comparison of Access Methods for Temporal Data, TimeCenter Technical Report, TR-13, 1997.

[30] M. Sharir and P. K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge-New York-Melbourne, 1995.

[31] A. P. Sistla and O. Wolfson, Temporal conditions and integrity constraints in active database systems, *Proc. SIGMOD Int. Conf. Management of Data*, 269–280, 1995.

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

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

[34] P.J. Varman and R.M. Verma, An efficient multiversion access structure, *IEEE Trans. on Knowledge and Data Engineering*, 9:391–410, 1997.

[35] J. S. Vitter, Online data structures in external memory, *Proc. 26th Annual International Colloquium on Automata, Languages, and Programming*, LNCS 1644, 119–133, 1999.

[36] O. Wolfson, S. Chamberlain, L. Jiang, and G. Mendez, Cost and imprecision in modeling the position of moving objects, *Proc. Intl Conf. Data Engineering*, 588–596, 1998.

[37] O. Wolfson, L. Jiang, A. P. Sistla, S. Chamberlain, and M. Deng, Databases for tracking mobile units in real time, *Proc. Int. Conf. Database Theory*, 169–186, 1999.

[38] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha, Updating and querying databases that track mobile units, *Distributed and Parallel Databases* 7:257–387, 1999.

[39] O. Wolfson, A. P. Sistla, B. Xu, S. J. Zhou, and S. Chamberlain, DOMINO: Databases for moving objects tracking, *Proc. SIGMOD Int. Conf. Management of Data*, 547–549, 1999.

[40] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, Moving objects databases: Issues and solutions, *Proc. Sympos. Spatial Database Management*, 111–122, 1998.