# Indexing Moving Points*

Pankaj K. Agarwal† Lars Arge‡ Jeff Erickson§

Submitted to *Journal of Computer and System Sciences*: December 13, 2000
Revised and resubmitted: February 6, 2002

**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 any query of the following form can be answered quickly: Given a rectangle $R$ and a real value $t$, report all $K$ points of $S$ that lie inside $R$ at time $t$. We first present an indexing structure that, for any given constant $\varepsilon > 0$, uses $O(N/B)$ disk blocks and answers a query in $O((N/B)^{1/2+\varepsilon} + K/B)$ I/Os, where $B$ is the block size. 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 the query time stamp $t$ and the current time. Finally, we develop an efficient indexing scheme to answer approximate nearest-neighbor queries among moving points.

*An extended abstract of this paper appeared in the Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems [2].

†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–9972879, and CCR–9732787, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, 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–9972879, and CCR–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, for example, one-dimensional range searching is a commonly used operation [27, 41]. Various two-dimensional range-searching problems are crucial for the support of new language features, such as constraint query languages [27] and class hierarchies in object-oriented databases [27]. 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 [25, 36, 43, 7] 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 [1, 13]. 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; either the database has to be continuously updated or a query output will be obsolete. A better approach is to represent the position of a moving object as a function $f(t)$ of time, so that changes in object position do not require any explicit change in the database system. With this representation, the database needs to be updated only when the function $f(t)$ changes, for example, when the velocity of an object changes. Recently there has been some work on extending the capabilities of existing database systems to handle moving-object databases (MOD); see, for example, [45, 46, 15].

This paper focuses 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 or in the past 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 assume that each point $p_i$ is moving at some fixed velocity, 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 stamp $t$, report all points of $S$ that lie inside $R$ at time $t$, *i.e.*, report $S(t) \cap R$; see Figure 1(a).

**Q2.** Given a rectangle $R$ and two time stamps $t_1$ and $t_2$, report all points of $S$ that lie inside $R$ at any time between $t_1$ and $t_2$, *i.e.*, report $\bigcup_{t=t_1}^{t_2} (S(t) \cap R)$; see Figure 1(b). In many applications, either $t_1 = \textit{now}$ or $t_2 = \textit{now}$, e.g., report all objects that were in $R$ in the last ten minutes, or the ones that will be inside $R$ within ten minutes.

**Q3.** Given a query point $\sigma \in \mathbb{R}^2$ and a time stamp $t$, report a *$\delta$-approximate nearest neighbor* of $\sigma$ in $S$ at time $t$, that is, a point $p \in S$ such that $d(\sigma, p(t)) \leq (1 + \delta) \cdot \min_{r \in S} d(\sigma, r(t))$, where $d(\cdot, \cdot)$ is the Euclidean distance; see Figure 1(c).

Our main interest is minimizing the number of disk accesses needed to answer a range query. Consequently, we will consider these problems in the standard external memory model. This model

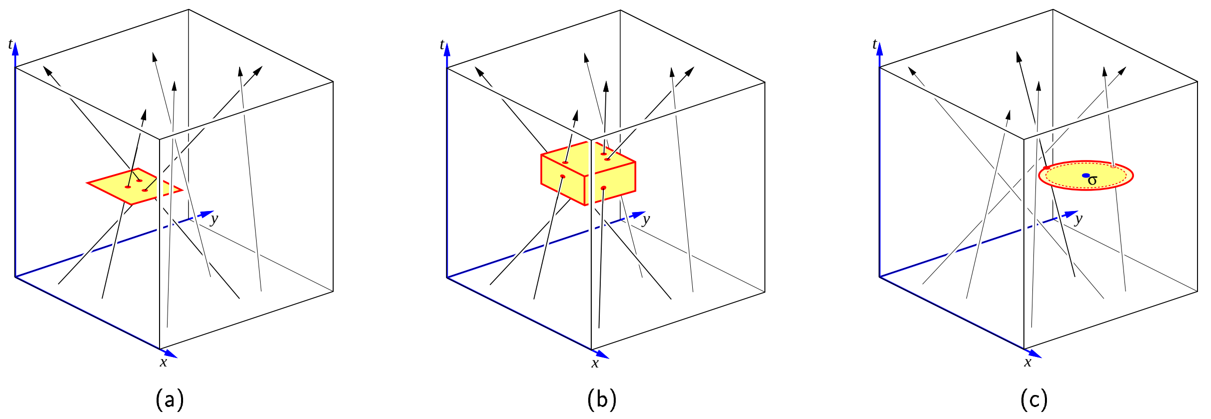

**Figure 1.** Instances of Q1, Q2, and Q3 queries, respectively.

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 $k = \lceil K/B \rceil$. We also assume that the size of internal memory is at least $B^2$.

## 1.2 Previous results

A detailed summary of early work on temporal databases can be found in the survey paper by Salzberg and Tsotras [42]. Most of the 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 [45] developed a temporal query language called *future temporal logic* (FTL) that supports proximity queries on moving objects. Sistla *et al.* [46] later refined FTL and proposed a data model called *moving objects spatio-temporal* (MOST) for moving objects. These models were later extended to incorporate several important issues, including uncertainty in the motion and communication cost [50, 51, 52, 53, 54]. Other spatio-temporal models can be found in [15, 23, 22].

Although a number of methods have been proposed for accessing and searching moving objects which seem to work well in practice [24, 30, 31, 47, 52], they all require $\Omega(n)$ I/Os in the worst case, even if the query range is empty. Kollios *et al.* [29] proposed an efficient indexing scheme, based on partition trees [3, 5], for storing a set of points moving on the real line. It uses $O(n)$ disk blocks, answers a 1-dimensional query of type Q1 or Q2 using $O(n^{1/2+\varepsilon} + k)$ I/Os[1], and allows a point to be inserted or deleted in $O(\log_2^2 n)$ I/Os. They also present a 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. Finally, they propose a data structure that answers queries of type Q1 for points moving in $\mathbb{R}^2$, but it requires $\Omega(n)$ I/Os to answer a query in the worst case. In another paper, Kollios *et al.* [28] proposed a data structure for answering nearest-neighbor queries for moving points on the real line, but it does not provide any nontrivial bound on the time taken to answer a query.

[1]In all time bounds of this form, $\varepsilon$ is an arbitrarily small positive constant. Multiplicative constants hidden by the big-Oh notation depend on $\varepsilon$ and typically go to infinity as $\varepsilon$ goes to zero.

Šaltenis *et al.* [49] propose an indexing scheme based on R-trees for answering queries of type Q1 and Q2; its worst-case query time is also $\Omega(n)$ I/Os. See also [39].

In the computational geometry community, early work on moving points focused on bounding the number of changes in various geometric structures such as convex hulls and Delaunay triangulations as the points move [44]. The notion of *kinetic data structures*, introduced by Basch *et al.* [10], has led to several interesting results related to moving objects, including results on kinetic space partition trees (also known as cell trees) [6]. The main idea in the kinetic framework is that even though the points move continuously, the relevant *combinatorial* structure changes only at certain predictable discrete times. Therefore one does not have to update the data structure continuously. In contrast to fixed-time-step methods, in which the entire structure is updated periodically at a fixed rate determined by the fastest moving object in the database, kinetic data structures are updated *only* when certain *kinetic events* occur. These events have a natural interpretation in terms of the underlying structure, for example, when the trajectory of a point changes or when the $x$- or $y$-projections of two points coincide. Each kinetic update effects only a small part of the overall data structure, so different portions of the structure can change at different rates. All previous work on kinetic data structures has been done in the standard RAM model, which ignores the overwhelming cost of paging large amounts of data.

## 1.3 Our results

This paper contains four main results on indexing moving points in the plane. We first present three indexing schemes for answering Q1 queries, by using two different approaches and then by combining them. The first approach regards time as a third spatial dimension and solves the problem directly in $xyt$-space. In the second approach, based on the work on kinetic data structures, we develop a two-dimensional indexing scheme and describe how to evolve it over time. To our knowledge, the structures we develop are the first linear-size indexing schemes with provable performance bounds for answering range queries on moving points in $\mathbb{R}^2$.

Our paper is organized as follows. 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, answers a query of type Q1 using $O(n^{1/2+\varepsilon} + k)$ I/Os in the worst case. The index can be constructed using $O(N \log_B n)$ I/Os, and the amortized cost of an insertion or deletion is $O(\log_B^2 n)$ expected I/Os.[2] It can also answer queries of type Q2 within the same I/O bound. Finally, we observe that our indexing scheme can handle certain forms of uncertainty in the velocity of points, without affecting the asymptotic performance. Since this index needs to be updated only when the trajectory of a point changes, we call it a *time-oblivious index.*

While the partition tree scheme is time-oblivious, the cost of a query is relatively high. In Sections 4 and 5, we show that by allowing an 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 by applying the kinetic data structure framework to an *external range tree* [8]. Kinetic range trees (with slightly worse performance) were first developed in the internal-memory setting by Basch *et al.* [11]. Our structure uses $O(n \log_B n / (\log_B \log_B n))$ disk blocks. If the points move with fixed velocity, then it processes $O(N^2)$ events, each of which can be processed in $O(\log_B^2 n / \log_B \log_B n)$ I/Os. Our structures works even if the trajectories of the points are polynomials of fixed degree, provided we can compute in $O(1)$ time when the $x$- or $y$-coordinates of two points coincide. We

[2]Unless we explicitly assume a distribution on the input points and their trajectories, the expectation is taken over the outcome of the coin flips performed by the algorithm; the expected bounds hold on any input. By amortized, expected cost of the update time to be $\tau$, we mean that for any $r \geq 1$, the expected cost of performing any sequence of $r$ updates is at most $r\tau$.

also show how to combine kinetic range trees and 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 $\Delta$ such that $NB \le \Delta \le N^2$, we can answer any 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 (or recent past) are more critical than queries far from the present time. 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 6, we propose such an indexing scheme. Using $O(n \log_B n / (\log_B \log_B n))$ disk blocks, it answers any query of type Q1 so that the number of I/Os required is a monotonically increasing function of $|t - \mathit{now}|$. The query time never exceeds $O(n^{1/2+\varepsilon} + k)$. If the points and their trajectories are distributed uniformly at random, then any query takes $O((\Delta_q/n)^{1/2} n^{\varepsilon} + k)$ expected I/Os (here the expectation is taken over the distribution of input points), where $0 \le \Delta_q \le \binom{n}{2}$ is the number of kinetic events in the time interval $[\mathit{now}, t]$ (or $[t, \mathit{now}]$).

Finally, in Section 7, we describe an indexing scheme for answering approximate-nearest-neighbor queries. Given a parameter $\delta > 0$, we construct an indexing scheme based on partition trees that uses $O(n/\sqrt{\delta})$ disk blocks. Then given a query point $\sigma \in \mathbb{R}^2$ and a time stamp $t$, our structure returns a $\delta$-approximate nearest neighbor using $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ I/Os. A point can be inserted or deleted in amortized $O((\log_B^2 n)/\sqrt{\delta})$ expected I/Os.

## 2 Geometric Preliminaries

In order to develop our results, we need a few concepts and results from computational geometry. For further geometric background, we refer the reader to the textbook of de Berg *et al.* [18] or the survey on geometric range searching by Agarwal and Erickson [5]. We assume the reader is familiar with standard randomized algorithms techniques; see Motwani and Raghavan [35] for details.

### 2.1 Duality

Duality is a popular and powerful technique used in geometric algorithms [18]; 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 $(a, b) \in \mathbb{R}^2$ 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 $\sigma^*$ denote the dual of an object (point or line) $\sigma$, and for any set of objects $\Sigma$, let $\Sigma^*$ denote the set of dual objects $\{\sigma^* \mid \sigma \in \Sigma\}$. Note that duality is an involution; for any object $\sigma$, we have $(\sigma^*)^* = \sigma$. 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 $\sigma$ is a vertical line segment $\sigma^*$, in the sense that a point $p$ lies inside $\sigma$ if and only if the dual line $p^*$ intersects $\sigma^*$. See Figure 2.

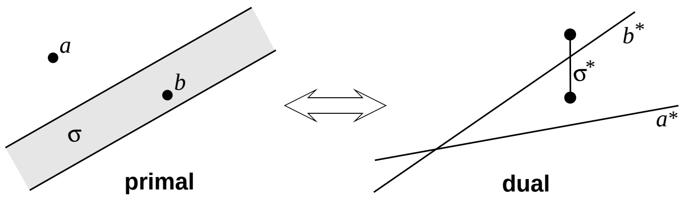

**Figure 2.** The duals of two points and a strip are two lines and a vertical line segment.

## 2.2 External partition trees

Partition trees are one of the most commonly used internal memory data structures for geometric range searching [5, 33]. Our first indexing scheme is based on partition trees, which were originally described in Matoušek [33], and later extended in [3] 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 [3].

In order to describe an algorithm for constructing a partition tree, we need to define two concepts, $(1/r)$-cuttings and simplicial partitions, which are interesting in their own rights. Let $L$ be a set of $N$ lines in the plane. For any integer $r$, a triangulation[3] $\Xi$ of the plane is called a $(1/r)$-*cutting* of $L$ if at most $N/r$ lines intersect the interior of every triangle in $\Xi$. The *size* of a cutting $\Xi$ is just the number of triangles. For a given weight function $w : L \to \mathbb{R}^+$, $\Xi$ is called a *weighted* $(1/r)$-*cutting* if the total weight of the lines intersecting any triangle of $\Xi$ is at most $w(L)/r$, where $w(L)$ is the total weight of all the lines in $L$. The following lemma follows from the known results in computational geometry [14, 33].

