Validity Information Retrieval for Spatio-Temporal Queries: Theoretical Performance Bounds

Yufei Tao[1], Nikos Mamoulis[2], and Dimitris Papadias[3]

[1] Department of Computer Science
Carnegie Mellon University, USA, 15213-3891
taoyf@cs.cmu.edu

[2] Department of Computer Science and Information Systems
University of Hong Kong , Hong Kong
nikos@csis.hku.hk

[3] Department of Computer Science
Hong Kong University of Science and Technology, Hong Kong
dimitris@cs.ust.hk

Abstract. The results of traditional spatial queries (i.e., range search, nearest neighbor, etc.) are usually meaningless in spatio-temporal applications, because they will be invalidated by the movements of query and/or data objects. In practice, a query result *R* should be accompanied with *validity information* specifying (i) the (future) time *T* that *R* will expire, and (ii) the change *C* of *R* at time *T* (so that *R* can be updated incrementally). Although several algorithms have been proposed for this problem, their worst-case performance is the same as that of sequential scan. This paper presents the first theoretical study on validity queries, and develops indexes and algorithms with attractive I/O complexities. Our discussion covers numerous important variations of the problem and different query/object mobility combinations. The solutions involve a set of non-trivial reductions that reveal the problem characteristics and permit the deployment of existing structures.

1 Problem Formulation

Traditional spatial query processing is insufficient in spatio-temporal applications, where the data and/or query objects change their locations. Consider, for example, a moving user who asks for the nearest hotel; the result (e.g., hotel *a*) of a conventional nearest neighbor query is by itself meaningless because it may be invalidated as soon as the user moves. In practice, the query result *R* should be accompanied by additional validity information: (i) the *expiry time T* of *R*, i.e., the future time when *a* ceases to be the nearest hotel (based on the user's moving direction and speed), and (ii) the result change *C* at *T* (e.g., the next nearest hotel after *a* expires). This problem is not specific to nearest neighbor search, but actually exists in all query types [TP02].

In this paper, we discuss validity information retrieval for the most common spatial queries, namely, (orthogonal) range search (RS) and nearest neighbor (NN) queries, considering one- or two- dimensional objects with linear movements. Following the common modeling in the literature [SJLL00, GBE+00], a moving point *p* is represented using its location $\mathbf{p}(0)$ at the system reference time 0, and its current velocity $\mathbf{v}$ ($\mathbf{p}(0)$ and $\mathbf{v}$ are 1D/2D vectors), such that its location $\mathbf{p}(t)$ at any future time *t* can be

computed as $\mathbf{p}(t)=\mathbf{p}(0)+\mathbf{v}\cdot t$. Similarly, we represent a moving rectangle with two moving points that decide its opposite corners, under the constraint that they have identical velocity vectors (i.e., the extent of the rectangle remains fixed at all times). Static objects are trivially captured with zero velocities. Without loss of generality, we assume (unless specifically stated) that the system reference time 0 coincides with the current time.

The result R of a spatial query is invalidated when some object "influences" it in the future, namely, (i) an object currently not in R starts qualifying the query predicate, or (ii) an object originally in R incurs predicate violation. A natural way to define validity retrieval is through the concept of "influence time" introduced in [TP02]. Specifically, given a RS (range search) query q with result R (containing all the objects currently covered by q), the *influence time* T_p of data point p in R, equals the earliest time that p falls out of q; on the other hand, for a point p not in R, T_p corresponds to the first timestamp that p lies in q. The concept of influence time also applies to NN (nearest neighbor) search. In this case, the influence time of a data point is the time when it will come closer to q than its current NN. Here the influence time should be interpreted as the time that the object will invalidate the query result R, only if it has not changed earlier. Now we are ready to formulate the problem of validity information retrieval.

Problem (Validity information retrieval): Given a set of points $S=\{p_1, p_2, ..., p_N\}$ and a spatial query at the current time 0, the corresponding *validity query* q returns (i) the expiry time $T=\min\{T_{p_i}\ (1\leq i\leq N)\}$, and (ii) the result change (at time T) $C=\{p_i\in S: T_{p_i}=T\}$, where T_{p_i} is the influence time of point p_i. ■

This paper presents the first study on the theoretical complexity of validity queries, aiming at solutions with good worst case performance. Our discussion is based on the popular memory-disk hierarchy [ASV99, AAE00], where each I/O access transfers a page of B (i.e., page size) units of information from the disk to the main memory, which contains at least B^2 pages (a reasonable assumption in practice). The query cost is measured as the number of disk pages visited. Our objective is to achieve fast query time with small space consumption.

