Time Responsive External Data Structures for Moving Points

Pankaj K. Agarwal[1*], Lars Arge[1**], and Jan Vahrenhold[2***]

[1] Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708, USA. EMail: {pankaj,large}@cs.duke.edu.
[2] Westfälische Wilhelms-Universität Münster, Institut für Informatik, 48149 Münster, Germany. EMail: jan@math.uni-muenster.de.

Abstract. We develop external data structures for storing points in one or two dimensions, each moving along a linear trajectory, so that a range query at a given time t_q can be answered efficiently. The novel feature of our data structures is that the number of I/Os required to answer a query depends not only on the size of the data set and on the number of points in the answer but also on the difference between t_q and the current time; queries close to the current time are answered fast, while queries that are far away in the future or in the past may take more time.

1 Introduction

I/O-communication, and not internal memory computation time, is often the bottleneck in a computation when working with datasets larger than the available main memory. Recently, external geometric data structures have received considerable attention because massive geometric datasets arise naturally in many applications (see [19, 5] and references therein). The need for storing and processing continuously moving data arises in a wide range of applications, including air-traffic control, digital battlefields, and mobile communication systems. Most of the existing database systems, which assume that the data is constant unless it is explicitly modified, are not suitable for representing, storing, and querying continuously moving objects because either the database has to be continuously updated or a query output will be obsolete. A better approach would be to represent the position of a moving object as a function $f(t)$ of time, so that the position changes without any explicit change in the database system and so that the database needs to be updated only when the function $f(t)$ changes (e.g., when the velocity of the object changes).

In this paper, we focus on developing efficient external data structures for storing a set of moving points in one- or two-dimensional space so that range

* Supported in part by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants ITR–333–1050, EIA–9870734, EIA–997287, and CCR–9732787, and by grant from the U.S.-Israeli Binational Science Foundation.

** Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA-9972879 and CAREER grant EIA-9984099.

*** Part of this work was done while visiting Duke University.

queries over their (future or past) locations can be answered quickly. Our focus is on what we call *time responsive* data structures that have fast response time for near-future or near-past queries but may take more time for queries that are far away in time. Time responsiveness is important in, e.g., air-traffic control, where queries in the near future are more critical than queries far away in the future.

1.1 Problem statement

Let $S = \{p_1, p_2, \ldots, p_N\}$ be a set of moving points in $\mathbb{R}^d$, $d = 1, 2$. For any time t, let $p_i(t)$ denote the position of p_i at time t, and let $S(t) = \{p_1(t), \ldots, p_N(t)\}$. We will assume that each point p_i is moving along a straight line at some fixed speed, that is, $p_i(t) = \mathbf{a}_i \cdot t + \mathbf{b}_i$ for some $\mathbf{a}_i, \mathbf{b}_i \in \mathbb{R}^d$. We will use t_{now} to denote the current time. We are interested in answering queries of the following form:

Q1. Given a set S of points moving along the y-axis, a y-range $R = [y_1, y_2]$, and a time t_q, report all points of S that lie inside R at time t_q, that is, $S(t_q) \cap R$.
Q2. Given a set S of points moving in the xy-plane, an axis-aligned rectangle R, and a time t_q, report all points of S that lie inside R at time t_q, that is, $S(t_q) \cap R$.

As our main interest is minimizing the number of disk accesses needed to answer query, we will consider the problem in the standard external memory model; see e.g. [5]. This model assumes that each disk access transmits a contiguous block of B units of data in a single *input/output operation* (or *I/O*). The efficiency of a data structure is measured in terms of the amount of disk space it uses (in units of disk blocks) and the number of I/Os required to answer a range query. As we are interested in solutions that are *output sensitive*, our query I/O bounds are not only expressed in terms of N, the number of points in S, but also in terms of K, the number of points reported by the query. The minimum number of disk blocks we need to store N points is $\lceil N/B \rceil$, and at least $\lceil K/B \rceil$ I/Os are needed to report K output points. We refer to these bounds as “linear.”

1.2 Previous results

Recently, there has been a flurry of activity in computational geometry and databases on problems dealing with moving objects. In the computational geometry community, Basch *et al.* [7] introduced the notion of *kinetic data structures*. Their work led to several interesting internal memory results related to moving points; see [12] and references therein. The main idea in the kinetic framework is that even though the points move continuously, the relevant combinatorial structure of the data structure change only at certain discrete times. Therefore the data structure does not need to be updated continuously. Instead *kinetic updates* are performed on the data structure only when certain *kinetic events* occur. These events have a natural interpretation in terms of the underlying structure. In contrast, in fixed-time-step methods, where the structure is updated at fixed time intervals, the fastest moving object determines an update time step for the entire data structure. Even though the kinetic framework often leads to very query efficient structures, one disadvantage of kinetic data structures is that queries can only be answered at the current time (i.e., in chronological order). Thus while kinetic data structures are very useful in simulation applications they are less suited for answering the types of queries we consider in this paper.