**Lemma 2.1.** *Let $L$ be a set of $N$ lines in the plane, let $w : L \to \mathbb{R}^+$ be a weight function, and let $r = O(B)$ be a parameter. A weighted $(1/r)$-cutting of $L$ of size $O(r^2)$ can be computed in expected $O(nr)$ I/Os.*

**Proof:** Matoušek [33] showed that by scaling the weights of lines appropriately, the problem of computing a weighted $(1/r)$-cutting can be reduced to computing a $1/(2r)$-cutting of unweighted lines. We therefore assume that each line in $L$ has weight 1 and we want to compute a $(1/r)$-cutting of $L$. Choose a random subset $R \subseteq L$ of $\alpha r$ lines, where $\alpha$ is a constant; each line is chosen with equal probability. Compute the arrangement $\mathcal{A}(R)$ of $R$ and triangulate each face of $\mathcal{A}(R)$.[4] Let $\mathcal{A}^*(R)$ denote the resulting triangulation. For each triangle $\triangle \in \mathcal{A}^*(R)$, compute the subset $L_\triangle \subseteq L$ of lines that intersect the interior of $\triangle$. A standard random-sampling argument [17, 14] implies that $|L_\triangle| \le (N/r)\log r$ and the expected value of $\sum_\triangle |L_\triangle|$ is $O(Nr)$. Since we assume that the size of the internal memory to be at least $B^2$, that is, that $|A^*(R)|$ is of size $O(M)$, this step can be done in $O(nr)$ I/Os.

For each triangle $\triangle \in \mathcal{A}^*(R)$, if $i(N/r) \le |L_\triangle| \le (i+1)N/r$ for an integer $i \ge 1$, then choose a random subset $R_\triangle \subseteq L_\triangle$ of $\alpha i \log i$ lines, compute the arrangement $\mathcal{A}(R_\triangle)$ within $\triangle$ and compute a triangulation $\mathcal{A}^*(R_\triangle)$ of $\mathcal{A}(R_\triangle)$ within $\triangle$. Set $\Xi = \bigcup_\triangle \mathcal{A}^*(R_\triangle)$. It was shown in [14] that if the constant $\alpha$ is chosen appropriately, then with high probability, each triangle in $\Xi$ intersects at most $N/r$ lines and the expected size of $\Xi$ is $cr^2$, for some constant $c > 1$. The expected number of I/Os needed in this step is also $O(nr)$. If our algorithm creates more than $2cr^2$ triangles, we run the algorithm again. By Markov's inequality the probability of this happening is at most 1/2, so the expected number of times we run the algorithm is at most 2. $\square$

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 (possibly unbounded) 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 $S_i$; see Figure 3. The size of $\Pi$, here denoted $r$, is the number of subset-triangle pairs. A simplicial partition is *balanced* if each subset $S_i$ contains

[3]A triangulation of any domain is a decomposition of that domain into a finite number of (possibly unbounded) triangles, such that the intersection of any two triangles is an edge of both, a vertex of both, or empty.

[4]The *arrangement* of a set $L$ of $N$ lines in the plane, denoted as $\mathcal{A}(L)$, is the planar subdivision whose vertices are the intersection points of lines, edges are the maximal portions of lines not containing any vertex, and faces are the maximal connected portions of the plane not containing any line of $L$. $\mathcal{A}(L)$ can be computed using $O(nN)$ I/Os.

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. The following lemma states how fast a simplicial partition can be constructed. The I/O bound proved here is smaller than the one in [3].

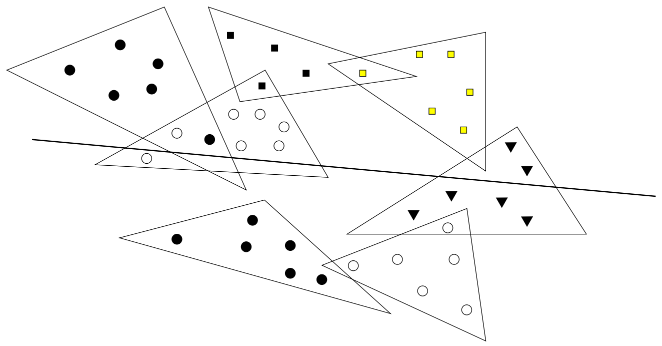

**Figure 3.** A balanced simplicial partition of size 7.

**Lemma 2.2.** *Let $S$ be a set of $N$ points in the plane, and let $r = O(B)$ be a positive integer. A balanced simplicial partition $\Pi$ of size $r$ for $S$ can be constructed in $O(nr)$ expected I/Os, such that the crossing number of $\Pi$ is $O(\sqrt{r})$.*

**Proof:** We construct $\Pi$ using the following simple randomized algorithm. Fix a constant $c > 0$ and compute a $1/(cr)$-cutting of the lines dual to the points in $S$. Let $L$ be the set of lines dual to the vertices of the triangles in the cutting. By Lemma 2.1, $|L| = O(r^2)$ and this step takes $O(nr)$ expected I/Os.

Next, we construct the subset/triangle pairs in $\Pi$ one at a time as follows. Suppose we have already constructed $(S_1, \triangle_1) \dots (S_{i-1}, \triangle_{i-1})$ and we want to compute the next pair $(S_i, \triangle_i)$. Let $S'_i = S \setminus \bigcup_{j=1}^{i-1} S_j$. For each line $\ell \in L$, we maintain the number $x_i(\ell)$ of triangles $\triangle_1, \dots, \triangle_{i-1}$ that intersect $\ell$; set $w_i(\ell) = 2^{x_i(\ell)}$ to be the *weight* of $\ell$. Let $r_i = |S'_i|r/N$ and fix a constant $a > 0$. We compute a $(a/\sqrt{r_i})$-cutting $\Xi$ of the set $(L, w_i)$ of weighted lines. The value of $a$ is chosen so that $\Xi$ has at most $r_i$ triangles. By construction, at least one triangle of $\Xi$ contains at least

$$\frac{|S'_i|}{r_i} = \frac{|S'_i|}{|S'_i|r/N} = \frac{N}{r}$$

points. Let $\triangle_i$ be one such triangle, and let $S_i$ be any $\lceil N/r \rceil$ points in $\triangle_i$. The most time-consuming step of the construction is determining how many points of $S'_i$ lie in each triangle in the arrangement, which requires $O(n)$ I/Os.

Since we iterate the previous process $r$ times, constructing the entire simplicial partition requires $O(nr)$ I/Os. It is obvious that the resulting partition is balanced. Matoušek [33] showed that crossing number of the resulting partition is at most $\alpha\sqrt{r}$, for a constant $\alpha \geq 1$, provided that the constant $c$ is chosen appropriately. □

We are now ready to describe how to construct a partition tree for a set $S$ of points in $\mathbb{R}^2$. Each node $v$ in a partition tree is associated with a subset $S_v \subseteq S$ of points and a triangle $\triangle_v$. For the root of the tree, 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

$$r_v = \min\{cB, 2n_v\}, \tag{2.1}$$

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 crossing number $O(\sqrt{r_v})$ and then recursively construct a partition tree $T_i$ for each subset $S_i$. For each $i$, we store the triangle $\triangle_i$ and a pointer to $T_i$ in $v$; the root of $T_i$ is the $i$th child of $v$, and it is associated with $S_i$ and $\triangle_i$. We need $O(c) = O(1)$ blocks to store any node $v$. Our choice of $r_v$ ensures that every leaf node contains $\Theta(B)$ points. Thus the height of the partition tree is $O(\log_B n)$ and since the tree contains $O(n)$ nodes it use $O(n)$ disk blocks. If we apply Lemma 2.2 recursively to build the entire partition tree, then the total expected construction time is $O(N \log_B n)$ I/Os.

We want to be able to answer queries of the following type: Find all points inside a query strip $\sigma$. 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 $\sigma$. Otherwise, we test each triangle $\triangle_i$ of $\Pi_v$. If $\triangle_i$ lies completely outside $\sigma$, we ignore it; if $\triangle_i$ lies completely inside $\sigma$, we report all points in $S_i$ by traversing the $i$th subtree of $v$; finally, if $\sigma$ crosses $\triangle_i$, we recursively visit the $i$th child of $v$. Note that we spend $O(c)$ I/Os at each node and that each point in $\sigma$ is reported exactly once.

Let $\Sigma_1(N_v)$ denote the number of I/Os required to answer a query starting at node $v$, excluding the $O(K_v/B)$ I/Os used to report the $K_v$ output points. Following the analysis in [3], we find that $\Sigma_1(N_v)$ obeys the following recurrence for some constant $\alpha > 0$ (defined in the proof of Lemma 2.2):

$$\Sigma_1(N_v) \leq \begin{cases} 1 + \alpha\sqrt{r_v} \cdot \Sigma_1\left(\dfrac{2N_v}{r_v}\right) & \text{if } N_v > B, \\ 1 & \text{if } N_v \leq B. \end{cases} \tag{2.2}$$

Let $\varepsilon$ be an arbitrarily small positive constant. By induction on $N_v$, we prove that

$$\Sigma_1(N_v) \leq A_1 n_v^{1/2+\varepsilon} \tag{2.3}$$

for some constant $A_1 > 1$, provided we choose $c = c(\varepsilon)$ sufficiently large in equation (2.1), the definition of $r_v$. Inequality (2.3) is obviously true if $N_v \leq B$. If $B < N_v \leq cB^2/2$, then $r_v = 2n_v$ and

$$\begin{aligned} \Sigma_1(N_v) &\leq 1 + \alpha\sqrt{2n_v} \cdot \Sigma_1(B) \\ &= 1 + \alpha\sqrt{2n_v} \\ &\leq A_1 n_v^{1/2+\varepsilon} \end{aligned}$$

provided that $A_1 > \sqrt{2}\alpha + 1$. Finally, if $N_v > cB^2/2$, and thus $r_v = cB$, then by the induction hypothesis,

$$\begin{aligned} \Sigma_1(N_v) &\leq 1 + \alpha\sqrt{cB} \cdot \Sigma_1\left(\frac{2N_v}{cB}\right) \\ &\leq 1 + \alpha\sqrt{cB} \cdot A_1(2n_v/cB)^{1/2+\varepsilon} \\ &\leq 1 + \frac{2\alpha}{(cB)^\varepsilon} A_1 n_v^{1/2+\varepsilon} \\ &\leq A_1 n_v^{1/2+\varepsilon} \end{aligned}$$

provided we choose $c > (4\alpha)^{1/\varepsilon}$ and $A_1 > 2$. This completes our induction proof. Putting everything together, we conclude that a query can be answered in $O(n^{1/2+\varepsilon} + k)$ I/Os; the constant of proportionality depends on $\varepsilon$, as we spend $O(c)$ I/Os at each of the $O(n^{1/2+\varepsilon})$ nodes visited by the query procedure.

To handle insertions and deletions, we use the *partial rebuilding* technique of Overmars [37]. At each node $v$ in the tree, we store $N_v$, the number of points 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 $\Pi_v$ that contains $p$, and recursively insert $p$ into the corresponding subtree. If more than one triangle in $\Pi_v$ contains $p$, we choose the one whose subtree is smallest. The deletion procedure is similar except that it has to know the leaf of the tree that stores the point to be deleted. This is accomplished by maintaining a separate dictionary that for each point $p$ maintains the index of the leaf containing $p$; see [9] for details.

In order to guarantee the same asymptotic query performance 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 the counter $N_v$ is 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 $O(\log_B^2 n)$ expected I/Os.

**Theorem 2.3.** *Given a set $S$ of $N$ points in $\mathbb{R}^2$ and a parameter $\varepsilon > 0$, we can preprocess $S$ into an external partition tree of size $O(n)$ blocks so that the points inside a query strip can be found 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.*

By exploiting the duality transformation described in Section 2.1, Kollios *et al.* [29] show that two-dimensional external partition trees can be used to efficiently answer range queries for linearly-moving one-dimensional points, as follows. Let $S$ be a set of $N$ moving points in $\mathbb{R}^1$. If we interpret time as a second spatial dimensions, $S$ traces out a set of $N$ lines in the $xt$-plane. Let $P$ denote the set of (static) points dual to these lines. A 1-dimensional Q1 query asks which of the lines traced by $S$ intersect a horizontal line segment, or equivalently, which points in $P$ lie inside a query strip. This query can be answered using a partition tree over $P$ as described above. Using Theorem 2.3 in this framework, we achieve the same query and space bounds as Kollios *et al.* [29], but improve upon their bound of $O(\log_2^2 N)$ I/Os per insertion or deletion.

A query of type Q2 in one dimension — report all points lying in an interval $I$ at any time during the interval $[t_1, t_2]$—is equivalent to reporting all lines that intersect the rectangle $\mathcal{B} = I \times [t_1, t_2]$. We can report all such lines by separately reporting the lines intersecting the bottom, top, and left edges of $\mathcal{B}$, using three separate copies of our two-level partition tree. Note that the same moving point may be reported more than once, but never more than twice. Hence, we obtain the following.

**Corollary 2.4.** *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 Q1 or 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.*

## 2.3 Simplified partition trees for random points

In this section, we describe a simplified version of the partition tree data structure. Although the worst-case query time for this structure is $\Omega(n)$, we expect it to be efficient in most practical situations. To support our intuition, we analyze the expected query time when the points in $S$ are distributed uniformly and independently in some rectangular domain. Without loss of generality, we assume the points all lie in the unit square $\square = [0,1]^2$. Although the above model on distribution of points and their trajectory is not realistic in many applications, the uniform distribution is widely used to analyze the expected performance of various algorithms.

For simplicity, assume that $B = 4^s$ for some integer $s > 0$. We call our simplified data structure a *grid tree*; it is a variant of the so-called hierarchical grid file [36]. At a high level, the grid tree $\mathcal{G}$ is a $B$-ary tree of depth $\log_B n$. Each node $v$ in $\mathcal{G}$ is associated with a square $\square_v$; the root is associated with the original square $\square$. Set $S_v = S \cap \square_v$ and $N_v = |S_v|$. If the depth of $v$ is $\log_B n$ or $N_v \le B$, then $v$ is a leaf. Otherwise, we partition $\square_v$ into $B$ equal squares by drawing a regular $\sqrt{B} \times \sqrt{B}$ grid. Each resulting square is associated with a child of $v$. $\mathcal{G}$ uses $O(n)$ blocks and can be constructed recursively in $O(n \log_B n)$ I/Os. The squares associated with the leaves of $\mathcal{G}$ induce a partition of $\square$. For any leaf $v$, the square $\square_v$ has area $B/N$. Thus, if the points $S$ are uniformly distributed, the expected size of any leaf subset $S_v$ is $B$, and so the expected number of blocks need to store any leaf subset is $O(1)$.