1.1 Previous results

A study of validity queries appear in [TP02] which deploys branch-and-bound algorithms on R*-trees [BKSS90] (for static objects) and TPR-trees [SJLL00] (for dynamic objects). In the worst case, these algorithms perform $O(N/B)$ I/Os (i.e., the complexity of a simple sequential scan), where N is the number of objects in the dataset. All the other attempts [SR01, TPS02, BJKS02] addressing variations of the problem, incur the same complexity. On the other hand, there is a significant amount of theoretical results on conventional spatial queries for static and moving objects.

For *orthogonal range search* on static points, Kanth and Singh [KS99] prove that the best possible query time using any structure consuming linear $O(N/B)$ space is $O((N/B)^{1/2}+K/B)$ I/Os, where K is the number of objects retrieved. This bound is tight and has been realized by the *O-tree* [KS99] and the *cross-tree* [GI99]. Applying the theory of indexability [HKP97], Arge et al. [ASV99] show that a structure achieving optimal query cost $O(\log_B(N/B)+K/B)$ must occupy $\Omega((N/B)\log_B(N/B)/\log\log_B(N/B))$

space. They propose the *external range tree* that achieves these bounds. A special RS is the so-called *3-sided query*, where an edge of the query rectangle lies on the boundary of the data space. Arge et al. [ASV99] design the *external priority search tree* that answers such queries optimally (i.e., logarithmic query cost and linear space consumption). Earlier, non-optimal structures for RS and 3-sided queries can be found in [IKO87, KRVV96, RS94, SR95].

The first study on RS queries for moving objects [KGT99a] deals with only 1D data. Agarwal et al. [AAE00] present several interesting results in the 2D space following the *kinetic approach* [BGH97]. In particular, they show that if queries arrive in chronological order, a RS can be answered with the same time complexity as the static case (i.e., optimally) using the kinetic external range tree. They also give two time-responsive indexing schemes (later refined in [AAV01]) where the query cost depends on the difference between the query issue time and the current time.

Fewer results exist for nearest neighbor search in secondary memory. For static data, a Voronoi diagram can be constructed in $O(N\log N)$ time [BKOS97], after which a NN query is reduced to a point-location problem (i.e., identifying the Voronoi cell that contains the query point), which can be solved optimally using linear space and logarithmic I/O overhead [ADT03]. Agarwal et al. [AAE+00] design another solution that avoids computing the Voronoi diagram, and answers a NN query in $O(\log_B(N/B))$ time, but uses non-linear $(N/B)\log(N/B)$ space. Little work has been done for NN retrieval on moving objects in external memory. Kollios et al. [KGT99b] develop various schemes, but do not prove any performance bound. Finally, several solutions exist for the different but related problem of approximate nearest neighbor search [CGR95, GLM00, AAE00]. In this paper we focus on exact NN queries.

1.2 Our results

For static data (and moving queries), we show that a validity RS query can be answered optimally in $O(\log_B(N/B))$ I/Os using $O(N/B)$ space, for a constant number of query sizes and directions. Then, we discuss the problem where the query has arbitrary size but its movement is restricted to be axis-parallel, and develop a structure that consumes $O((N/B)\log_B(N/B))$ space and answers a query in $O(\log_B(N/B))$ I/Os. Based on the external range tree, we present another solution that occupies $O((N/B)\log_B(N/B)/\log\log_B(N/B))$ space and solves a query in $O(\log_B^2(N/B)/\log\log_B(N/B))$ I/Os. The persistent version of this structure answers a query with arbitrary size and moving direction with the same cost but higher space complexity $O((N^2/B)\log_B(N/B)/\log\log_B(N/B))$. All the above problems can also be solved using a modified partition tree with linear space and query overhead $O((N/B)^{1/2+\varepsilon})$, where ε is an arbitrarily small positive number. For moving data and static RS, a validity query can be answered with optimal $O(\log_B(N/B))$ I/O cost, using an index with $O((N^2/B)\log_B(N/B))$ space. If both objects and queries move on a linear space, we propose a solution that achieves $O(\log_B(N/B))$ query time and requires $O(N/B)$ space. Further, given a Voronoi diagram on static data, a validity NN query can be answered in $\log(N/B)$ time based on an optimal external memory structure for point location queries. The same complexity can also be achieved for one-dimensional dynamic queries and objects.