In the database community, a number of practical methods have been proposed for handling moving objects (see [20,16] and the references therein). Almost all of them require $\Omega(N/B)$ I/Os in the worst case to answer a Q1 or Q2 query—even if the query output size is $O(1)$. Kollios *et al.* [13] proposed the first provably efficient data structure, based on partition trees [2,3,15], for queries of type Q1. The structure uses $O(N/B)$ disk blocks and answers queries in $O((N/B)^{1/2+\epsilon} + K/B)$ I/Os, for any $\epsilon > 0$. Agarwal *et al.* [1] extended the result to Q2 queries. Kollios *et al.* [13] also present a scheme that answers a Q1 query using optimal $O(\log_B N + K/B)$ I/Os but using $O(N^2/B)$ disk blocks. These data structures are *time-oblivious*, that is, they do not evolve over time. Agarwal *et al.* [1] were the first to consider kinetic data structure in external memory. Based on external range trees [6], they developed a data structure that answers a Q2 query in optimal $O(\log_B N + K/B)$ I/Os using $O((N/B) \log_B N/(\log_B \log_B N))$ disk blocks—provided, as discussed above, that the queries arrive in chronological order. The amortized cost of a kinetic event is $O(\log_B^2 N)$ I/Os, and the total number of events is $O(N^2)$. They also showed how to modify the structure in order to obtain a tradeoff between the query bound and the number of kinetic events.

Agarwal *et al.* [1] were also the first to propose time responsive data structures in the context of moving points. They developed an $O(N/B)$ space structure for Q1 queries and a $O((N/B) \log_B N/(\log_B \log_B N))$ space structure for Q2 queries, where the cost of a query at time t_q is a monotonically increasing function of the difference between t_{now} and t_q. The query bound never exceeds $O((N/B)^{1/2+\epsilon} + K/B)$. They were only able to prove more specific bounds when the positions and velocities of the points are uniformly distributed inside a unit square.

1.3 Our results

In this paper we combine the ideas utilized in time-oblivious and kinetic data structures in order to develop the first time responsive external data structures for Q1 and Q2 queries with provably efficient specific query bounds depending on the difference between t_{now} and t_q. Our structures evolve over time, but unlike previous kinetic structures, queries can be answered at any time in the future. Our data structures are of a combinatorial nature and we therefore measure time in terms of kinetic events. We define $\varphi(t)$ to be the number of kinetic events that occur (or have occurred) between t_{now} and t, and our query bounds will depend on $\varphi(t_q)$, the number of kinetic events between the current time and t_q. For brevity, we will focus on queries in the *future*, i.e., $t_q \geq t_{now}$. The somewhat simpler case of queries in the *past* ($t_q \leq t_{now}$) can be handled similarly.

In Section 3, we describe a time responsive data structure for Q1 queries. A kinetic occurs when two points pass through each other (become equal) and their relative orderings change. Our data structure uses $O((N/B)\log_B N)$ disk blocks and answers a Q1 query with time stamp t_q such that $\varphi(t_q) \leq NB^{i-1}$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os. The expected amortized cost of a kinetic event is $O(\log_B^2 N)$ I/Os. Note that a query at a time t_q with $\varphi(t_q) \leq NB$ can be answered in optimal $O(\log_B N + K/B)$ I/Os. Previously such query efficient structures either used $O(N^2/B)$ space [13] or required the queries to arrive in chronological order [1]. Our data structure is considerably simpler than the one proposed in [1] and does not make any assumptions on the distribution of the trajectories in order to prove a bound on the query time.

In Section 4, we describe a time responsive data structure for Q2 queries. A kinetic event now occurs when the x- or y-coordinates of two points become equal. This data structure uses $O((N/B)\log_B N)$ disk blocks and answers a Q2 query with time stamp t_q such that $\varphi(t_q) \leq NB^i$ in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os. Each kinetic event is handled in $O(\log_B^3 N)$ expected I/Os. If $\varphi(t_q) \leq NB$ the query is answered in $O(\sqrt{N/B}\log_B N + K/B)$ I/Os, an improvement over previous $O((N/B)^{1/2+\epsilon} + K/B)$ I/O structures for non-chronological queries.

2 Preliminaries

Arrangements. Given a set S of N lines in $\mathbb{R}^2$, the *arrangement* $\mathcal{A}(S)$ 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 S. $\mathcal{A}(S)$ has $\Theta(N^2)$ vertices, edges, and faces. For each $1 \leq k \leq N$, the *k-level* $\mathcal{A}_k(S)$ of $\mathcal{A}(S)$ is defined as the closure of all edges in $\mathcal{A}(S)$ that have exactly k lines of S (strictly) below them. The k-level is a polygonal chain that is monotone with respect to the horizontal axis. Dey [10] showed that the maximum number of vertices on the k-level in an arrangement of N lines in the plane is $O(Nk^{1/3})$. Recently, Tóth proved a lower bound of $\Omega(N2^{\sqrt{\log k}})$ on the complexity of a k-level [17]. Using a result by Edelsbrunner and Welzl [11], Agarwal *et al.* [2] discussed how a given level of an arrangement of lines can be computed I/O-efficiently.