The algorithm for finding the points of $S$ in a strip $\sigma$ is exactly the same as for standard partition trees. We visit $\mathcal{G}$ in a top-down fashion. If $v$ is a leaf, we report all points of $\square_v \cap \sigma$ using $\lceil N_v/B \rceil$ I/Os. At any internal node $v$, we examine whether $\sigma$ intersects $\square_v$. If $\square_v \subseteq \sigma$, then we report all the points in $S_v$; if the boundary of $\sigma$ intersects $\square_v$, then we recursively visit the children of $v$. Since the boundary of $\sigma$ intersects $O(\sqrt{B^i})$ squares associated with nodes of $\mathcal{G}$ at level $i$, the algorithm visits at most $O(\sqrt{n})$ nodes $v$ such that $\square_v$ intersects the boundary of $\sigma$. We examine the points in $S_v$ for such a node only if $v$ is a leaf, in which case the expected size of $S_v$ is $B$. Therefore we spend $O(\sqrt{n})$ I/Os at such nodes. Next, suppose the procedure visits $\mu$ nodes whose squares $\square_v$ are contained in $\sigma$. Since the area of each square $\square_v$ is at least $B/N$, the area of $\sigma$ is at least $\mu B/N$, so the expected size of $\sigma \cap S$ is at least $\mu B$. Consequently, the expected number of I/Os spent in answering a query is $O(\sqrt{n}+k)$. Using Chernoff's bound [35], we can prove that the query time is $O(\sqrt{n}+k)$ with probability at least $1-1/N$.

Finally, similarly to partition trees above, we can make the grid tree fully dynamic using the partial rebuilding technique of Overmars. The expected amortized cost to insert or delete a random point is only $O(\log_B n)$ I/Os, in part because we can directly compute which cell of a grid contains a given point at each node.

**Theorem 2.5.** *Given a set $S$ of $N$ uniformly distributed points in $[0,1]^2$, we can preprocess $S$ into an index of size $O(n)$ blocks so that the points inside a query strip can be found in $O(\sqrt{n}+k)$ I/Os with probability at least $1-1/N$. The index can be constructed in $O(n \log_B n)$ I/Os, and random points can be inserted or deleted at an expected amortized cost of $O(\log_B n)$ I/Os each.*

**Corollary 2.6.** *Given a set $S$ of $N$ linearly moving points in $\mathbb{R}$ whose initial positions and velocities are uniformly distributed in the interval $[0,1]$, we can preprocess $S$ into an index of size $O(n)$ blocks so that a Q1 or Q2 query can be answered in $O(\sqrt{n}+k)$ I/Os with probability at least $1-1/N$. The index can be constructed in $O(n \log_B n)$ I/Os, and points can be inserted or deleted at an expected amortized cost of $O(\log_B)$ I/Os each.*

**Remark.** Unlike the index presented in the previous subsection, the query time of the grid tree is only provably efficient for queries over random point sets. In the worst case, the query time for a grid tree can be $\Omega(n)$ I/Os, even if the query strip is empty.

## 3 Time-Oblivious Indexing

We now describe our first indexing scheme to answer Q1 queries for points in the plane. 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 $R$ parallel to the $xy$-plane. Kollios *et al.* [29] proposed mapping each line to a point in $\mathbb{R}^4$ and using four-dimensional partition trees to answer Q1 queries, but the resulting query time is quite large. Instead, we use a *multilevel* partition tree. Multilevel data structures are a general technique that allows us to answer complex queries by decomposing them into several simpler components and by designing a separate data structure for each component. See [5] for a general discussion of this powerful technique.

Observe that a line $\ell$ 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, as described in Section 2.1. 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 $\sigma^x$ and $\sigma^y$, 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 4.

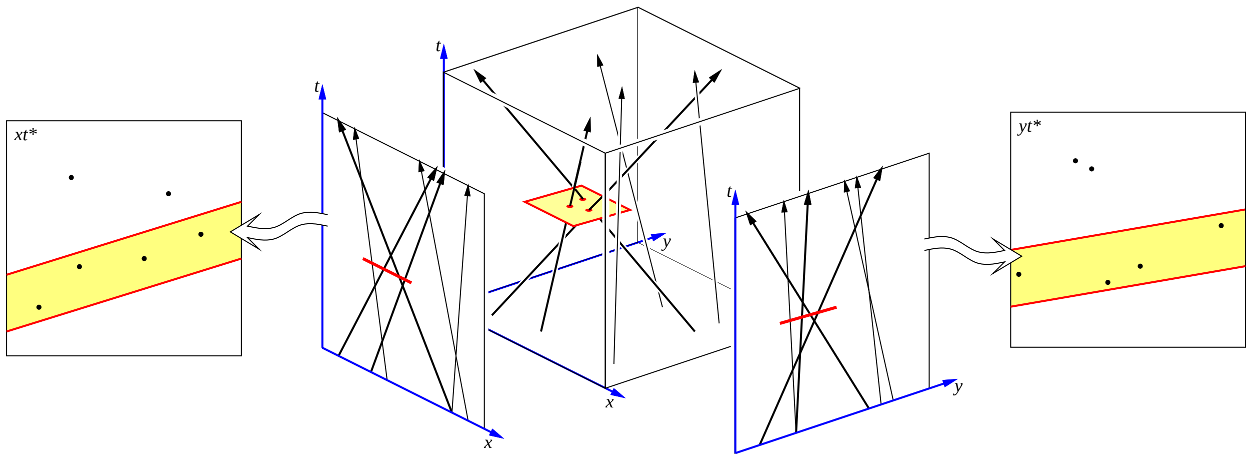

**Figure 4.** Decomposing a rectangle query among moving two-dimensional points into two strip queries among static two-dimensional points, by dualizing the $xt$- and $yt$-projections. A line intersects the rectangle if and only if both corresponding points lie inside the strips.

In the following, we first describe a multilevel partition tree that answers a query efficiently for an arbitrary set of moving points in the plane. We then describe a simpler indexing scheme that works well for uniformly distributed point sets. Finally, we discuss a few extensions of these schemes.

### 3.1 Multilevel partition trees

We construct our multilevel partition tree $\mathcal{T}$ as follows. Let $\delta < 1/2$ be an arbitrarily small positive constant. First, we build a *primary* partition tree $T^x$ for the points $P^x$, where the fanout of each node $v$ is defined as

$$r_v = \min\{n^\delta, cB, 2n_v\}. \tag{3.1}$$

Then at certain nodes $v$ of $T^x$, we attach a *secondary* partition tree $T^y_v$ for the points $S^y_v$. Specifically, if $n^\delta > cB$, we attach secondary trees to every node whose depth is a multiple of $\delta \log_{cB} n$; otherwise, we attach secondary trees to *every* node of $T^x$.[5] In either case, we attach secondary trees to $O(1/\delta) = O(1)$ levels of $T^x$. Each secondary tree $T_v^y$ requires $O(n_v)$ blocks, so the total size of all the secondary trees is $O(n/\delta) = O(n)$ blocks. Moreover, we can construct all the secondary trees in $O(N \log_B n)$ expected I/Os, and using the partial rebuilding technique, we can insert or delete a point in $O(\log_B^2 n)$ amortized expected I/Os. See Figure 5.

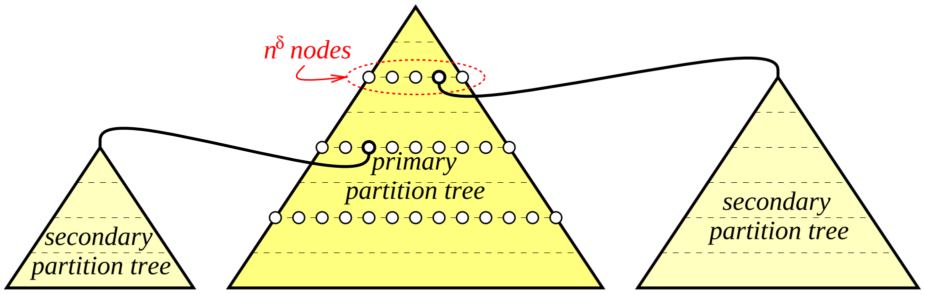

**Figure 5.** Schematic of our multilevel partition tree structure. Each node in certain levels of the primary tree points to a secondary structure. Only two secondary structures are shown.

The algorithm for answering a query is nearly the same as for the basic partition tree. Given two query strips $\sigma^x$ and $\sigma^y$, we first 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 $\sigma^x$, 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 $P_v^y \cap \sigma^y$.

As in Section 2, 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. Similarly, let $\Sigma_2(N_v)$ denote the number of I/Os to answer a query at the multilevel data structure $T_v^x$ over $N_v$ points, excluding the $O(K_v/B)$ output I/Os.

**Lemma 3.1.** $\Sigma_2(N_v) = O(n^\delta n_v^{1/2+\varepsilon})$.

**Proof:** Let $v$ be a node in $T^x$ and let $w$ be one of its proper descendants in $T^x$. We call $w$ an *important descendant* of $v$ if $w$ has a secondary structure $T_w^y$, but no node between $v$ and $w$ has a secondary structure. Each node $v$ has at most $n^\delta$ important descendants, whose secondary structures collectively store all $N_v$ points in $S_v$. Whenever we visit $v$ recursively during a query, we perform secondary queries at a subset of the important descendants of $v$; we perform at most $n^\delta$ secondary queries, each over at most $N_v/n^\delta$ points. Thus, $\Sigma_2(N_v)$ obeys the following recurrence for some constant $\alpha > 0$:

$$\Sigma_2(N_v) \le n^\delta \cdot \Sigma_1\left(\frac{N_v}{n^\delta}\right) + \alpha\sqrt{r_v} \cdot \Sigma_2\left(\frac{2N_v}{r_v}\right).$$

By equation (2.3), this expands to

$$\Sigma_2(N_v) \le A_1 (n^\delta)^{1/2-\varepsilon} n_v^{1/2+\varepsilon} + \alpha\sqrt{r_v} \cdot \Sigma_2\left(\frac{2N_v}{r_v}\right) \tag{3.2}$$

[5]If we always use fanout $cB$ and attach a secondary structure to every node, the resulting data structure uses $O(n \log_B n)$ blocks and answers queries in $O(n^{1/2} N^\varepsilon + k)$ I/Os.

for some constant $A_1$. The base case for the recursion is $N_v < Bn^\delta$, when no descendant of $v$ has an attached secondary structure. In this case, the query algorithm may visit every descendant of $v$, so $\Sigma_2(N_v) = O(n_v) = O(n^\delta)$.

We solve this recurrence similarly to recurrence (2.2) for $\Sigma_1(N_v)$. Specifically, we prove by induction on $N_v$ that

$$\Sigma_2(N_v) \le A_2 n^\delta n_v^{1/2+\varepsilon} \tag{3.3}$$

for some constant $A_2 > 0$, provided we choose $c = c(\varepsilon, \delta)$ sufficiently large in the definition of $r_v$ (equation (3.1)).

First suppose that $cB \le n^\delta$. Inequality (3.3) is obviously true when $N_v < Bn^\delta$, provided $A_2$ is sufficiently large, so assume $N_v \ge Bn^\delta$. In this case, we have $r_v = cB$, so by the induction hypothesis,

$$\begin{aligned}
\Sigma_2(N_v) &\le A_1 (n^\delta)^{1/2-\varepsilon} n_v^{1/2+\varepsilon} + \alpha\sqrt{cB} \cdot \Sigma_2\left(\frac{2N_v}{cB}\right) \\
&\le A_1 n^{\delta/2-\varepsilon\delta} n_v^{1/2+\varepsilon} + \alpha\sqrt{cB} \cdot A_2 n^\delta \left(\frac{2n_v}{cB}\right)^{1/2+\varepsilon} \\
&\le \left(\frac{A_1}{A_2 n^{\delta/2+\varepsilon\delta}} + \frac{2\alpha}{(cB)^\varepsilon}\right) A_2 n^\delta n_v^{1/2+\varepsilon} \\
&\le A_2 n^\delta n_v^{1/2+\varepsilon},
\end{aligned}$$

provided we choose $c > (4\alpha)^{1/\varepsilon}$ and $A_2 > 2A_1$.

On the other hand, suppose that $cB > n^\delta$. Again, equation (3.3) is trivial if $N_v < Bn^\delta$. If $Bn^\delta < N_v \le 2cB^2$, then $r_v = 2n_v$, so

$$\begin{aligned}
\Sigma_2(N_v) &\le A_1 (n^\delta)^{1/2-\varepsilon} n_v^{1/2+\varepsilon} + \alpha\sqrt{n_v}\Sigma_2(B) \\
&= A_1 n^{\delta/2-\delta\varepsilon} n_v^{1/2+\varepsilon} + \alpha\sqrt{n_v} \\
&\le A_1 n^\delta n_v^{1/2+\varepsilon} + \alpha n^\delta n_v^{1/2} \\
&\le A_2 n^\delta n_v^{1/2+\varepsilon},
\end{aligned}$$

provided $A_2 \ge A_1 + \alpha$. Finally, if $N_v > 2cB^2$, then $r_v = n^\delta$, so by the induction hypothesis,

$$\begin{aligned}
\Sigma_2(N_v) &\le A_1 (n^\delta)^{1/2-\varepsilon} n_v^{1/2+\varepsilon} + \alpha\sqrt{n^\delta} \cdot \Sigma_2\left(\frac{2N_v}{n^\delta}\right) \\
&\le A_1 (n^\delta)^{1/2-\varepsilon} n_v^{1/2+\varepsilon} + \alpha\sqrt{n^\delta} \cdot A_2 n^\delta \left(\frac{2n_v}{n^\delta}\right)^{1/2+\varepsilon} \\
&\le \left(\frac{A_1}{A_2 n^{\delta/2+\varepsilon\delta}} + \frac{2\alpha}{(n^\delta)^\varepsilon}\right) A_2 n^\delta n_v^{1/2+\varepsilon} \\
&\le A_2 n^\delta n_v^{1/2+\varepsilon}
\end{aligned}$$

provided $A_2 \ge 2A_1$. This completes our inductive proof. □

**Theorem 3.2.** *Given a set $S$ of $N$ points in $\mathbb{R}^2$, each moving linearly with a fixed velocity, 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 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.*

Recall that 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 $\mathcal{B} = R \times [t_1, t_2]$. As in the one dimensional case, we can report all such lines by separately reporting the lines intersecting the top, bottom, left, right, and front facets of $\mathcal{B}$, using five separate copies of our earlier index. Again, the same moving point may be reported more than once, but never more than twice.