The rest of the paper is organized as follows. Section 2 reviews the previous indexes fundamental to our discussion, and elaborates the problems solved together with the corresponding query time and space complexities. Section 3 presents our results for validity RS queries, while Section 4 focuses on nearest neighbor search. Section 5 concludes the paper with a set of open problems.

2. Preliminary Structures

The *persistent* (also known as *multi-version*) B-tree [BGO+96] is an efficient storage scheme for a set of B-trees B_1, B_2, ..., B_β. The idea is to store in B_i ($2 \le i \le \beta$) only the "changes" from the previous version B_{i-1}, while sharing the index nodes that contain their common records. Particularly, in each node, there is either none or at least $\Theta(B)$ records for any B_i ($1 \le i \le \beta$), which guarantees that querying each B_i has the same complexity as a normal B-tree. As proven in [BGO+96], if the total number of distinct records in all B-trees (i.e., the total number of "changes" between consecutive trees) is $O(N)$, a persistent B-tree uses linear space $O(N/B)$ and can be constructed in $O((N/B)\log_B(N/B))$ I/Os [A94]. A range-query in any B-tree B_i ($1 \le i \le \beta$) is answered in $O(\log_B(N_i/B)+\log_B(\beta/B)+K/B) = O(\log_B(N/B)+K/B)$ I/Os, where N_i is the number of records in B_i, β the total number of B-trees, and K is the number of records retrieved. Both the space and query time complexities are optimal. As shown shortly, the persistent structure constitutes a powerful tool for our problem.

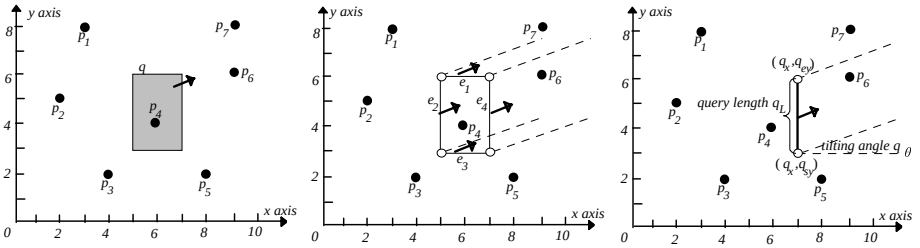
The *partition tree* [AAE00] is a popular 2D structure for the *simplex range query*, which specifies a constant number of half-planes and retrieves the set of points in their intersection (i.e., the search region). Given a set S with N points, the idea is to construct a *balanced simplicial partition* $\Lambda=\{\Lambda_1, \Lambda_2, ..., \Lambda_r\}$, where $r=O(B)$ and all points in Λ_i ($1 \le i \le r$) are covered by a triangle Δ_i. The partitions have the following properties: (i) $\Lambda_1 \cup \Lambda_2 \cup ... \cup \Lambda_r = S$, (ii) for any i, j in $[1,r]$, $\Lambda_i \cap \Lambda_j = \emptyset$ (i.e., the first two properties guarantee that any data point belongs to a unique partition), (iii) each Λ_i ($1 \le i \le r$) contains at most $2N/r$ points, and (iv) the number of triangles Δ_i crossed by *any* line l in the data space is $O(\sqrt{r})=O(\sqrt{B})$. Δ_1, Δ_2, ..., Δ_r constitute the entries in the root node of the partition tree. The next level is created by constructing balanced simplicial partitions for each Λ_i ($1 \le i \le r$), and this process is repeated until each partition contains $O(B)$ points, in which case all points in it are stored collectively as a leaf node. Since every point appears in a unique leaf node, the whole tree consumes $O(N/B)$ disk pages. A simplex range query is answered by visiting those nodes whose bounding triangles intersect its search region. Agarwal et al. [AAE+00] show that the query cost is bounded by $O((N/B)^{1/2+\varepsilon}+K/B)$ I/Os, where ε is an arbitrarily small positive number.