Lemma 1 (Agarwal *et al.* [2]). *A given level with T vertices in an arrangement of N lines can be computed in $O(N\log_2 N + T\log_2 N\log_B N)$ I/Os.*

B-trees. A *B-tree*, one of the most fundamental external data structures, stores N elements from an ordered domain using $O(N/B)$ disk blocks so that a one-dimensional range query can be answered in $O(\log_B N + K/B)$ I/Os [9]. An element can be inserted/deleted in $O(\log_B N)$ I/Os. A standard B-tree answers queries only on set of elements currently in the structure. A *persistent* (or multi-version) B-tree, on the other hand, supports range queries in all states (*versions*) of the data structure in $O(\log_B N + K/B)$ I/Os [8, 18]. Updates can be performed in $O(\log_B N)$ I/Os on the current structure, and we refer to the structure existing after T updates as the structure existing at *time* T.

Lemma 2 ([8, 18]). *A persistent B-tree constructed by performing N updates using $O(\log_B N)$ I/Os each, uses $O(N/B)$ disk blocks and supports range queries at any time in $O(\log_B N + K/B)$ I/Os.*

3 Data Structure for Moving Points in $\mathbb{R}$

In this section we consider queries of type Q1. If we interpret time as the t-axis in the parametric ty-plane, each point in S traces out a line in this ty-plane. Abusing the notation a little, we will use S to denote the resulting set of N lines. A Q1 query then corresponds to reporting all lines of S that intersect a vertical segment (Figure 1 (i).)

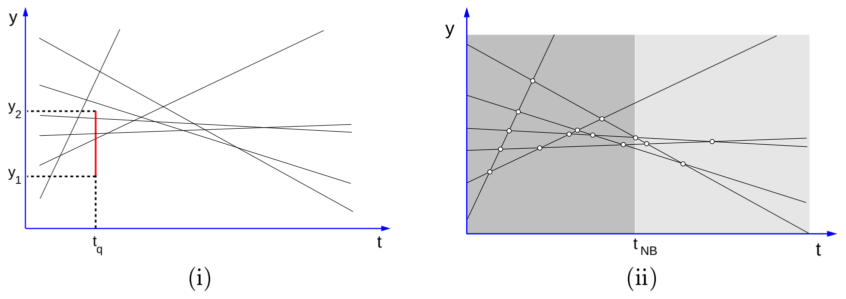

Fig. 1. (i) Lines in ty-plane traced by S. A Q1 query corresponds to finding the lines intersected by segment $[(t_q, y_1), (t_q, y_2)]$. (ii) Two windows for $N = 6$ and $B = 2$.

Since a vertical line $\ell : t = \alpha$ induces a total order on the lines in S—namely the relative ordering of the points in $S(\alpha)$—and since this ordering does not change until two points pass through each other, we can design an efficient data structure using a persistent B-tree as follows: We sweep $\mathcal{A}(S)$ from left to right ($-\infty$ to ∞) with a vertical line, inserting a segment from $\mathcal{A}(S)$ in a persistent B-tree when its left endpoint is encountered and deleting it again when its right endpoint is encountered. We can then answer a Q1 query with range $R = [y_1, y_2]$ by performing a range query with R at time t_q. Since the arrangement is of size $O(N^2)$, Lemma 2 implies that this data structure answers queries in the optimal $O(\log_B N + K/B)$ I/Os using $O(N^2/B)$ space.[1] In order to improve the size to $O((N/B) \log_B N)$, while at the same time obtaining a time responsive data structure, we divide the ty-plane into $O(\log_B N)$ vertical slabs (or *windows*) and store a modified linear-space version of the above data structure in each window.

[1] Strictly speaking, Lemma 2 as described in [8] does not hold for line segments since not all segments are above/below-comparable. However, in [5] it is discussed how to extend the result to line segments.

3.1 Overall structure

Let $t_1, \ldots, t_v$, $v \leq \binom{N}{2}$, be the sorted sequence of the t-coordinates of the vertices in the arrangement $\mathcal{A}(S)$. For $1 \leq i \leq \lceil \log_B N \rceil$, set $\tau_i = t_{NB^i}$, i.e., NB^i events occur before τ_i. We define the first window $\mathcal{W}_1$ to be the vertical slab $[-\infty, \tau_1] \times \mathbb{R}$, and the ith window $\mathcal{W}_i$, $2 \leq i \leq \lceil \log_B N \rceil$, to be the vertical slab $[\tau_{i-1}, \tau_i] \times \mathbb{R}$. Figure 1 (ii) shows an example of an arrangement of six lines with two windows.

Our data structure consists of a B-tree $\mathcal{T}$ on $\tau_1, \tau_2, \ldots$, as well as a *window structure* $\mathcal{WI}_i$ for each window $\mathcal{W}_i$. In Section 3.2 below we first describe the algorithm for constructing the windows, and in Section 3.3 we then describe the data structure $\mathcal{WI}_i$ that uses $O(N/B)$ disk blocks and answers a query with $t_q \in [\tau_{i-1}, \tau_i]$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os. To answer a Q1 query we first use $\mathcal{T}$ to determine in $O(\log_B N)$ I/Os the window $\mathcal{W}_i$ containing t_q, and then we search $\mathcal{WI}_i$ with R to report all K points of $R \cap S(t)$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os.