**Theorem 3.3.** *Given a set $S$ of $N$ points in $\mathbb{R}^2$, each moving linearly with a fixed velocity, 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.*

## 3.2 Multilevel grid trees

We can simplify our multilevel structure by using a grid tree, as in Section 2.3. For purposes of analysis, we assume that both the initial position and velocity of each point are chosen uniformly at random from the unit square $[0, 1]^2$.

Our multilevel grid tree consists of a primary grid tree $T^x$ for the points $P^x$ with secondary grid trees $T_v^y$ for the points $P_v^y$ attached to all nodes whose depth is an integer multiple of $\delta \log_{cB} n$. Each node $v$ in the primary grid tree that stores $N_v$ points has fanout $4^s$, where $s = \lceil \log_4 \min\{n^\delta, cB, 2n_v\} \rceil$. If a node $v$ has at most $B$ points or if it is at depth $\log_B n$, $v$ is a leaf. Nodes in the secondary trees have fanout $B$ as before. Unlike our earlier structure, however, each secondary tree is constructed as though it contained *exactly* its expected number of points. By similar arguments as above, our multilevel grid tree uses $O(n/\delta) = O(n)$ blocks of space, can be built in $O(n \log_B n)$ I/Os, and allows random insertions and deletions in $O(\log_B n)$ expected (amortized) I/Os.

A query is answered similarly to a multilevel partition tree. Given two query strips $\sigma^x$ and $\sigma^y$, the query algorithm recursively visits $O(n^{1/2+\delta})$ nodes in the primary tree and possibly searches the secondary structures at these nodes. A secondary query at a node $v$ uses $O(\sqrt{n_v} + K_v/B)$ expected I/Os. Adapting the analysis in the previous subsection, we can prove that the total expected number of expected I/Os required by the query procedure is $O(n^{1/2+\delta} \log_2 n + k)$.

**Theorem 3.4.** *Given a set $S$ of $N$ linearly moving points in $\mathbb{R}^2$ whose initial positions and velocities are uniformly distributed in $[0, 1]^2$, we can preprocess $S$ into an index of size $O(n)$ blocks so that a Q1 query can be answered in $O(n^{1/2+\varepsilon} + k)$ expected 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 $O(\log_B n)$ expected I/Os each.*

**Remark.** We can establish a tradeoff between the size of the multilevel grid tree and the query time by changing the fanout of the primary tree. If we use fanout $r$, the resulting structure uses $O(n \log_r n)$ blocks and answers queries in $O(\sqrt{rn} \log_r n + k)$ expected I/Os. For example, if we use fanout 4 for the primary tree (i.e., $T^x$ is a quad tree), then the resulting structure uses $O(n \log_2 n)$ blocks and answers queries in $O(\sqrt{n} \log_2 n + k)$ expected I/Os.

## 3.3 Further extensions

To conclude this section, we briefly sketch a few extensions of our data structures. As these extensions combine the results of this section with existing techniques from computational geometry [5], we will omit most of the details.

**Higher dimensions.** By building further levels of partition trees, we can generalize our results to any (constant) number of dimensions. In particular, let $S$ be a set of $N$ points in $\mathbb{R}^d$, each moving with a fixed velocity. Each point traces a line in $(d+1)$-dimensional space-time $(x_1, \ldots, x_d, t)$. As earlier, a line $\ell$ intersects an orthogonal $d$-rectangle $R$ in the $(d+1)$-dimensional space if and only if their projections onto the two-dimensional $x_i t$-plane intersect for all $1 \leq i \leq d$. We therefore map each point $p \in S$ to a two-dimensional point $p^i$ in the dual $x_i t$-plane, and map a query rectangle $R$ to a strip $\sigma^i$ in the dual $x_i t$-plane. A point $p(t)$ lies in $R$ if and only if $p^i \in \sigma^i$ for every $1 \leq i \leq d$. We therefore construct a $d$-level partition tree, one for each $x_i t$-plane, and proceed as in the two-dimensional case. The asymptotic query time remains $O(n^{1/2} + k)$ I/Os, but the constant of proportionality increases exponentially with dimension. A Q2 query can also be answered in an analogous manner.

**Theorem 3.5.** *Given a set $S$ of $N$ points in $\mathbb{R}^d$, each moving with a fixed velocity, and a parameter $\varepsilon > 0$, we can preprocess $S$ into an index of size $O(n)$ blocks so that a Q1 or 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.*

**Higher-degree motion.** Returning to the two-dimensional case, we can modify our external partition tree to handle algebraic motion of higher degree, by using the so-called linearization technique. Suppose the position of each point $p \in S$ is given by a polynomial of degree $D$ in the time parameter $t$. Each point now traces out an algebraic curve in $xyt$-space-time. As in the case of linear motion, a curve $\gamma$ intersects a horizontal rectangle $R$ if and only if their projections onto the $xt$-plane and $yt$-plane both intersect. We map any polynomial curve $\gamma$ with equation $x = a_0 + a_1 t + a_2 t^2 + \ldots a_D t^D$ to the point $\gamma^* = (a_1, \ldots, a_D, -a_0) \in \mathbb{R}^{D+1}$ and a point $p = (\alpha, \beta) \in \mathbb{R}^2$ to the hyperplane $p^* : x_{d+1} = -\beta + \sum_{i=1}^{D} \alpha^i x_i$ in $\mathbb{R}^{D+1}$. A vertical segment $\sigma = pq$ maps to a slab $\sigma^*$ in $\mathbb{R}^{D+1}$, bounded by the hyperplanes $p^*$ and $q^*$. Then $\gamma$ intersects a vertical line segment $\sigma$ if any only if the point $\gamma^* \in \sigma^*$. Thus, we can reduce a query of type Q1 to a pair of slab queries among static points in $\mathbb{R}^{D+1}$.

To answer high-dimensional slab queries, we can construct an external partition tree based on cuttings, exactly as in the two-dimensional case. However, the performance of the data structure deteriorates with the dimension. Combining our earlier analysis with techniques of Matoušek [33], as described in [3], we can show that the query time for a $(D+1)$-dimensional slab query is $O(n^{1-1/(D+1)+\varepsilon} + k)$ I/Os. Finally, we can combine these higher-dimensional partition trees into a multilevel data structure exactly as we did for linear motion.

**Theorem 3.6.** *Given a set $S$ of $N$ points in $\mathbb{R}^2$, each moving according to a polynomial funcion of degree $D$, and a parameter $\varepsilon > 0$, we can preprocess $S$ into an index of size $O(n)$ blocks so that a Q1 or Q2 query can be answered in $O(n^{1-1/(D+1)+\varepsilon} + k)$ 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.*

**Uncertainty.** Returning to linear motion in the plane, the external partition tree can also handle certain forms of uncertainty in the velocity, without affecting its asymptotic performance. More precisely, let $\delta$ be an uncertainty parameter, defined as follows. Suppose a moving point in the plane is represented as $p(t) = at + b$, where $a = (a_x, a_y)$ is its velocity and $b = (b_x, b_y)$ is its initial position. Because of uncertainty, the actual position of point $p$ at time $t$ is $\overline{p}(t) = \overline{a}t + b$, where $\overline{a} = (\overline{a}_x, \overline{a}_y)$, $\overline{a}_x \in [a_x - \delta, a_x + \delta]$, and $\overline{a}_y \in [a_y - \delta, a_y + \delta]$. The actuial velocity vector $\overline{a}$ could change with time, but the uncertainty $\delta$ is fixed. If we denote by $\rho$ the rectangle of size $2\delta$ centerted at the origin, then $\overline{p}(t) \in \rho t + p(t)$. Set $\gamma_p(t) = \rho t + p(t)$. We now restate a Q1 query as follows:

**Q1′.** Given an axis-aligned rectangle $R$ in the $xy$-plane and a time value $t$, report all points of $S$ that potentially lie inside $R$ at time $t$, i.e., all points $p \in S$ such that $\gamma_p(t)$ intersects $R$.

We again work in the dual $xt$- and $yt$-planes. For a rectangle $R$, define the strips $\sigma^x$ and $\sigma^y$ as earlier. For a point $p \in S$, we define two intervals $I_p^x = [(a_x - \delta, b_x), (a_x + \delta, b_x)]$ and $I_p^y = [(a_y - \delta, b_y), (a_y + \delta, b_y)]$ in the dual $xt$- and $yt$-planes, respectively. We want to report $p$ if $I_p^x$ intersects $\sigma^x$ and $I_p^y$ intersects $\sigma^y$. Let $\ell_-^x$ (resp. $\ell_+^x$) be the halfplane lying to the right (resp. left) of the left (resp. right) boundary of $\sigma^x$. Then $I_p^x$ intersects $\sigma^x$ if and only if $(a_x - \delta, b_x) \in \ell_+^x$ and $(a_x + \delta, b_x) \in \ell_-^x$. In other words, the condition "$\sigma^x$ intersecting $I_p^x$" can be stated as the conjunction of two halfspace containment conditions. Similarly, we can state the condition "$I_p^y$ intersecting $\sigma^y$" as the conjunction of two halfspace containments. We construct a 4-level partition tree in which each level of the tree filters out the points that satisfy one of the halfspace containment conditions. Omitting further details, we conclude the following.

**Theorem 3.7.** *Given a set $S$ of $N$ points in $\mathbb{R}^2$, each moving linearly with a fixed velocity, an uncertainty parameter $\delta$, 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 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 One-Dimensional Chronological Queries

In the next two sections, we describe how to significantly improve the query time 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.* We develop our results in the *kinetic data structure* framework of Basch *et al.* [10, 11]. The main idea is to store 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 sorted order of points along $x$- and $y$-axes) 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.

In Section 4.1, we develop a data structure called the *kinetic B-tree* to efficiently answer current Q1 queries for moving one-dimensional points. In Section 4.2, we discuss a tradeoff between the query time and the total cost of maintaining the data structure. Section 5 extends these results to the more difficult two-dimensional case.

### 4.1 Kinetic B-trees

Let $S = \{p_1, \ldots, p_n\}$ be a set of $n$ points in $\mathbb{R}$, each moving with a fixed velocity, and let $S(t)$ denote the point set at time $t$. We store $S$ in sorted order in a B-tree $\mathcal{T}$, which is updated periodically so that at all times $t$, $\mathcal{T}(t)$ is a valid B-tree for $S(t)$. If $\mathcal{T}$ is valid at time $t$, then it remains valid as long as the ordering of points in $S$ does not change, so $T$ has to be updated only when the ordering of points changes. We therefore define the *events* to be time instances $t_{ij}$ at which $p_i(t_{ij}) = p_j(t_{ij})$. For every adjacent pair of points $p_i, p_j$ in $S(t)$, we store the time $t_{ij}$ in an external priority queue $\mathcal{B}$, so that insert, delete, and delete-min operations on $\mathcal{B}$ can be performed in $O(\log_B n)$ I/Os. Let $t^* = t_{ij}$ be the minimum value stored in $\mathcal{B}$; the current tree remains valid for the interval $[now, t^*)$. At $now = t^*$, we delete $t^*$ from $\mathcal{B}$ and swap $p_i$ and $p_j$ in $\mathcal{T}$. If the ordering of $S$ before the swap was $\ldots, p_a, p_i, p_j, p_b, \ldots$, then $(p_a, p_i)$ and $(p_j, p_b)$ are no longer adjacent pairs after the swap, so we delete $t_{ai}$ and $t_{jb}$ from $\mathcal{B}$. We also insert $t_{aj}$ and $t_{ib}$ because $(p_a, p_j)$ and $(p_i, p_b)$ now become adjacent pairs. We thus spend $O(\log_B n)$ I/Os at each event. A trajectory change can be handled in $O(\log_B n)$ I/Os in a similar way. A query at current time is answered using $O(\log_B n + k)$ I/Os in a straightforward manner.

**Lemma 4.1.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$. We can store $S$ in an index of size $O(n)$ blocks, so that a current range query can be answered in $O(\log_B n + k)$ I/Os. The cost of any kinetic event or trajectory change is $O(\log_B n)$ I/Os. If the trajectories of the points do not change, there are at most $\binom{N}{2}$ events.*

**Remark.** The kinetic B-tree works within the same I/O bounds even if the points move along more complex trajectories, provided we can quickly compute when two moving points become equal. For rational motion of fixed degree there are still $O(N^2)$ events, where the constant depends on the degree of the motion.

## 4.2 Query/update tradeoffs

If we are going to perform only a few queries, it is inefficient to spend $O(N^2 \log_B n)$ I/Os evolving the kinetic B-tree through $O(N^2)$ events. We can combine the kinetic B-tree with the external partition tree to obtain a tradeoff between the cost of answering queries and the total number of kinetic events. Our construction is similar to a general technique used to establish tradeoffs between data structure size and query time [5].

Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$. 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 points corresponding to $P_v$; set $|S_v| = N_v$.