The *external priority search tree* [ASV99] answers a *3-sided query* q optimally. Specifically, given a set S of N 2D points, this tree consumes $O(N/B)$ space, and answers q in $O(\log_B(N/B)+K/B)$ I/Os, where K is the number of points retrieved. Interestingly, it also optimally solves *stabbing* queries on 1D intervals. Specifically, given a set of N 1D intervals $\{i_1, i_2, ..., i_N\}$ where $i_j=[i_{sj},i_{ej}]$ ($1 \le j \le N$), a stabbing query specifies a value q and retrieves all intervals i_j ($1 \le j \le N$) such that $i_{sj} \le q \le i_{ej}$. To solve the problem with the external priority search tree, each interval $i_j=[i_{sj},i_{ej}]$ is converted to a 2D point

(i_{sj},i_{ej}); the intervals satisfying q correspond to those points in the 3-sided rectangle with x-projection [0, q] and y-projection [q, ∞).

3. Validity Information Retrieval for Range Search

The result of a RS query q changes (as q or data move) when the boundary of q hits a data point p. Thus, we can retrieve, for each edge e_i ($1 \le i \le 4$) of q, the first point p_i that will be hit by e_i, together with the time t_i when this happens. The expiry time T of the current result then equals the smallest t_i ($1 \le i \le 4$), and the result change C is due to the point p_i with $t_i=T$ (i.e., T is the influence time of p_i). Therefore, a validity RS query can be reduced to four *dragging queries*[1] each specifying a moving horizontal/vertical line segment (the direction of movement can be arbitrary), and finds the first point it crosses. Consider, for example, Figure 3.1a with static data points p_1, p_2, ..., p_7 (similar observations also hold for dynamic objects). The validity RS query q is reduced to dragging queries e_1, e_2, e_3, e_4 as in Figure 3.1b, for which the first points hit are ∅, p_4, ∅, p_6, respectively (the result of the validity query is p_4 since it is crossed earlier than p_6). In the sequel we focus on the vertical dragging query q (the same solutions apply to horizontal queries by symmetry), whose initial position is a vertical line segment at $x=q_x$ with y-projection [q_{sy},q_{ey}] (represented as q_x:[q_{sy},q_{ey}]). A query is characterized using two parameters: (i) *query length* $q_L=q_{ey}-q_{sy}$, and (ii) *tilting angle* q_θ between the query's moving direction and the x-axis, as shown in Figure 3.1c using e_4 as an example. The subsequent sections solve vertical dragging queries in various problem settings, covering both static data points (Sections 3.1-3.3) and the dynamic ones (Sections 3.4, 3.5).



(a) Validity RS query q (b) Four dragging queries (c) Vertical dragging query

Figure 3.1: Reducing RS validity retrieval to dragging queries

3.1 Static data and predictable queries

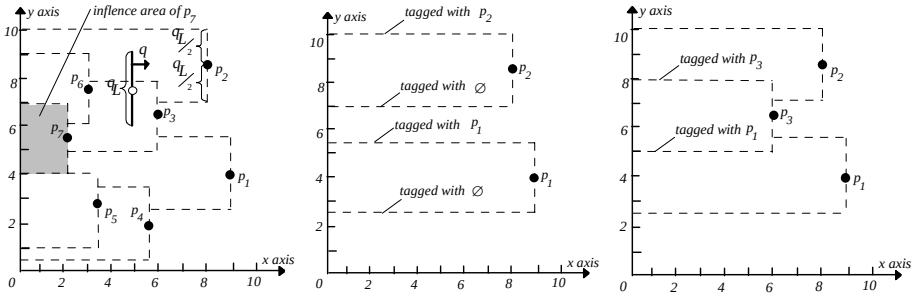
We first present an optimal solution that consumes linear space $O(N/B)$ and answers a "predictable" dragging query q in $O(\log_B(N/B))$ I/Os, namely, the query's length and tilting angle (of its movement) are chosen from a *constant* number of combinations[2].