3.2 Computing windows

We describe an algorithm that computes the $O(\log_B N)$ t-coordinates $\tau_1, \tau_2, \ldots$ using $O((N/B) \log_2 N \log_B^2 N)$ I/Os. We cannot afford to compute all vertices of the arrangement $\mathcal{A}(S)$ and then choose the desired t-coordinates. Instead we present an algorithm that, for any $k \in \mathbb{N}$, computes the kth leftmost vertex of $\mathcal{A}(S)$ I/O-efficiently. To obtain the τ_i's we run this algorithm with $k = NB^i$ for $1 \leq i \leq \lceil \log_B N \rceil$.

To find the kth left most vertex we first choose N random vertices of $\mathcal{A}(S)$ and sort them by their t-coordinates. As shown in [14], the merge-sort algorithm can be modified to compute the N vertices without explicitly computing all vertices of $\mathcal{A}(S)$. By using external merge-sort [4] we use $O((N/B) \log_B N)$ I/Os to compute the N vertices, and we can sort them in another $O((N/B) \log_B N)$ I/Os.[2] Let $p_1, p_2, \ldots, p_N$ be the sequence of vertices in the sorted order, and let W_i be the vertical slab defined by vertical lines through p_{i-1} and p_i. A standard probabilistic argument, omitted from this abstract, shows the following.

Lemma 3. *With probability at least $1 - 1/N$, each slab W_i contains $O(N)$ vertices of $\mathcal{A}(S)$.*

Suppose we have a procedure $\textsc{Count}(W_i)$ that counts using $O((N/B) \log_B N)$ I/Os the number of vertices of $\mathcal{A}(S)$ lying inside a vertical slab W_i. By performing a binary search and using $\textsc{Count}$ at each step of the search, we can compute in $O((N/B) \log_2 N \log_B N)$ I/Os the index i such that the kth leftmost vertex of $\mathcal{A}(S)$ lies in the vertical strip W_i. Using $\textsc{Count}(W_i)$ again we can check whether W_i contains more than cN vertices of $\mathcal{A}(S)$, where c is the constant hidden in the big-Oh notation in Lemma 3. If it does, we restart the algorithm.

[2] To obtain this bound we assume that the internal memory is capable of holding B blocks. The more general bound obtained when the internal memory is of size M will be given in the full paper.

By Lemma 3, the probability of restarting the algorithm is at most $1/N$. If W_i contains at most cN vertices, we find all σ_i vertices of $\mathcal{A}(S)$ lying inside the strip W_i in $O((N/B)\log_B N)$ I/Os and choose the desired vertex, using a simple modification of merge-sort; details will appear in the full paper.

What remains is to describe the COUNT(W_i) procedure. Note that two lines ℓ and ℓ' intersect inside W_i if they intersect its left and right boundaries in different order, i.e., ℓ lies above ℓ' at the left boundary of W_i but below ℓ at the right boundary, or vice-versa. The problem of counting the number of vertices of $\mathcal{A}(S)$ inside W_i reduces to counting the number of pairs of lines in S that have different relative orderings at the two boundaries of W_i. It is well known that the number of such pairs can be counted using a modified version of merge-sort. Hence, we can count the number of desired vertices using $O((N/B)\log_B N)$ I/Os by modifying external merge-sort [4]. Putting everything together, we can compute the τ_i's in a total of $O((N/B)\log_2 N \log_B^2 N)$ I/Os.

3.3 Window data structure

Let $\mathcal{W}_i = [\tau_{i-1}, \tau_i] \times \mathbb{R}$ be a window containing NB^i vertices of $\mathcal{A}(S)$. We describe how to preprocess $\mathcal{A}(S)$ into a data structure $\mathcal{WI}_i$ that uses $O(N/B)$ disk blocks and answers a Q1 query in $O(B^{i-1} + \log_B N + K/B)$ I/Os. Since $\mathcal{W}_i$ contains NB^i vertices of $\mathcal{A}(S)$, a persistent B-tree constructed on W_i would use $\Theta(NB^{i-1})$ disk blocks. We therefore instead only build the persistent structure on N/B^i carefully selected levels of $\mathcal{A}(S)$ and build separate structures for each of the N/B^i *bundles* defined by these levels. Our algorithm relies on the following simple lemma about randomly chosen levels of the arrangement $\mathcal{A}(S)$.

Lemma 4. *For a given integer $\xi \in [0, B^i - 1]$, let $\Lambda_\xi = \{\mathcal{A}_{jB^i+\xi}(S) \mid 1 \le j \le N/B^i\}$ be N/B^i levels of $\mathcal{A}(S)$. If the integer ξ is chosen randomly, then the expected number of vertices in all the levels of Λ_ξ whose t-coordinates lie inside $\mathcal{W}_i$ is $O(N)$.*