Let $\Delta$ be a parameter between $B^2N$ and $\binom{N}{2}$. We discard all nodes of $\mathcal{T}$ whose parents are associated with at most $\Delta/N$ points. By equation (2.1), every remaining node in $\mathcal{T}$ has fanout $r = cB$. The depth of the truncated tree is at most $\ell = \log_{r/2}(N^2/\Delta)$. For each leaf $v$ of the truncated tree, we construct a kinetic B-tree on $S_v$. See Figure 6. The total number of events processed by all the kinetic range trees is at most $\sum_v N_v^2$, where the summation is taken over all the leaves of the truncated tree. Since $N_v \leq \Delta/N$ and $\sum_v N_v \leq N$, we obtain $\sum_v N_v^2 = O(\Delta)$.

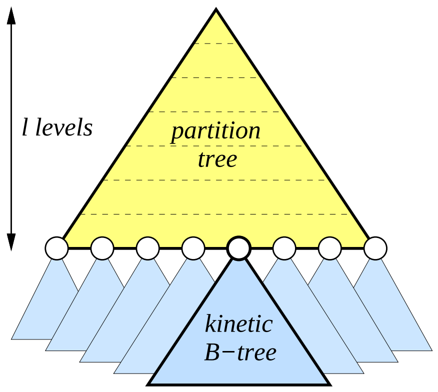

**Figure 6.** Schematic picture of our one-dimensional query/update tradeoff structure.

A query is answered in the same way as earlier. Let $\tilde{\Sigma}_1(N_v)$ by the number of I/Os need to answer a query excluding the $O(K/B)$ I/Os needed to report the points. We obtain the following recurrence for $\tilde{\Sigma}_1(N_v)$, similar to recurrence (2.2).

$$\tilde{\Sigma}_1(N_v) \leq \begin{cases} 1 + \alpha\sqrt{r} \cdot \tilde{\Sigma}_1\left(\frac{2N_v}{r}\right) & \text{if } N_v > \Delta/N, \\ O(\log_B n_v) & \text{if } N_v \leq \Delta/N. \end{cases} \tag{4.1}$$

Since the depth of the truncated partition tree is at most $\ell$, the query procedure visits at most $(\alpha\sqrt{r})^\ell$ leaves of the truncated tree, and it spends $O(\log_B n)$ I/Os at each such node. Hence, the total number of I/Os used is

$$(\alpha\sqrt{r})^{\log_{r/2}(N^2/\Delta)} \log_B n = \left(\frac{N}{\sqrt{\Delta}}\right)^{1+\log_{r/2} 2\alpha^2} \log_B n \leq \left(\frac{N}{\sqrt{\Delta}}\right)^{1+\varepsilon},$$

provided that the constant $c$ is chosen so that $r = cB > 2(2\alpha)^{1/\varepsilon}$. Hence, we conclude the following.

**Theorem 4.2.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$. Given a parameter $\Delta$ such that $BN \leq \Delta \leq \binom{N}{2}$, we can store $S$ in an index of size $O(n)$ blocks, so that a current range query can be answered in $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os. The amortized cost of a kinetic event is $O(\log_B(\Delta/N))$ I/Os, and the amortized cost of inserting or deleting a point is $O(\log_B^2(N^2/\Delta) + \log_B(\Delta/N))$ expected I/Os. If the trajectories of the points do not change, there are $O(\Delta)$ events.*

If know in advance how many queries we need to answer, we can balance the number of I/Os required to answer the queries with the number of I/Os required to maintain the data structure. For $q$ queries, we get the minimum total number of I/Os when $\Delta \approx N^{2/3}q^{2/3}$.

**Corollary 4.3.** *Given a set $S$ of $N$ linearly moving points in $\mathbb{R}$ and a parameter $q$, we can answer any chronological sequence of $q$ range queries using $O(N^{2/3+\varepsilon}q^{2/3} + k)$ I/Os and $O(n)$ blocks.*

## 4.3 Tradeoffs for grid trees

If the initial positions and velocities of the points are uniformly distributed, we can obtain similar tradeoffs by using grid trees instead of partition trees. Given a parameter $BN \leq \Delta \leq \binom{N}{2}$, set $g = \binom{N}{2}/\Delta$. Our structure consists of a grid tree up to level $\ell = \log_B g$. At each leaf $v$, if $|S_v| > B$, we construct a kinetic B-tree on $S_v$. The truncated grid tree has $O(g) = O(N^2/\Delta)$ leaves, each associated with a point set of expected size $O(\Delta/N)$. Two points lie in the same cell of $\mathcal{G}$ with probability $1/g$, so the expected number of kinetic events for the entire data structure is at most $\Delta$. The query algorithm recursively visits $O(n/\sqrt{\Delta})$ nodes of the truncated grid tree, and the expected search time for any of the kinetic B-trees is $O(\log_B(\Delta/N))$ I/Os, so the total expected query time is $O((N/\sqrt{\Delta})\log_B(\Delta/N) + k)$ I/Os.

**Theorem 4.4.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$, whose initial positions and velocities are uniformly distributed in $[0, 1]$. Given a parameter $\Delta$ such that $BN \leq \Delta \leq \binom{N}{2}$, we can store $S$ in an index of size $O(n)$ blocks, so that a current range query can be answered in $O((N/\sqrt{\Delta})\log_B(\Delta/N) + k)$ expected I/Os. The expected amortized cost of a kinetic event or trajectory change is $O(\log_B(\Delta/n))$ I/Os. If the trajectories of the points do not change, the expected number of events is $\Delta$.*

## 5 Two-Dimensional Chronological Queries

This section extends the results of Section 4 to the more complex case of moving points in the plane. Our so-called *kinetic external range tree* is based on a kinetic range tree developed by Basch *et al.* [11] and an external range tree with optimal query cost developed by Arge *et al.* [8]. We give a brief overview of the external range tree in Section 5.1, and then discuss how to *kinetize* it in Section 5.2. Finally, in Section 5.3, we discuss a query update tradeoff similar to the one-dimensional case.

### 5.1 External range trees

The external range tree $\mathcal{K}$ presented in [8] is a three-level structure. The primary structure is a tree over the $x$-coordinates of the $N$ points in $S$, similar to a B-tree, with fan-out $\log_B n$. An $x$-range $X_v$ is associated with each node $v$ in the tree in a natural way, and this range is subdivided into $\log_B n$ *slabs* by $v$'s children $v_1, v_2, \ldots, v_{\log_B n}$. We store the $x$-coordinates of slab boundaries in a $B$-tree so that the slab containing a query point can be determined in $O(\log_B \log_B n)$ I/Os. Let $S_v \subseteq S$ be the set of points whose $x$-coordinates lie in the $x$-range $X_v$; set $N_v = |S_v|$. $S_v$ is stored in four secondary data structures associated with $v$. One of the structures is a B-tree $\mathcal{B}_v$ on $S_v$, using the $y$-coordinates of its points as the keys. The three other structures are external versions of the *priority search tree* [34]. An external priority search tree is used to answer three-sided range queries, i.e., reporting the points lying in a rectangle of the form $[a, b] \times [c, \infty)$, in $O(\log_B n + k)$ I/Os. We discuss priority search trees in detail in the next subsection.

The first two priority search trees, $\mathcal{P}^{\sqsupset}(v)$ and $\mathcal{P}^{\sqsubset}(v)$, store the points in $S_v$ such that queries of the forms $(-\infty, a] \times [b, c]$ and $[a, \infty) \times [b, c]$ can be answered in $O(\log_B n + k)$ I/Os. The third priority search tree $\mathcal{P}^{\div}(v)$ stores points derived from the $y$-coordinates of the points in $S_v$ as follows. Let $p = (x_p, y_p) \in S_v$ be a point lying in the $j$th slab (i.e., $p \in S_{v_j}$). If $p$ is not the point with the maximum $y$-coordinate in $S_{v_j}$, then let $q = (x_q, y_q) \in S_{v_j}$ be the successor of $p$ in the $+y$-direction. We map $p$ to the point $p^* = (y_p, y_q)$. Let $S_v^*$ be the resulting set of points. We construct $\mathcal{P}^{\div}(v)$ so that all $t$ points lying in a range of the form $(-\infty, a] \times [a, \infty)$ can be reported in $O(\log_B n + t/B)$ I/Os. Note that a point $p^* = (y_p, y_q)$ lies in such a range if and only if the $y$-interval $[y_p, y_q]$ intersects the horizontal line $y = c$; see Figure 7. Since there is only one such interval within each slab, we have $t = \log_B n$. For each point $p^*$ stored in $\mathcal{P}^{\div}(v)$, we also store a pointer to the leaf of $\mathcal{B}_{v_j}$ that stores the corresponding point $p$ of $S_{v_j}$. Since external priority search trees and B-trees use linear space [8], and each point $p$ is stored in secondary structures of all the $O(\log_{\log_B n} n) = O(\log_B n / (\log_B \log_B n))$ nodes on the path from the root to the leaf storing the $x$-coordinate of $p$, the structure use $O(n \log_B n / (\log_B \log_B n))$ blocks in total.

External range trees can be used to find the points inside a query rectangle $q = (a, b, c, d)$ in $O(\log_B n + k)$ I/Os as follows [8]. We first find, in $O(\log_B n / (\log_B \log_B n)) \times O(\log_B \log_B n) = O(\log_B n)$ I/Os, the highest node $v$ in $\mathcal{K}$ so that $a$ and $b$ lie in different slabs of $v$. Suppose $a$ lies in the $x$-range of $v_i$ and $b$ in the $x$-range of $v_j$. The query rectangle $q$ is naturally decomposed into three parts: $q^{\sqsubset} = ([a, b] \cap X_{v_i}) \times [c, d]$, $q^{\sqsupset} = ([a, b] \cap X_{v_j}) \times [c, d]$, and $q^{\div} = q \setminus (q^{\sqsupset} \cup q^{\sqsubset})$. See Figure 7. The points contained in $q^{\sqsubset}$ and $q^{\sqsupset}$ can be reported in $O(\log_B n + k)$ I/Os using $\mathcal{P}^{\sqsubset}(v_i)$ and $\mathcal{P}^{\sqsupset}(v_j)$, respectively. To report the points in $q^{\div}$, we first query $\mathcal{P}^{\div}(v)$ with $(-\infty, c] \times [c, \infty)$ to find the lowest point in each $S_{v_l}$, for $i < l < j$, that lies in $q^{\div}$ (and thus the pointer to the corresponding point in $\mathcal{B}_{v_l}$). Using the $B$-trees $\mathcal{B}_{v_l}$, we report all the points of $S_{v_l} \cap q^{\div}$. The total number of I/Os needed is $O(\log_B n + k)$ I/Os.

To perform an update on the external range tree we need to perform $O(1)$ updates on the secondary structures on each of the $O(\log n / (\log_B \log_B n))$ levels of the base tree. These updates

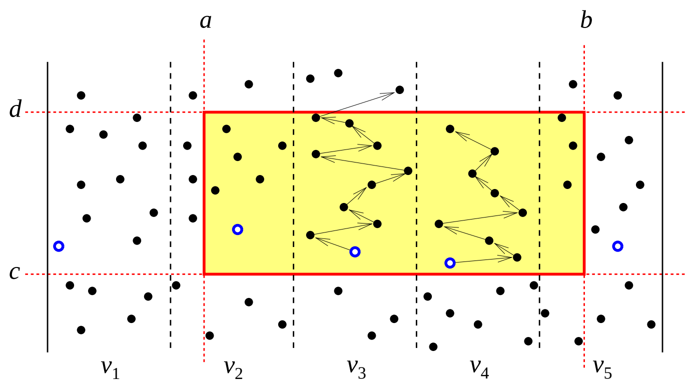

**Figure 7.** Slabs corresponding to a node $v$ in the primary tree. To find the points in the rectangle, we answer three-sided queries using $\mathcal{P}^{\sqsubset}(v_2)$ and $\mathcal{P}^{\sqsupset}(v_5)$, find the lowest point in each slab above the bottom of the rectangle using $\mathcal{P}^{\div}(v)$, and then walk upwards through the points inside slabs $v_3$ and $v_4$.

take $O(\log_B N)$ I/Os each, since the external priority search tree can be updated in $O(\log_B n)$ I/Os. One also needs to update the primary structure. Arge *et al.* [8] discuss how this can also be done in $O(\log_B^2 n/(\log_B \log_B n))$ amortized I/Os using a weight-balanced B-tree [9].

**Lemma 5.1 (Arge *et al.* [8]).** *A set of $N$ points in $\mathbb{R}^2$ can be stored in an index using $O(n \log_B n/(\log_B \log_B n))$ 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 $O(\log_B^2 n/\log_B \log_B n)$ I/Os each.*

**External priority search trees.** We now discuss the linear space external priority search tree for answering queries of the form $[a, b] \times [c, \infty)$ on a set $S$ of $N$ points in $O(\log_B n + k)$ I/Os. As in the range tree discussed above, the structure consists of a base B-tree on the $x$-coordinates of the points in $S$. The fanout of the tree is $B$. As above, each internal node $v$ is associated with an $x$-range $X_v$, which is divided into slabs by the $x$-ranges of its children. For each child $v_i$ of $v$, we store the highest $B$ points in the corresponding slab (if any) that have not been stored in ancestors of $v$. We store these $O(B^2)$ points in an auxiliary *catalog structure* $\mathcal{A}_v$ that uses $O(B)$ blocks and supports three-sided queries and updates in $O(1 + k)$ I/Os. We will describe the catalog structure in detail below. Since every point is stored in precisely one catalog structure, the external priority search tree can be stored in $O(n)$ blocks.

To answer a three-sided query $q = (a, b, c)$, we start at the root of the external priority search tree and proceed recursively to the appropriate subtree. At each node $v$, we first query the catalog structure $\mathcal{A}_v$ to report the points of $A_v \cap q$. If $a$ or $b$ lies in $X_{v_i}$ or if the $x$-coordinates of at least $B$ points of $A_v \cap q$ lie in $X_{v_i}$, we recursively query at $v_i$. See Figure 8. The query procedure visits only $O(\log_B n)$ nodes $v$ because $a$ or $b$ lies in $X_v$. For every other node $v$ visited by the query procedure, it reported at least $B$ points at the parent of $v$ whose $x$-coordinates lie in $X_v$. It follows that the query procedure reports all $K$ points in $q$ in $O(\log_B n + k)$ I/Os.