[1] The term "dragging query" was first used in [C88] for orthogonal line segments with axis-parallel movements. Here, we use this term for queries with arbitrary moving directions.

[2] Note that, in practice where query's actual length and moving direction can be measured only discretely, the number of such combinations is indeed constant.

Our solution consists of a set of structures, each one targeting a specific combination. For simplicity, we use queries with length q_L moving towards the positive direction of the x-axis; the extension to the other directions is straightforward.

As shown in Figure 3.2a, our goal is to divide the space into disjoint *influence areas* according to the data points. Specifically, the influence area $A(p)$ of a point $p=(p_x,p_y)$ is a 3-sided rectangle whose projections on the x- and y- axes are $(-\infty, p_x)$ and $[p_y-q_L/2, p_y+q_L/2]$ respectively, where q_L is the targeted query length (e.g., the shaded area in Figure 3.2a represents $A(p_7)$). Furthermore, for two data points p_i, p_j such that $p_{ix}<p_{jx}$ (i.e., p_{ix} is to the "left" of p_{jx}), $A(p_i)$ "overwrites" $A(p_j)$ if they overlap (e.g., part of $A(p_6)$ is obstructed by $A(p_7)$). The crucial observation is that, given a dragging query $q=q_x:[q_{sy},q_{ey}]$, the first point p hit by q is the one whose influence area $A(p)$ covers the query center $(q_x,(q_{sy}+q_{ey})/2)$. As an example, the result of the query q=5:[6,9] in Figure 3.2a is p_3 because its center (the white point) at coordinates (5,7.5) falls into $A(p_3)$. Note that, if such an influence area does not exist, the query result is empty, namely, the dragging query will not hit any point.



(a) The final areas (b) After p_1, p_2 are inserted (c) After p_3 is inserted

Figure 3.2: Computing the influence areas

We use a persistent B-tree to index the influence areas. Specifically, the persistent B-tree contains a (logical) B-tree at each x-coordinate x, which stores the y-coordinates of the horizontal boundaries of the influence areas spanning x. For instance, the B-tree at x=5 stores 5 values {0.5, 3.5, 5, 8, 10}, corresponding to the (lower, upper, lower, upper, upper) boundaries of $A(p_4)$, $A(p_4)$, $A(p_3)$, $A(p_3)$, $A(p_2)$, respectively. To construct such a persistent tree, we insert the influence areas of points in descending order of their x-coordinates. In Figure 3.2b, p_1 is the first point processed, after which the persistent tree consists of a single logical B-tree (at the x-coordinate 9 of p_1) containing entries 3.5, 5.5 (i.e., the y-coordinates of the boundaries of $A(p_1)$). Entry 5.5 is "tagged" with p_1 (using constant space), indicating that the region below it belongs to $A(p_1)$, while entry 3.5 is tagged with ∅, meaning that the region below it is not in any influence area. Similarly, the next point p_2 inserts two entries 7, 10 (tagged with ∅, p_2 respectively) in the B-tree at x=8 (all the B-trees in the x-range (8,10] have the same content as the one at x=9). Handling the next point p_3 is more complex since its influence area overlaps $A(p_1)$ and $A(p_2)$. In this case, two entries 5.5 and 7 are removed from the B-tree at x=6, "terminating" the upper and lower boundaries of $A(p_1)$ and $A(p_2)$ respectively. Then, two entries 5, 8 (for the boundaries of $A(p_3)$) are inserted into the same tree, after which the tree contains 4 entries 3.5, 5, 8, 10 tagged with ∅, p_1, p_3, p_2 respectively. The remaining points are inserted in the same way. A dragging

query q=q_x:[q_{sy},q_{ey}] is directed to the B-tree at x=q_x, and finds the smallest entry larger than (q_{sy}+q_{ey})/2 (i.e., the y-coordinate of the query center). In Figure 3.2a, for example, the B-tree inspected is at x=5, and the entry returned has value 8. Then, the result (i.e., the point first hit by q) is the point tagged with the retrieved entry (p_3 in the case). The whole processing incurs $O(\log_B(N/B))$ I/Os.