The preprocessing algorithm works as follows. We choose a random integer $\xi \in [0, B^i - 1]$. For $1 \le j < N/B^i$, let λ_j be the $(jB^i + \xi)$-level of $\mathcal{A}(S)$. Set $\Lambda_\xi = \{\lambda_j \mid 1 \le j < N/B^i\}$. We refer to Λ_ξ as the set of *critical levels*. We compute Λ_ξ using Lemma 1. If during the construction, the size of Λ_ξ becomes more than $2cN$, where c is the constant hidden in the big-Oh notation in Lemma 4, then we abort the construction, choose another random value of ξ, and repeat the above step. This way we make sure that the critical levels are of size $O(N)$. By Lemma 4, the algorithm is aborted $O(1)$ expected times, so the expected number of I/Os needed to construct the critical levels is $O(N \log_2 N \log_B N)$ (Lemma 1). We store Λ_ξ in a persistent B-tree $\mathcal{T}_i$ by sweeping the ty-plane with a vertical line from $t = \tau_{i-1}$ to τ_i so that for a vertical segment $R = [y_1, y_2]$ and a time instance $t_q \in [\tau_{i-1}, \tau_i]$, we can compute the critical levels intersected by R. Since there are $O(N)$ vertices in Λ_ξ, $\mathcal{T}_i$ uses $O(N/B)$ space and can be constructed in $O((N/B)\log_B N)$ I/Os.

For $1 \le j \le N/B^i$, let *bundle* $\mathcal{B}_j$ be the union of the levels that lie between λ_{j-1} and λ_j, including λ_{j-1}. See Figure 2 (i). For each bundle $\mathcal{B}_j$, we store the set of lines in $\mathcal{B}_j$ at any time $t \in [\tau_{i-1}, \tau_i]$ in a separate persistent B-tree $\mathcal{D}_j^i$. More precisely, we assume that every line in S has a unique identifier and keep the lines ordered by this identifier in $\mathcal{D}_j^i$. Let $\mathcal{D}_j^i(t)$ denote the version of $\mathcal{D}_j^i$ at time t. We sweep the window $\mathcal{W}_i$ from left to right. Initially $\mathcal{D}_j^i$ stores the lines of S in $\mathcal{B}_j$ at time τ_{i-1}. A line leaves or enters $\mathcal{B}_j$ at a vertex of λ_{j-1} or λ_j. Therefore we update $\mathcal{D}_j^i$ and $\mathcal{D}_{j+1}^i$ at each vertex of λ_j. Since Λ_ξ is of size $O(N)$, the total number of I/Os spent in constructing all the bundle structures is $O((N/B) \log_B N)$ and the total space used is $O(N/B)$ blocks. Finally, for every vertex $v = (t_v, y_v)$ on the critical level λ_j, we store a pointer to the roots of structures $\mathcal{D}_j^i(t_v)$ and $\mathcal{D}_{j+1}^i(t_v)$.

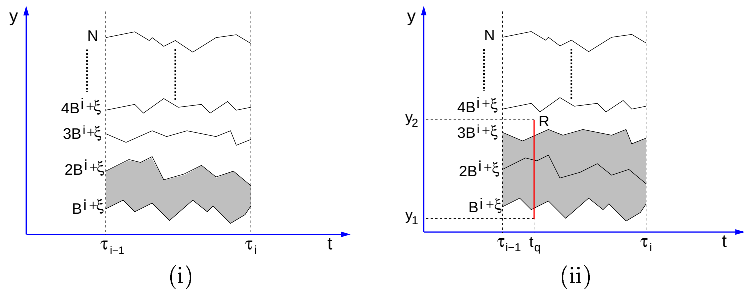

Fig. 2. (i) The N/B^i critical levels in $\mathcal{W}_i = [\tau_{i-1}, \tau_i] \times \mathbb{R}$. Bundle $\mathcal{B}_2$ is shaded. (ii) Query with $R = [y_1, y_2]$ at time t_q. All points in the shaded bundles at time t_q and all relevant points in the bundles containing the endpoints of R are reported.

To answer a Q1 query, we first use $\mathcal{T}_i$ to find the critical levels intersecting R at time t_q in $O(\log_B N)$ I/Os. Next we use the pointers from the vertices on the critical levels to find the bundle structures $\mathcal{D}_j^i$ of bundles completely spanned by R at time t_q. We report all points in these structures using $O(K/B)$ I/Os. Finally, we scan all lines in the (at most) two bundles intersected but not completely spanned by R using $O(B^i/B) = O(B^{i-1})$ I/Os and report the relevant points. In conclusion, we answer the query in $O(B^{i-1} + \log_B N + K/B)$ I/Os; see Figure 2 (ii).

Lemma 5. *Let S be a set of N points moving along the y-axis with fixed velocities, and let $\mathcal{W}_i = [\tau_{i-1}, \tau_i] \times \mathbb{R}$ be a window such that $\mathcal{A}(S)$ has $O(NB^i)$ vertices inside $\mathcal{W}_i$. We can preprocess S into a data structure $\mathcal{WI}_i$ in $O(N \log_2 N \log_B N)$ expected I/Os such that a Q1 query with $t_q \in [\tau_{i-1}, \tau_i]$ can be answered in $O(B^{i-1} + \log_B N + K/B)$ I/Os. The number of disk blocks used by the data structure is $O(N/B)$.*

This completes the description of the basic data structure, but there is one technical difficulty that one has to overcome. Immediately after building our data structure, a query in the first window W_1 can be answered in the optimal $O(\log_B N + K/B)$ I/Os. However, if we do not modify the data structure, the query performance of the structure deteriorates as the time elapses.