Using the fact that a catalog structure can be updated in $O(1)$ I/Os, Arge *et al.* [8] showed how to update the external priority search tree in $O(\log_B n)$ I/Os and thus obtained the following.

**Lemma 5.2 (Arge *et al.* [8]).** *A set of $N$ points in $\mathbb{R}^2$ can be stored in an external priority search tree of size $O(n)$ blocks, so that a three-sided range query can be answered in $O(\log_B n + k)$ I/Os. Points can be inserted or deleted using $O(\log_B n)$ I/Os.*

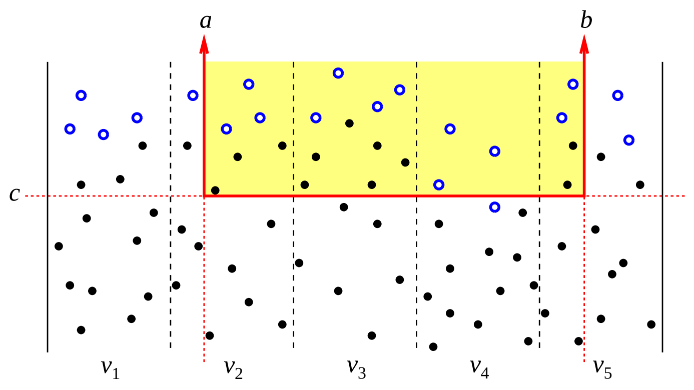

**Figure 8.** Slabs corresponding to a node $v$ in a priority search tree. The highest $B$ points in each slab (shown here in white) are stored in the catalog structure $\mathcal{A}_v$; the other (black) points are stored recursively. To answer the query, we query $\mathcal{A}_v$ to find the relevant white points and then recursively query in $v_2$, $v_3$, and $v_5$ for the relevant black points.

**Catalog structure.** We now describe the so-called *catalog structure* for answering three-sided range queries on a set $S$ of $B^2$ points using $O(1+k)$ I/Os. Let $S$ consist of points $(x_i, y_i)$ sorted in increasing $x$-coordinate order. The catalog structure $\mathcal{A}$ consists of $2B-1$ blocks $b_1, b_2, \ldots, b_{2B-1}$ storing the points, plus a constant number of *catalog blocks*. We associate a rectangle $[x_{l_i}, x_{r_i}] \times [y_{d_i}, y_{u_i}]$ with each block $b_i$; the catalog blocks store these $2B-1$ rectangles. Block $b_i$ contains a point $(x_j, y_j) \in S$ if and only if the point lies inside or directly above the block's rectangle:

$$x_{l_i} \le x_j \le x_{r_i} \quad \text{and} \quad y_{d_i} \le y_j. \tag{5.1}$$

The blocks $b_i$ are constructed as follows. Initially, we create $B$ blocks $b_1, b_2, \ldots, b_B$. For each $1 \le i \le B$, the rectangle of $b_i$ has left $x$-coordinate $x_{(i-1)B+1}$, right $x$-coordinate $x_{iB}$, and bottom $y$-coordinate $-\infty$. 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 $y = y_j$, the top $y$-coordinate of the rectangles 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. Instead, a new block $b_r$ is created whose rectangle has left $x$-coordinate $x_{l_i}$, right $x$-coordinate $x_{r_{i+1}}$, and bottom $y$-coordinate $y_j$. Thus, $b_r$ contains the (at most $B$) points of $b_i$ and $b_{i+1}$ that lie above the line $y = y_j$. The sweep continues in this manner until the line reaches $+\infty$, at which point at most $B + (B-1) = 2B-1$ blocks have been created. See Figure 9(a). The entire catalog structure can be constructed in $O(B)$ I/Os [8].

To answer a three-sided query $(a, b, c)$, we first load the catalog blocks into main memory using $O(1)$ I/Os. We then identify all blocks whose rectangle intersects the bottom edge $[a, b] \times c$ of the query range. We load these blocks into main memory one at a time and report the relevant points. The query takes $O(1+k)$ I/Os since every consecutive pair of blocks, except possibly for the blocks containing $a$ and $b$, contributes at least $B/2$ points to the output. See Figure 9(b).

The structure can easily be made dynamic using global rebuilding [37]. Updates are simply logged in an extra block $\mathcal{U}$. After $B$ updates, when $\mathcal{U}$ 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 $\mathcal{U}$ to ensure that query results are consistent with recorded updates.

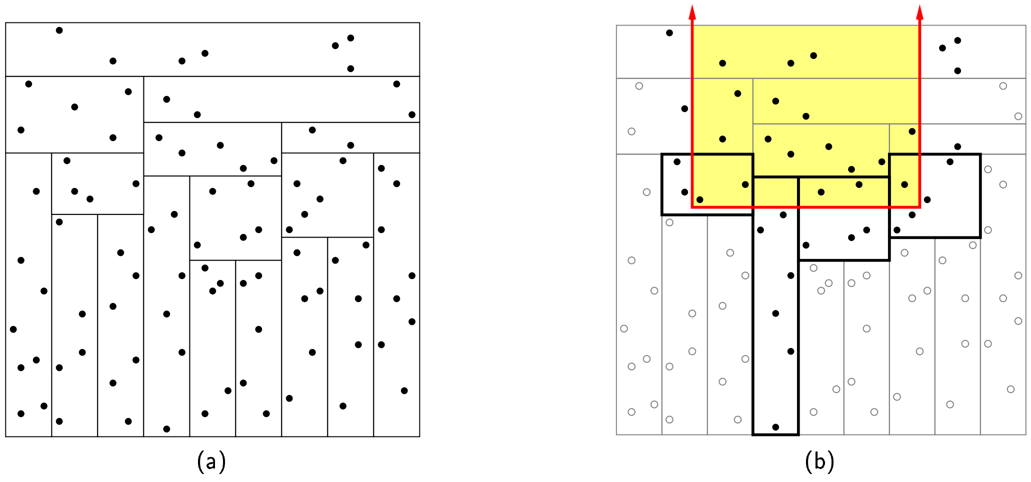

**Figure 9.** (a) An example catalog structure with $B = 10$. Each block contains the points inside or above its rectangle. (b) The blocks loaded by the query algorithm are indicated by the bold rectangles and contain the solid points.

**Lemma 5.3 (Arge *et al.* [8]).** *A set of $B^2$ points can be stored in an index of size $O(B)$ blocks, so that a three-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.*

## 5.2 Kinetic external range trees

We now discuss how to *kinetize* the external range tree, so that it can store moving points and quickly answer range queries at the current time. We explain the necessary modifications from the bottom up—first for the catalog structure $\mathcal{A}$, then for the external priority search tree $\mathcal{P}$, and finally for the top-level structure. Our techniques are similar to the method used by Basch *et al.* [11] to kinetize internal-memory range trees.

First consider the catalog structure $\mathcal{A}$. Recall that the rectangle of each block $b_i$ in $\mathcal{A}$ was defined by four points of $S$. In the case of moving points, we define the rectangle of $b_i$ at time $t$ to be $[x_{l_i}(t), x_{r_i}(t)] \times [y_{d_i}(t), y_{u_i}(t)]$. Thus the rectangle of each block changes continuously with time. However, a point $p_j(t) = (x_j(t), y_j(t))$ in $b_i$ continues to satisfy condition (5.1) until some time $t$ when $x_j(t) = x_{l_i}(t)$, $x_j(t) = x_{r_i}(t)$, or $y_j(t) = y_{d_i}(t)$. We can thus continue to use $\mathcal{A}$ to answers queries until time $t$, at which we will have to update the structure.

To detect exactly when condition (5.1) is violated, we maintain two kinetic B-trees $\mathcal{B}_x$ and $\mathcal{B}_y$ over the $x$- and $y$-coordinates of the points in $S$, respectively, with a common event queue $\mathcal{Q}$. Recall that a kinetic B-tree undergoes a *swap* event when two of its values become equal, so we observe a swap event whenever two points $p_j(t)$ and $p_{j'}(t)$ have the same $x$- or $y$-coordinate. At each swap event, we check whether condition (5.1) still holds for $p_j(t)$ and $p_{j'}(t)$; if not, we simply remove the offending point from $\mathcal{A}$ and reinsert it using $O(1)$ I/Os (Lemma 5.3). In total we handle a swap event in $O(\log_B B^2) = O(1)$ I/Os. We can also change the trajectory of a point $p$ in $O(1)$ I/Os, simply by first deleting $p$ from $\mathcal{A}$, $\mathcal{B}_x$, and $\mathcal{B}_y$, deleting the $O(1)$ event times in $\mathcal{Q}$ involving $p$, and then inserting $p$ and the $O(1)$ new event times again. If the trajectories of the points never change, there are $O(B^4)$ events.

We answer a query $q = (a, b, c)$ at time $t_q$ exactly as on the non-kinetic catalog structure, except that when considering the rectangles stored in the catalogue blocks, as well as when considering points in the loaded blocks $b_i$, we calculate the relevant $x$- and $y$-coordinates at time $t$. As previously, the query procedure takes $O(1+k)$ I/Os.

**Lemma 5.4.** *Let $S$ be a set of $B^2$ linearly moving points in $\mathbb{R}^2$. We can store $S$ in an index of size $O(B)$ blocks, so that a current three-sided query can be answered in $O(1+k)$ I/Os. The amortized cost of each event or trajectory change is $O(1)$ I/Os. If the trajectories of the points do not change, there are $O(B^4)$ events.*

Recall that an external priority search tree $\mathcal{P}$ on a set $S$ of $N$ points consists of a $x$-coordinate base B-tree with points stored in auxiliary catalog structures of the internal nodes based on $y$-coordinates. Since the definition of the structure is based only on the $x$- and $y$-coordinates of the points in $S$, it is easy to see that if $\mathcal{P}$ is a valid structure for a set of moving points at time $t$, it will remain a valid structure until the next swap event. Like the catalog structure, we can update $\mathcal{P}$ after a swap event simply by performing two deletions and two insertions in $O(\log_B n)$ I/Os (Lemma 5.2). To determine the kinetic event times, we maintain kinetic coordinate B-trees $\mathcal{B}_x$ and $\mathcal{B}_y$ on $S$, as well as an event queue B-tree $\mathcal{Q}$. Just like the catalog structure, these structures can all be maintained in $O(\log_B n)$ I/Os per event. In fact, we can maintain one global version of each of the three structures for the base priority search tree and all its auxiliary catalog structures $\mathcal{A}_v$. As previously, we can also easily change the trajectory of a point.

**Lemma 5.5.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}^2$. We can store $S$ in a kinetic external priority search tree of size $O(n)$ blocks, so that a current three-sided query can be answered in $O(\log_B n + k)$ I/Os. The amortized cost of each event or trajectory change is $O(\log_B n)$ I/Os. If the trajectories of the points do not change, there are of $O(N^2)$ events.*

Like the external priority search tree, the primary structure of the external range tree only depends on the $x$- and $y$-coordinates of the $N$ points. Thus as previously, the structure remains valid until the $x$- or $y$-coordinates of two points become equal. When a kinetic event occurs, we update the structure simply by performing two deletions and two insertions, in $O(\log_B^2 n/\log_B \log_B n)$ I/Os (Lemma 5.1). The kinetic event times can be determined in $O(\log_B n)$ I/Os using three global B-trees as previously. Thus, we obtain the main result of this section.

**Theorem 5.6.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}^2$. We can store $S$ in a kinetic external range tree 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 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 $O(N^2)$.*

**Remark.** Like kinetic B-trees, kinetic range trees work within the same asymptotic I/O bounds when the points move along more complex trajectories, provided we can quickly compute when two points lie on a common horizontal or vertical line.

### 5.3 Query/update tradeoffs

Just as in the one-dimensional case, the $O(N^2 \log_B^2 n/\log_B \log_B n))$ I/Os spent evolving the kinetic range tree through $N^2$ events is excessively high if we only need to answer a few queries. We can obtain tradeoffs between the query cost and the number of events using the technique of Section 4.2; Attaching kinetic range trees to the nodes at certain levels in the multi-level external partition tree $\mathcal{T}$.

Recall that $\mathcal{T}$ consists of a primary partition tree $T^x$ with secondary partition trees $T_v^y$ attached to certain nodes $v \in T^x$. As before, let $\Delta$ be a parameter between $B^2N$ and $N^2$, and let $r = cB$ be the fanout of a node in the primary tree $T^x$. As in Section 4.2, we discard the nodes of the primary structure $T^x$ whose parents are associated with at most $\Delta/N$ points and construct kinetic external range trees at each leaf of the truncated tree. The depth of the tree is at most $\ell = \log_{r/2}(N^2/\Delta)$. Let $v$ be a node in $T^x$ at level $l$ at which we want to store a secondary structure $T^y_v$. We construct the tradeoff structure described in Section 4.2 on $S_v$ with $\Delta_v = \Delta/r^l$ as the number of events processed by the secondary structure. Since there are at most $r^l$ nodes at level $l$, at most $\Delta$ events are processed by all the secondary structures at level $l$. Hence, the total number of events processed by all the secondary structures is $O(\Delta/\delta) = O(\Delta)$. The same analysis as in Section 4.2 can be used to show that $O(\Delta)$ events are processed by the kinetic range trees stored at the bottom of $T^x$.