Theorem 3.1: Given a dataset S of N static 2D points, we can pre-process S into a set of persistent B-trees that occupy totally $O(N/B)$ space and can be constructed in $O((N/B)\log_B(N/B))$ I/Os, such that a validity RS query q can be answered in $O(\log_B(N/B))$ I/Os, provided that the query rectangle's size and movement direction are decided from a constant number of combinations. ■

3.2 Static data and axis-parallel moving queries

In this section, we consider (vertical) dragging queries with arbitrary lengths but horizontal movements (i.e., zero tilting angles). It suffices to discuss queries moving towards the positive direction of the x-axis (by symmetry those towards the negative side can be solved in the same way). The first point hit by such a query q_x:[q_{sy},q_{ey}] is the one with the smallest x-coordinate in the 3-sided rectangle [q_x,∞]:[q_{sy},q_{ey}]. In Figure 3.3, for example, q=5:[1,8] and its 3-sided rectangle [5,∞]:[1,8] covers points p_1, p_3, p_4, among which p_4 has the smallest x-coordinate and hence, is the result of q. Next, we propose two solutions with different tradeoffs between query time and space.

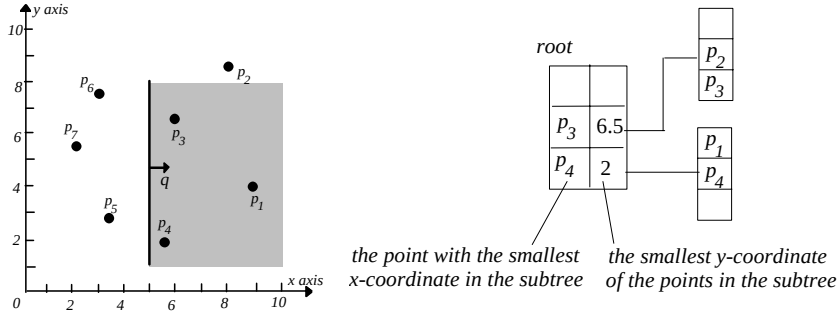
- **Using the persistent aggregate tree**

We maintain an aggregate B-tree (aB-tree) at every x-coordinate x, indexing the y-coordinates of all the points whose x-coordinates are larger than x. In addition to the search key, each non-leaf entry of the aB-tree also stores, using $O(1)$ space, the point with the smallest x-coordinate among all the points in its subtree. Figure 3.3b demonstrates the aB-tree at x=5, where the first root entry <p_4, 2>, for example, indicates that the smallest y-coordinate of all the data points (i.e., p_4, p_1) in its subtree is 2, and p_4 has the smallest x-coordinate. We store all the aB-trees in a *persistent aB-tree*, which as shown [TPZ02][3] consumes $O((N/B)\log_B(N/B))$ space and can be constructed in $O((N/B)\log_B(N/B))$ I/Os.

Given a query q_x:[q_{sy},q_{ey}], the aB-tree at x=q_x allows us to answer it in $O(\log_B(N/B))$ I/Os. To illustrate this, consider the query (5:[1,8]) in Figure 3.3a and the tree in Figure 3.3b. The algorithm starts from the root and descends a non-leaf entry if its "y-range" (enclosing the y-coordinates of points in its subtree) intersects, but is not totally contained in, that of the query ([1,8] in this case). For example, the y-range [2,6.5] of the first root entry in Figure 3.3b (where 6.5 is obtained from the second entry) is contained in [1,8], in which case we simply take point p_4 (stored in this entry) as the candidate result. The subtree of the second entry must be explored because its y-range [6.5,∞] partially intersects [1,8]. In its child node, we examine every data point and update our candidate result accordingly. In this case, p_3 has larger x-coordinate than p_4 (our current candidate) while p_2 does not fall in the 3-sided rectangle [5,∞]:[1,8] of the query; hence, no result update is necessary and the algorithm

[3] The structure in [TPZ02] is slightly different from the aB-tree in our case, but the complexity analysis still applies.

terminates by returning p_4. At each level of the aB-tree, the query y-range *partially* intersects those of at most two entries (notice that the y-ranges of the entries at the same level are disjoint); thus, the algorithm accesses (at most) two complete paths of the tree, i.e., incurring the same complexity as the tree height $O(\log_B(N/B))$.