For example, for $t_{now} > \tau_1$, a query within the first NB events after t_{now} requires $\Omega(B + \log_B N + K/B)$ I/Os. To circumvent this problem, we rebuild the entire structure during the interval $[t_{NB/2}, t_{3NB/4}]$ as though t_{now} were $t_{3NB/4}$, and switch to this structure at time $t_{3NB/4}$. This allows us to always answer a query at time t_q with $\varphi(t_q) \leq NB^i/4$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os. Since we use $O(N \log_2 N \log_B^2 N)$ I/Os to rebuild the structure, we charge $O((\log_2 N \log_B^2 N)/B) = O(\log_B^3 N)$ I/Os to each of the $NB/4$ events to pay for the reconstruction cost. Putting everything together, we obtain the following.

Theorem 1. *Let S be a set of N points moving along the y-axis with fixed velocities. S can be maintained in a data structure using $O(\log_B^3 N)$ expected I/Os per kinetic event such that a Q1 query at time t_q, with $NB^{i-1} \leq \varphi(t_q) \leq NB^i$, can be answered in $O(B^{i-1} + \log_B N + K/B)$ I/Os. The structure can be built in $O(N \log_2 N \log_B^2 N)$ expected I/Os and uses $O((N/B)\log_B N)$ disk blocks.*

Remarks.

(i) Our window structure $\mathcal{WI}_i$ (Lemma 5) can easily be modified to work even if it is built on a set of line segments instead of lines, provided the lines obtained by extending the line segments have $O(NB^i)$ intersections in $\mathcal{W}_i$. We simply construct the critical levels λ_i on the lines but the bundle structures $\mathcal{B}_j$ on the segments.

(ii) Our result can be extended to the case in which the trajectory of each point is a piecewise-linear function of time, i.e., in the ty-plane, S is a set of N polygonal chains with a total of T vertices. Then the query time remains the same and the total size of the data structure is $O(T/B)$ disk blocks.

4 Data Structure for Moving Points in $\mathbb{R}^2$

We now turn to points moving in $\mathbb{R}^2$. Analogously to the $1D$-case, $S(t)$ traces out N lines in xyt-space, and our structure for Q2 queries utilizes the same general ideas as our structure for Q1 queries. While a kinetic event in the $1D$-case corresponds to two points passing each other, an event now occurs when the x- or y-coordinates of two points coincide. We divide the xyt-space into $O(\log_B N)$ horizontal *slices* along the t-axis such that slice $\mathcal{S}_i$ contains $O(NB^i)$ kinetic events; see Figure 3 (i). That is, we choose a sequence $\tau_1 < \tau_2 < \dots$ of $\log_B N$ time instances so that $O(NB^i)$ events occur in the interval $[\tau_{i-1}, \tau_i]$. We set $\mathcal{S}_i = \mathbb{R}^2 \times [\tau_{i-1}, \tau_i]$. As previously, our data structure consists of a B-tree on $\tau_1, \tau_2, \dots$, and a linear-space *slice structure* $\mathcal{SL}_i$ for each slice $\mathcal{S}_i$. Below, we will design a linear space data structure $\mathcal{SL}_i$, which can be used to answer a Q2 query with $t_q \in [\tau_{i-1}, \tau_i]$ in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os. Each $\mathcal{SL}_i$ can be constructed in $O(N \log_2 N \log_B N)$ expected I/Os and updated in $O(\log_B^2 N)$ I/Os.

To answer a Q2 query at time t_q with a rectangle R, we first determine the slice $\mathcal{S}_i$ containing t_q and then we query $\mathcal{SL}_i$ to report all K points of $S(t) \cap R$ in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os. Our global data structure

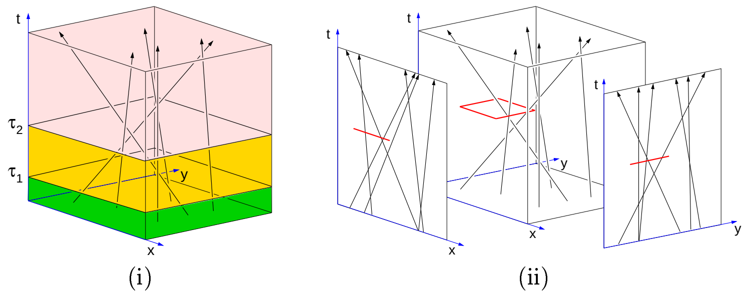

Fig. 3. (i) Three slices (ii) The lines in xyt-space traced by $S(t)$ and their projections.