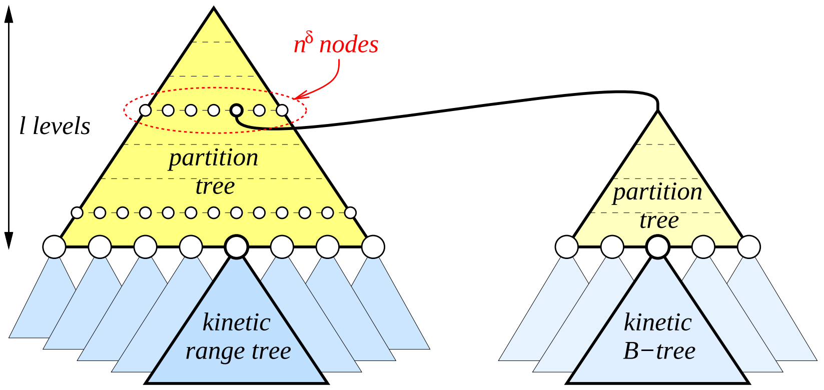

**Figure 10.** Schematic of our two-dimensional query/update tradeoff structure. Each node in certain levels of the primary tree points to a secondary structure. Only one secondary structure is shown.

Suppose the query procedure visits a node $v$ of $T^x$. If $|S_v| \le B$, we examine all the points in $S_v$ to determine which of them lie in the query rectangle $R$. If the level of $v$ is $\ell$, then we use the kinetic range tree stored at $v$ to answer the query. Otherwise, the query is processed using the algorithm of Section 3.1. Let $\tilde{\Sigma}_2(N_v)$ be the number of I/Os taken by the query procedure at a node $v$ of $T_x$. The query procedure visits at most $n^\delta$ secondary structures, each over at most $N_v/n^\delta$ points and with at most $\Delta N_v/(Nn^\delta)$ events. Using Theorem 4.2, the total number of I/Os spent is $O(n^{\delta/2+\varepsilon}N_v^{1/2+\varepsilon}/\sqrt{\Delta})$. Following the analysis in Section 3.1, we obtain the following recurrence for $\tilde{\Sigma}_2(N_v)$.

$$\tilde{\Sigma}_2(N) \le \begin{cases} O(n^{\delta/2+\varepsilon}N_v^{1/2+\varepsilon}/\sqrt{\Delta}) + \alpha\sqrt{r}\cdot\tilde{\Sigma}_2\left(\dfrac{2N_v}{r}\right) & \text{if } N_v > \Delta/N, \\ O(\log_B n) & \text{if } N_v \le \Delta/N. \end{cases}$$

Combining the analysis in Sections 3.1 and 4.2, we solve this recurrence and obtain the following.

**Theorem 5.7.** *Given a set $S$ of $N$ points in $\mathbb{R}^2$, each moving linearly with a fixed velocity, and a parameter $BN \le \Delta \le N^2$, 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. A point can be inserted or deleted in $O(\log_B^2(N^2/\Delta) + \log_B^2(\Delta/N)/\log_B\log_B(\Delta/N))$ amortized, expected I/Os, and the amortized cost of processing a kinetic event is $O(\log_B^2(\Delta/N)/\log_B\log_B(\Delta/N))$ I/Os. If the trajectories of the points do not change, the number of kinetic events is $O(\Delta)$.*

## 6 Time-Responsive Indexing

Partition trees can answer an arbitrary sequence of timed range queries, but the cost of answering each query is high. On the other hand, kinetic range trees answer queries very quickly, but only if the queries arrive 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's time stamp $t$ 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. As with our earlier results, we first explain our ideas for moving points on the real line and then extend them to the more complex two-dimensional case.

## 6.1 Recent-past and near-future queries

As observed in the previous section, the combinatorial structure of a kinetic data structure $\mathcal{K}$ is fixed until there is an *event*. For kinetic B-trees, an event occurs when two points have the same value; for kinetic range trees, an event occurs when some pair of points share a common $x$- or $y$-coordinate. At some of these events, we update the kinetic data structure, using $O(\log_B^2 n)$ I/Os. Let $t_1, t_2, \ldots$ be the sequence of event times at which $\mathcal{K}$ is updated, and let $\mathcal{K}_i$ be the version of $\mathcal{K}$ between time $t_{i-1}$ and time $t_i$. (For notational convenience, we define $t_0 = -\infty$.) If the current time is between $t_{i-1}$ and $t_i$, our idea is to maintain the versions $\mathcal{K}_{i-\mu}, \mathcal{K}_{i-\mu+1}, \ldots, \mathcal{K}_{i+\mu}$ of $\mathcal{K}$ for some parameter $\mu$, that is, we maintain $\mu$ past versions and $\mu$ future versions of $\mathcal{K}$. Note that the future versions $\mathcal{K}_i, \ldots \mathcal{K}_{i+\mu}$ 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 will also change.

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 [12, 20, 48]. There are two main differences between our structures and standard persistent data structures. First, instead of storing all the past versions, we maintain only a few past versions and we delete a past version when it becomes too old. Second, we also maintain several future versions of the data structure, which we must update when the trajectory of a point changes.

**Multiversion kinetic B-trees.** In the case of one-dimensional moving points, we can directly apply the ideas of Driscoll *et al.* [20], Becker *et al.* [12], and Varman and Verma [48] to obtain a persistent (or multiversion) B-tree. 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 in a persistent B-tree is performed almost like a normal insertion. We first find the relevant leaf $z$ and insert the elements if there is room for it. Otherwise we have an *overflow*. In this case we first copy all alive elements in $z$ and make the current time the *death time* of $z$ (i.e., $z$ becomes inactive). Depending on how many elements we copied, we either split them into two equal size groups and construct two new leaves on them, construct one new leaf on them, or we copy the alive elements from one of the siblings of $z$ and construct one or two leaves out of all the copied elements. In all cases we ensure that there is room for $\Theta(B)$ future updates in each of the new leaves. We then insert the new element into the relevant leaf and set the birth time of all new leaves to the current time. Finally, we insert pointers to the new leaves at the parent of $z$ and (persistently) delete the reference to $z$. This may result in similar overflow operations cascading up one path in the tree; these are handled in a similar manner. We refer to this procedure as the *persistent node copying* (or *pointer updating*) procedure. A deletion is performed similarly. Becker *et al.* [12] show that each update operation takes $O(\log_B n)$ I/Os, and that $\nu$ update operations require $O(\nu/B)$ additional disk blocks.

If we want to maintain $\mu = N$ past and future version of the tree, we store the death times of all nodes in a global priority queue. Let $t_i$ denote the time at which the $i$th event occurs, then at time $t_i$, we delete all nodes of the tree whose death times are in the range $[t_{i-N}, t_{i-N+1}]$. Since there are $O(\log_B n)$ such nodes, this step requires $O(\log_B n)$ I/Os. The analysis of Becker *et al.* [12] implies that the total size of the structure remains $O(n)$ blocks.

There is also a simpler alternative method. Suppose we have a multiversion B-tree for the time interval $[t_{i-N}, t_{i+N}]$. During the time interval $[t_i, t_{i+N}]$, we construct a separate multiversion B-tree for the time interval $[t_i, t_{i+2N}]$, using $O(\log_B n)$ I/Os per event. At time $t_{i+N+1}$, we discard the old multiversion B-tree, begin using the just-finished tree to answer queries, and begin constructing a new multiversion B-tree for the time interval $[t_{i+N}, t_{i+3N}]$. This approach avoids the usage of the global event queue for storing the death times and guarantees $O(\log_B n)$ processing time at each event. The disadvantage is that we have to maintain two multiversion B-trees. We will refer to this scheme of maintaining a partial multiversion B-tree as the *replication method.*

**Theorem 6.1.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$. We can store $S$ in an index 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 $t$ and the current time. The amortized cost of a kinetic event is $O(\log_B n)$ 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.*

**Multiversion external kinetic range trees.** Recall that an external range tree has three levels. The primary and secondary structures are variants of B-trees, and we can store their different versions by adapting standard techniques as above [20, 12, 48]. Before discussing them, we first describe the third level structure, namely the catalog structure $\mathcal{A}_v$ stored at each node $v$ of a priority search tree. This structure answers 3-sided range queries on a set of $O(B^2)$ points.

Recall that $\mathcal{A}_v$ maintains an extra update block $\mathcal{U}$ that records updates (kinetic events); after $B$ kinetic events have occurred, $\mathcal{A}_v$ is reconstructed and the old version of $\mathcal{A}_v$ is discarded. To maintain multiple versions of $\mathcal{A}_v$, 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 $\mathcal{U}$ in addition to the identity of the point. When $\mathcal{A}_v$ 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 $\mathcal{A}_v^1, \mathcal{A}_v^2, \dots, \mathcal{A}_v^{T/B}$ of $\mathcal{A}_v$, only the last of which is active. For each inactive version $\mathcal{A}_v^j$, we store its *death* time $d_j$, the time at which $\mathcal{A}_v^j$ became inactive. To maintain the $T$ versions of $\mathcal{A}_v$, we need $O(T)$ disk blocks.

To report all points of $\mathcal{A}_v$ that lie in a 3-sided rectangle $R$ at time $t$, we first find the version $\mathcal{A}_v^j$ that is active at time $t$, *i.e.*, such that $d_{j-1} \le t < d_j$. Since we also maintain multiple versions of the primary and secondary structures of the kinetic range tree, $\mathcal{A}_v^j$ can be found using $O(1)$ I/Os. Then we simply query $\mathcal{A}_v^j$ as in Section 5, except that we report only those points in the update block that were inserted before $t$, and we do not report points deleted before $t$. In total we use $O(1 + K_v/B)$ I/Os, where $K_v$ is the number of reported points.

Next, we describe how to maintain multiple versions of a priority search tree $\mathcal{P}$. Suppose we want to insert a point into $\mathcal{P}$. The insertion procedure follows a path $\Pi$ from the root to a leaf of $\mathcal{P}$ and possibly inserts (and/or deletes) a point into the catalog structure at each node on $\Pi$. Whenever a new copy of a catalog structure $\mathcal{A}_v$ is constructed at a node $v$, we keep the old copy of $v$ and attach the new copy at $v$, in the persistent manner described above. The insertion procedure also updates the priority search tree in the same way as the insertion procedure for a multiversion

B-tree, as sketched earlier. Since the insertion of a point in $\mathcal{P}$ causes a constant number of updates at most $O(\log_B n)$ catalog structures, it follows from the above discussion and the analysis in [12] that the total size of the structure for maintaining $\mu$ versions of $\mathcal{P}$ is $O(\mu \log_B(n))$. Each query can still be answered using $O(\log_B n + k)$ I/Os.

Finally, the primary tree is also maintained as a multiversion B-tree. Whenever a new copy of the root of one of the priority search trees or the B-tree is made at a node $v$, we copy the new root persistently at $v$, as described earlier. The primary tree itself is updated using partial rebuilding. Whenever the subtree rooted at a node $v$ is reconstructed, we keep the old copy of the subtree and attach the new copy of the subtree at the parent of $v$ in a persistent manner. The subtree rooted at $v$ is reconstructed after $\Omega(N_v)$ update operations, and the insertion or deletion of a point in $\mathcal{K}$ inserts or deletes a point in the secondary structures at all the ndoes along a path from the root to a leaf. Therefore it performs $O(\log_B n/\log_B \log_B n)$ update operations on secondary structures. Therefore maintaining $\mu = n/\log_B n$ versions of the overall structure requires

$$O\left(\frac{n}{\log_B n} \cdot \frac{\log_B n}{\log_B \log_B n} \cdot \log_B n\right) = O\left(\frac{n \log_B n}{\log_B \log_B n}\right)$$

blocks. We still spend $O(\log_B^2 n)$ amortized I/Os at each event to update $\mathcal{K}$. A query is answered in the same way as described in Section 5.1 except that at each node we use the procedure for multiversion B-trees to decide which child or which secondary structures of a node we should visit. We omit the remaining straightforward but tedious details.

Finally, we store the death times of all versions of all auxiliary structures in a global priority queue. When the $i$th kinetic event occurs, at time $t_i$, we delete all the versions of auxiliary structures whose death times lie in the interval $[t_{i-\mu}, t_{i-\mu+1})$. Again, we can also use the alternative method, which is simpler but maintains two structures at any time. Either way, we obtain the following.

**Theorem 6.2.** *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 $t$ 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.*

## 6.2 Answering distant-future queries

We now combine partition trees with multiversion kinetic data structures to obtain an index whose query cost is a monotone function of $|t - \mathit{now}|$. The combination is similar to the multilevel data structures discussed in Sections 3 and 5.3. For simplicity, we describe the indexing scheme for one-dimensional points only; a similar approach works in $\mathbb{R}^2$.

Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$, and let $P$ denote the corresponding set of static points in the dual $xt$-plane. 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 $P_v \subseteq P$; let $S_v \subseteq S$ be the corresponding subset of $S$. Let $\delta$ be an arbitrarily small constant. For each node $v$ whose depth is a multiple of $\delta \log_{cB} n$, we construct a multiversion kinetic B-tree $\mathcal{K}_v$ on $S_v$ (Theorem 6.1) that stores $N_v$ versions of the B-tree. See Figure 11. We also maintain the time interval $[t_v^-, t_v^+]$ during which $\mathcal{K}_v$ is valid. The total size of the resulting structure is $O(n/\delta) = O(n)$ blocks. Note that deeper secondary structures store fewer points, and thus maintain fewer versions; for the same reason, however, the events that define those versions are typically more spread out through time, so the valid time interval may actually be longer.

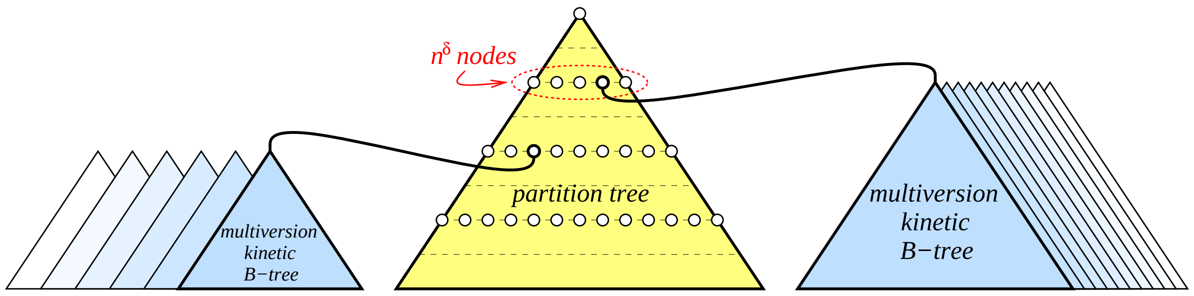

**Figure 11.** Schematic of our data structure for distant queries. Each node in certain levels of the partition tree points to a multiversion kinetic B-tree. Only two secondary structures are shown.