(a) A dragging query (b) The aggregate B-tree at x=5

Figure 3.3: Dragging queries with arbitrary lengths

Theorem 3.2: Given a dataset S of N static 2D points, we can pre-process S into a set of persistent aB-trees that consume totally $O((N/B)\log_B(N/B))$ space, and can be constructed in $O((N/B)\log_B(N/B))$ I/Os, such that a validity RS query can be answered in $O(\log_B(N/B))$ time, provided that the moving direction of the query rectangle is axis-parallel. ■

- **Using the external range tree**

To decrease the space complexity, we present another solution based on a simplified version of the external range tree. Specifically, as shown in Figure 3.4 (for the data points in Figure 3.3a), the primary structure is a B-tree with fanout $O(\log_B(N/B))$ built on the y-coordinates of the data points (the height of the tree is thus $O(\log_B(N/B)/\log\log_B(N/B))$). Let $v_1, v_2, \ldots, v_r$ (r=3 in Figure 3.4) be the subtrees of an intermediate node v of the tree (e.g., the root of the B-tree). For each branch v_i ($1 \le i \le r$), let $|v_i|$ be the number of points it contains, and $p_{i_1}, p_{i_2}, ..., p_{i|v_i|}$ be these points sorted in ascending order of their x-coordinates $p_{i_1x}, p_{i_2x}, ..., p_{i|v_i|x}$. In Figure 3.4, the sorted lists for the root entries 2, 4, 7.5 are $\{p_5, p_4\}$, $\{p_7, p_3, p_1\}$, $\{p_6, p_2\}$, respectively. We define the *interval set* of v_i as $\{[-\infty, p_{i_1x}], [p_{i_1x}, p_{i_2x}], \ldots, [p_{i|v_i|x}, \infty]\}$, namely, the intervals produced by the projections of points in v_i on the x-axis.

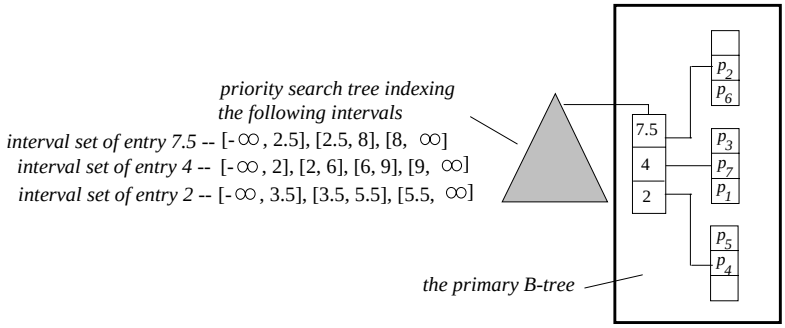


Figure 3.4: A simplified external range tree (for the dataset in Figure 3.3a)

For example, the interval sets for root entries 2, 4, 7.5 in Figure 3.4 are, {[−∞,3.5], [3.5,5.5], [5.5,∞]}, {[−∞,2], [2,6] [6,9], [9,∞]} and {[−∞,2.5], [2.5,8], [8,∞]}. The intervals in the interval sets of *all* v_i are indexed using a priority search tree associated with v. Note that, for the non-leaf nodes at the same level of the B-tree, their priority search trees index *disjoint* sets of intervals, the total number of which is $O(N)$. Given that the height of the primary B-tree is $O(\log_B(N/B)/\text{loglog}_B(N/B))$, the total size of the priority search trees (at all B-tree levels) is $O((N/B)\log_B(N/B)/\text{loglog}_B(N/B))$, which dominates the B-tree size $O(N/B)$ and constitutes the overall space complexity. This simplified external range tree can be constructed with $O((N/B)\log_B(N/B))$ I/Os, same as the complete external range tree [ASV99].