uses $O((N/B)\log_B N)$ disk blocks in total. The structure can be constructed in $O(N \log_2 N \log_B^2 N)$ expected I/Os since we can compute the τ_i's as follows. Let $\mathcal{A}^x(S)$ be the arrangement obtained by projecting the lines of S onto the tx-plane (Figure 3 (ii)). Let $k_i = NB^i$ for $1 \le i \le \log_B N$. As in Section 3, we can compute, using $O(N \log_2 N \log_B^2 N)$ I/Os, the time instances τ_i^x, $1 \le i \le \log_B N$, so that the t-coordinate of k_i vertices of $\mathcal{A}^x(S)$ is at most τ_i^x. Similarly, we compute the time instances τ_i^y, $1 \le i \le \log_B N$, so that the t-coordinate of k_i vertices of $\mathcal{A}^y(S)$ is at most τ_i^y. Set $\tau_0^x = \tau_0^y = -\infty$ and $\tau_i = \min\{\tau_i^x, \tau_i^y\}$ for $0 \le i \le \log_B N$. Define the slice $\mathcal{S}_i$ to be $\mathbb{R}^2 \times [\tau_{i-1}, \tau_i]$ for $1 \le i \le \log_B N$. $\mathcal{S}_i$ is guaranteed to contain less than $2NB^i$ events. Finally, as in the $1D$-case, we rebuild the data structure every $\Theta(NB)$ events and obtain the following.

Theorem 2. *Let S be a set of N points moving in the xy-plane with fixed velocities. S can be maintained in a data structure using $O(\log_B^3 N)$ expected I/Os per kinetic event so that a Q2 query at time t_q, with $NB^{i-1} \le \varphi(t_q) \le NB^i$, can be answered in $O(\sqrt{N/B^i}(B^{i-1} + \log_B N) + K/B)$ I/Os. The structure can be built in $O(N \log_2 N \log_B^2 N)$ expected I/Os and uses $O((N/B)\log_B N)$ disk blocks.*

Slice Structure. We now describe our slice data structure $\mathcal{SL}_i$. Answering a Q2 query corresponds to finding the lines traced out by S in xyt-space that intersect a rectangle R on the plane $t = t_q$ parallel to the xy-plane. Note that a line l intersects R if and only if their projections onto the tx- and ty-planes both intersect. See Figure 3 (ii). Let S^x denote the projection of S onto the tx-plane, and let $\mathcal{S}_i^x, \mathcal{S}_i^y$ be the projection of the slice $\mathcal{S}_i$ onto the tx- and ty-planes, respectively. As earlier, we define a set of *critical levels* λ_j of $\mathcal{A}(S^x)$ within the projected window $\mathcal{S}_i^x$ and store it in a persistent B-tree $\mathcal{T}_i$. We choose a random integer $\xi \in [0, \sqrt{NB^i}]$ and set λ_j to be the $(j\sqrt{NB^i} + \xi)$-level of $\mathcal{A}^x(S)$. We have $\sqrt{N/B^i}$ critical levels, and we define *bundle* $\mathcal{B}_j^i$ to be the $\sqrt{NB^i}$ levels between critical level λ_{j-1} and λ_j. Refer to Figure 4 (i). Using Lemma 4, we can prove that the total expected size of the critical levels is $O(N + NB^i/\sqrt{NB^i}) = O(N)$. For each bundle $\mathcal{B}_j^i$ we construct a $1D$-data structure on S^y, the projection of S onto the ty-plane, as follows.

Fix a bundle $\mathcal{B}_j^i$. A point $p \in S$ can leave and return to $\mathcal{B}_j^i$ several times, i.e., its x-projection may cease to lie between the levels λ_{j-1} and λ_j and then it may appear there again. Therefore, for a point $p \in S$, let $\triangle_1, \ldots, \triangle_r$ be the maximal time intervals, each a subset of $[\tau_{i-1}, \tau_i]$, during which p lies in $\mathcal{B}_j^i$. We map p to a set $S_p = \{e_1, \ldots, e_r\}$ of segments in the ty-plane, where $e_z = \bigcup_{t \in \triangle_z} p^y(t)$ is a segment contained in the ty-projection of the trajectory of p. Define $S_j^y = \bigcup_{p \in S} S_p$; set $N_j = |S_j^y|$. Since the endpoint of a segment in S_j^y corresponds to a vertex of λ_{j-1} or λ_j, $\sum_j N_j = O(N)$. We construct the window structure $\mathcal{WI}_j^i$ on S_j^y. Our construction of slices guarantees that there are $O(NB^i)$ vertices of $\mathcal{A}^y(S)$ in the ty-projection $\mathcal{S}_i^y$ of the slice $\mathcal{S}_i$. Therefore, the remark at the end of Section 3.3, implies that all the bundle structures use a total of $O(N/B)$ disk blocks and that they can be constructed in $O(N \log_2 N \log_B N)$ I/Os.

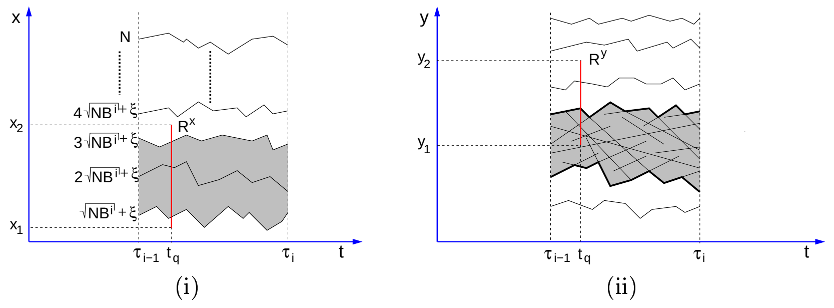

Fig. 4. (i) The $\sqrt{N/B^i}$ critical levels in the arrangement $\mathcal{A}(S^x)$ of the projection of S onto the tx-plane. (ii) One bundle in window structure built on the ty-projection of segments corresponding to points in bundle $\mathcal{B}_j$ in the tx-plane.