An event is now defined to be the moment when two points stored in some secondary structure collide. The kinetic B-trees stored at level $d$ process $O(N^2/N^{d\delta})$ events, so the total number of events processed is $O(N^2)$. At each event, the secondary data structures can be updated in $O(\log_B n)$ I/Os.

A 1-dimensional Q1 query—report all points of $S(t)$ 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 $t \in [t_v^-, t_v^+]$, then we report all points of $S_v(t) \cap I$ using $\mathcal{K}_v$. This requires $O(\log_B n + K_v/B)$ I/Os, where $K_v = |S_v(t) \cap I|$. 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 the timestamp $t$, is $O(n^{1/2+\varepsilon} + k)$; in the worst case, we do not use any of the secondary kinetic structures $\mathcal{K}_v$. However, if $t$ 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 the same secondary structure in a short period of time.

**Theorem 6.3.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$. We can store $S$ in an index of size $O(n)$ blocks so that the cost of a Q1 query at time $t$ is a monotonically increasing function of $|t - \textit{now}|$, and it is $O(n^{1/2+\varepsilon} + k)$ I/Os in the worst case.*

For random points, however, we can simplify the structure and prove a bound on the query time as a function of the number of events between $t$ and *now*. For simplicity, we describe only the modifications necessary to handle queries in the past; similar modifications also allow us to handle queries in the future in the same I/O bounds. As usual for random points, we replace the partition tree with a grid tree. Let $\delta$ be an arbitrarily small constant. For simplicity of exposition, assume that the grid tree has fanout $n^\delta$. (If $n^\delta > B$, we can represent each "node" of this grid tree with a smaller grid tree with fanout $B$ and depth $\delta \log_b n$.) We attach a multiversion kinetic B-tree $\mathcal{K}_v$ at every node $v$ of the grid tree. Queries are answered exactly as for partition trees above.

In order to get a guarantee on the query cost, we modify the indexing scheme as follows. A *global* event occurs whenever two points of $S$ collide. Let $t_i$ denote the time at which the $i$th global event occurs. An event at which two points $p$ and $q$ collide is called *local* at a node $v$ if both $p$ and $q$ belong to $S_v$. Let $d$ be the depth of a node $v$. $\mathcal{K}_v$ maintains every version of the kinetic B-tree on $S_v$ for $2Nn^{\delta d} = 2\mu$ global events, using the replication method described above. More precisely, during the interval $[t_{i\mu}, t_{(i+2)\mu}]$, we maintain all versions of the B-tree on $S_v$ corresponding to the local events that occur in that interval. The replication method will construct $\mathcal{K}_v$ for the window $[t_{(i+1)\mu}, t_{(i+3)\mu}]$ during the interval $[t_{(i+1)\mu}, t_{(i+2)\mu}]$. For any $i$, computing the time $t_i$ of the $i$th global event is equivalent to computing the $i$th leftmost intersection point in the arrangement of a set of $N$ lines. Adapting a randomized algorithm by Matoušek [32] to the external memory model, we can compute $t_i$ in $O(N \log_B n)$ I/Os. We compute the value of $t_i$ $O(N^2/Nn^{\delta d})$ times, so we compute a total of $O(N)$ $t_i$'s. Hence, we spend $O(\log_B n)$ time in computing them at each event. Recall that an event is local for $v$ if and only if both of the relevant points lie in the square $\square_v$. Since the points are distributed randomly, it can be shown that $t_i$ is local for $v$ with probability $n^{-2\delta d}$. Thus, the expected number of versions of $\mathcal{K}_v$ that we must maintain is $N/n^{\delta d}$. The expected size of $S_v$ is $N/n^{\delta d}$, so $\mathcal{K}_v$ uses $O(n^{1-\delta d})$ expected blocks. It follows that the expected total size of all the secondary structures is $O(n)$ blocks.

Now consider a query whose time stamp is $\Delta$ global events into the past, and let $\ell$ be the smallest integer such that $\Delta < Nn^{\delta \ell}$. The query algorithm uses the first $\ell$ levels of the grid tree, and then switches to the secondary kinetic B-trees, exactly as in the query/update tradeoff data structure in Section 4.2. The query algorithm recursively visits $O(n^{\delta \ell/2})$ nodes in the grid tree and performs $O(n^{\delta \ell/2})$ kinetic B-tree queries, so the total time is $O(n^{\delta \ell/2} \log_B n) = O(\sqrt{\Delta/n} \log_B n)$.

**Theorem 6.4.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}$ whose initial positions and velocities are uniformly distributed in $[0,1]$. We can store $S$ in an index of expected size $O(n)$ blocks so that a Q1 query at time $t$ can be answered in $O(\sqrt{\Delta/n} \log_B n + k)$ expected I/Os, where $0 \le \Delta \le \binom{N}{2}$ is the number of events between $t$ and now.*

In $\mathbb{R}^2$ we can prove a similar result, with slightly worse bounds on the size of the data structure and its update time.

**Theorem 6.5.** *Let $S$ be a set of $N$ linearly moving points in $\mathbb{R}^2$. We can store $S$ in 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 $t$ is a monotonically increasing function of $|t - now|$. If the initial positions and velocities of the points are uniformly distributed in $[0,1]^2$, 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 $t$ and now.*

## 7 Approximate 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 polyhedral 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 $d_P(a,b) = \inf\{\lambda \mid b \in \lambda P + a\}$. It is well known 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 12(i). Let $v_1, \ldots, v_m$ be the sequences of vertices in counterclockwise order, with $v_1$ lying on the $x$-axis. Let $C_i$ be the cone formed by the rays $ov_i$ and $ov_{i+1}$; here $v_{m+1} = v_1$. 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 12(ii).

Decompose $P$ into $m$ triangles $\triangle_1, \ldots, \triangle_m$, by connecting every vertex of $P$ to the origin. Note that $d_{\triangle_i}(a,b)$ is finite if and only if $b \in C_i + a$. It is easily seen that

$$d_P(a,b) = \min_{1 \le i \le k} d_{\triangle_i}(a,b).$$

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 $\triangle_1$.

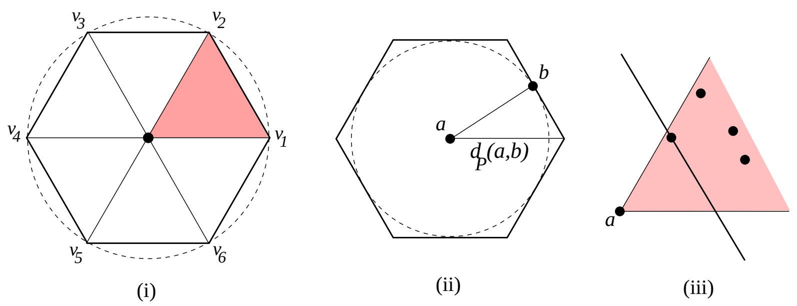

**Figure 12.** (i) A regular $m$-gon $P$ centered at the origin. (ii) The difference between the Euclidean and $d_P$ metrics. (iii) The wedge $Q_a$ and the nearest neighbor to $a$ under the distance function $d_{\triangle_1}$.

For a point $a$, let $Q(a)$ be the cones formed by the rays emanating from $a$ in directions $ov_1$ and $ov_2$; see Figure 12 (iii). The nearest neighbor of a point $\sigma$ under the distance function $d_{\triangle_1}$ is the point in $Q(\sigma) \cap S$ nearest to $\sigma$ in the direction normal to $v_1v_2$; see Figure 12(iii). For any point $p \in S$, let $f(p)$ be the dot product of $p$ with the vector normal to $v_1v_2$. The nearest $d_{\triangle_1}$-neighbor in $S \cap Q(a)$ 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 $\sigma$ and a time $t$, compute the point in $p(t) \in Q(\sigma)$ minimizing $f(p(t))$.

For simplicity, we assume that $ov_2$ is the $y$-axis. We map $S$ to a set of points $P^x$ in the dual $xt$-plane and to another set $P^y$ in the dual $yt$-plane. We construct a two-level partition tree $\mathcal{T}$ described in Section 3. Let $T^y$ be a second level partition tree, and let $P_v^y \subseteq P^y$ be the subset of points associated with a node $v$ of $T^y$. Let $S_v$ be the corresponding subset of points in $S$; set $N_v = |S_v|$. If the depth of $v$ in $T^y$ is an integer multiple of constant $\alpha > 0$. Define $F_v(t) = \min_{p \in S_v} f(p(t))$ to be the *lower envelope* of the functions $f(p(t))$. If $p$ is moving linearly, then $f(p(t))$ is a linear function, so 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 [26]. 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 [3, 38]. The three-level data structure requires $O(N_v/B)$ blocks.

Given a query point $\sigma$ and time value $t$, we answer a Q3$'$ query as follows. We first search the primary and secondary partition trees to compute the set of points that lie inside $Q(\sigma)$. Let $v$ be a node of the secondary tree visited by the query procedure so that $\triangle_v$ lies inside $Q(\sigma)$ and a tertiary data structure is stored at $v$. Instead of reporting all the points in $S_v$, we use the tertiary structure stored at $v$ and compute $F_v(t)$ using $O(\log_B n)$ I/Os. Let $p_v$ be the point such that $f(p_v(t)) = F_v(t)$. Otherwise, we the query procedure proceeds as earlier. Among all points $p_v$ returned by the algorithm, we choose the one that is nearest to $\sigma$ at time $t$. The total number of I/Os taken by this procedure is $O(n^{1/2+\varepsilon})$. Hence, we obtain the following.

**Lemma 7.1.** *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 7.2.** *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.*

**Remark.** 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, using a scheme of Dudley [21], we can $\delta$-approximate the sphere by a polytope with $O(1/\delta^{(d-1)/2})$ facets; see [11, 16] for similar results. Using a separate data structure for each facet of this polytope, similar to the Q3$'$ structure above, we can compute $\delta$-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. We omit further details.

## 8 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. Most of the indexing schemes presented in the paper are too complex to be of practical use. Can one develop simpler indexing scheme with similar provable bounds? Some progress in this direction has been made in [40], but the problem remains largely open.

2. In many applications the trajectories of points is updated frequently. Can one update the index more efficiently than simply deleting and re-inserting a point? Can one obtain a tradeoff between the time spent in answering queries and in updating the trajectories?

3. Develop an efficient indexing scheme for answering exact nearest-neighbor queries for moving points. Even the best known internal-memory data structures are either prohibitively large or have very slow query times [19].

4. Can the indexing scheme described in Section 6.2 be extended so that the bounds in Theorem 6.5 hold in the worst case, not just for random points and trajectories? We believe the more recent techniques of the authors [4] may be helpful in answering this question.

## References

[1] Arcview gis, arcview tracking analyst, 1998.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. 19th Annu. ACM Sympos. Principles Database Syst.*, pages 175–186, 2000.

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

[4] P. K. Agarwal, L. Arge, and J. Vahrenhold. A time responsive indexing scheme for moving points. In *Proc. 7th 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.Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. In *Proc. ACM Symp. on Computational Geometry*, pages 107–116, 1998.

[7] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets.* Kluwer Academic Publishers, 2002. (To appear).

[8] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. 18th ACM Symp. Principles of Database Systems*, pages 346–357, 1999.

[9] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. 37th IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.

[10] 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.

[11] J. Basch, L. J. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 344–351, 1997.

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

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

[14] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

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

[16] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.

[17] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

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

[19] O. Devillers, M. Golin, K. Kedem, and S. Schirra. Queries on Voronoi diagrams of moving points. *Comput. Geom. Theory Appl.*, 6:315–327, 1996.

[20] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[21] R. M. Dudley. Metric entropy of some classes of sets with differentiable boundaries. *J. Approx. Theory*, 10:227–236, 1974.

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

[23] M. Erwig and M. Schneider. Developments in spatio-temporal query languages. DEXA Workshop, 1999.

[24] L. Forlizzi, R. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proc. SIGMOD Intl. Conf. Management of Data*, pages 319–330, 2000.

[25] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

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

[27] 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(3):589–612, 1996.

[28] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Proc. Intl. Workshop on Spatio-Temporal Database Management, LNCS 1678*, pages 119–134, 1999.

[29] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM Symp. Principles of Database Systems*, pages 261–272, 1999.

[30] M. Koubarakis. The complexity of query evaluation in indefinite temporal constraint databases. *Theoretical Computer Science*, 171:25–60, 1997.

[31] M. Koubarakis and S. Skiadopoulos. Tractable query answering in indefinite constraint databases: Basic results and applications to querying spatiotemporal information. In *Proc. Intl. Workshop on Spatio-Temporal Database Management*, pages 204–223, 1999.

[32] J. Matoušek. Randomized optimal algorithm for slope selection. *Inform. Process. Lett.*, 39:183–187, 1991.

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

[34] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[35] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[36] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, pages 153–197. Springer-Verlag, LNCS 1340, 1997.

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

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

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

[40] 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.

[41] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 25–35, 1994.

[42] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 199.

[43] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison Wesley, MA, 1990.

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

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

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

[47] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

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

[49] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. López. Indexing the positions of continuously moving objects. In *Proc. SIGMOD Intl. Conf. Management of Data*, pages 331–342, 2000.

[50] O. Wolfson, S. Chamberlain, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proc. IEEE Intl. Conference on Data Engineering*, pages 588–596, 1998.

[51] O. Wolfson, L. Jiang, A. P. Sistla, S. Chamberlain, and M. Deng. Databases for tracking mobile units in real time. In *Proc. Intl. Conference on Database Theory*, pages 169–186, 1999.

[52] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–287, 1999.

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

[54] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Intl. Conf. on Scientific and Statistical Database Management*, pages 111–122, 1998.