To answer a query Q2, we first use $\mathcal{T}_i$ to find the critical levels, and thus bundles, intersecting the tx-projection R^x of R in $O(\log_B N)$ I/Os. For all $O(\sqrt{N/B^i})$ bundles $\mathcal{B}_j$ completely spanned by R^x, we query the corresponding window structure to find all points also intersecting the yt-projection R^y of R using $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os (Lemma 5). Finally, we scan the (at most) two bundles intersected but not completely spanned by R^x using $O(\sqrt{NB^i}/B + K/B) = O(\sqrt{N/B^i} \cdot B^{i-1} + K/B)$ I/O and report the remaining points in R at time t_q. Refer to Figure 4 (i).

Lemma 6. *Let S be a set of N points moving in the xy-plane with fixed velocities, and let $\mathcal{S}_i$ be a time slice $\mathbb{R}^2 \times [\tau_{i-1}, \tau_i]$ that contains NB^i events. We can preprocess S into a data structure $\mathcal{SL}_i$ of size $O(N/B)$ in $O(N \log_2 N \log_B N)$ I/Os such that a Q2 query at time $\tau_{i-1} \leq t_q \leq \tau_i$ can be answered in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os.*

References

1. P. K. Agarwal, L. Arge, and J. Erickson, Indexing moving points, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 2000, pp. 175–186.
2. P. K. Agarwal, L. Arge, J. Erickson, P. Franciosa, and J. Vitter, Efficient searching with linear constraints, *Journal of Computer and System Sciences*, 61 (2000), 194–216.
3. P. K. Agarwal and J. Erickson, Geometric range searching and its relatives, in: *Advances in Discrete and Computational Geometry* (B. Chazelle, J. E. Goodman, and R. Pollack, eds.), *Contemporary Mathematics*, Vol. 223, American Mathematical Society, Providence, RI, 1999, pp. 1–56.
4. A. Aggarwal and J. S. Vitter, The Input/Output complexity of sorting and related problems, *Communications of the ACM*, 31 (1988), 1116–1127.
5. L. Arge, External memory data structures, in: *Handbook of Massive Data Sets* (J. Abello, P. M. Pardalos, and M. G. C. Resende, eds.), Kluwer Academic Publishers, 2001. (To appear).
6. L. Arge, V. Samoladas, and J. S. Vitter, On two-dimensional indexability and optimal range search indexing, *Proc. ACM Symp. Principles of Database Systems*, 1999, pp. 346–357.
7. J. Basch, L. J. Guibas, and J. Hershberger, Data structures for mobile data, *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, 1997, pp. 747–756.
8. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, An asymptotically optimal multiversion B-tree, *VLDB Journal*, 5 (1996), 264–275.
9. D. Comer, The ubiquitous B-tree, *ACM Computing Surveys*, 11 (1979), 121–137.
10. T. K. Dey, Improved bounds on planar k-sets and related problems, *Discrete Comput. Geom.*, 19 (1998), 373–382.
11. H. Edelsbrunner and E. Welzl, Constructing belts in two-dimensional arrangements with applications, *SIAM J. Comput.*, 15 (1986), 271–284.
12. L. J. Guibas, Kinetic data structures — a state of the art report, in: *Proc. Workshop Algorithmic Found. Robot.* (P. K. Agarwal, L. E. Kavraki, and M. Mason, eds.), A. K. Peters, Wellesley, MA, 1998, pp. 191–209.
13. G. Kollios, D. Gunopulos, and V. J. Tsotras, On indexing mobile objects, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 1999, pp. 261–272.
14. J. Matoušek, Randomized optimal algorithm for slope selection, *Inform. Process. Lett.*, 39 (1991), 183–187.
15. J. Matoušek, Efficient partition trees, *Discrete Comput. Geom.*, 8 (1992), 315–334.
16. D. Pfoser, C. S. Jensen, and Y. Theodoridis, Novel approaches to the indexing of moving objects trajectories, *Proc. International Conf. on Very Large Databases*, 2000, pp. 395–406.
17. G. Toth, Point sets with many k-sets, *Proc. 16th Annu. Symposium on Computational Geometry*, 2000, pp. 37–42.
18. P. J. Varman and R. M. Verma, An efficient multiversion access structure, *IEEE Transactions on Knowledge and Data Engineering*, 9 (1997), 391–409.
19. J. S. Vitter, Online data structures in external memory, *Proc. Annual International Colloquium on Automata, Languages, and Programming, LNCS 1644*, 1999, pp. 119–133.
20. S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, Indexing the positions of continuously moving objects, *Proc. ACM SIGMOD International Conference on Management of Data*, 2000, pp. 331–342.