A right-moving dragging query $q=q_x:[q_{sy},q_{ey}]$ is answered as follows. At the root of the B-tree, we search its associated priority search tree for all the 1D intervals containing value q_x, which is a stabbing query as reviewed in Section 2. Given the query q=5:[1,8] (Figure 3.3a), for example, this search in the tree of Figure 3.4 returns intervals [3.5,5.5], [2,6], [2.5,8] (containing q_x=5). It is important to note that, each of these intervals comes from the interval sets of *distinct* subtrees ([3.5,5.5], [2,6], [2.5,8] are from root entries 2, 4, 7.5, respectively). Now, let us consider the data points corresponding to the second numbers of these intervals, namely, p_4, p_3, and p_2 (5.5, for example, is the x-coordinate of p_4). It is safe to conclude that p_3 is the point first hit by q, among all the points in the subtree of entry 4, whose y-range [4,7.5) is contained in that [1,8] of q. Similar conclusions, however, cannot be made for p_4 (p_2), originating from the subtree of root entry 2 (7.5), because its y-range [2,4) ([7.5,∞)) partially intersects [1,8]. In this case, we must visit the child nodes of these entries, where we discover point p_4 that is hit earlier than p_3, and becomes the final result.

In general, the above algorithm visits the nodes of the B-tree whose y-ranges *partially* intersect that of the query. Similar to the case of Figure 3.3b, the number of such nodes has the same complexity as the tree height, i.e., $O(\log_B(N/B)/\text{loglog}_B(N/B))$. At each node visited, a stabbing query is performed in its associated priority search tree, which returns as many intervals as the node fanout, i.e., $O(\log_B(N/B))$, incurring $O(\log_B(N/B)+\log_B(N/B)/B)=O(\log_B(N/B))$ I/Os (see the performance of the priority search tree in Section 2). As a result, the total query cost is bounded by $O(\log_B^2(N/B)/\text{loglog}_B(N/B))$ I/Os.

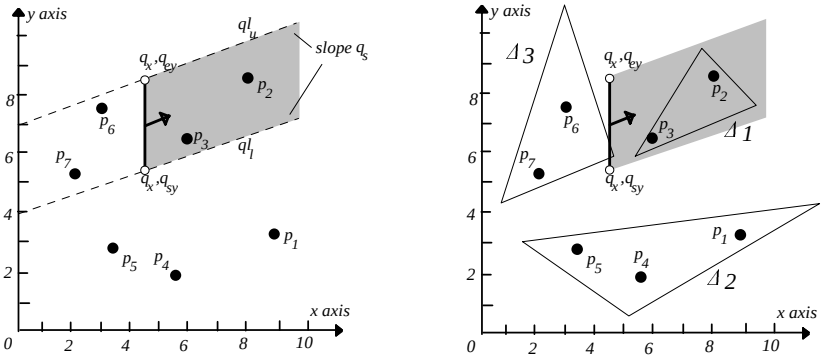
Theorem 3.3: Given a dataset S of N static 2D points, we can pre-process S into a set of simplified external search trees that consume $O((N/B)\log_B(N/B)/\text{loglog}_B(N/B))$ space and can be constructed in $O((N/B)\log_B(N/B))$ I/Os, such that a validity RS query q can be answered in $O(\log_B^2(N/B)/\text{loglog}_B(N/B))$ time, provided that the moving direction of the query rectangle is axis-parallel. ■

3.3 Static data and arbitrary queries

This section focuses on dragging queries with arbitrary lengths and tilting angles q_θ. Figure 3.5 shows an example query $q_x:[q_{sy},q_{ey}]$ whose moving direction has slope[4] $q_s=\text{tg}(q_\theta)$. Similar to Figure 3.3a, the goal is to find the point (i.e., p_3) with the smallest

[4] The slope is not defined for tilting angle $q_\theta=\pi/2$ and $3\pi/2$, namely, the movement of the query is vertical. This special case is solved in Section 3.2.

x-coordinate among the points in the shaded *3-sided parallelogram*. In the sequel, we present two solutions with different characteristics. It is worth mentioning that, these solutions also apply to the problems in the previous two sections, which are special instances of the queries discussed here.



(a) Dragging query and 3-sided parallelogram (b) The simplicial partitions

Figure 3.5: Handling dragging queries with non-axis-parallel movement

- **Using the partition tree**

As mentioned in Section 2, given a set of N data points, the partition tree answers a simplex query in $O((N/B)^{1/2+\varepsilon}+K/B)$ I/Os (where ε is an arbitrary positive number, and K is the number of points retrieved). Note that, the parallelogram produced by the dragging query (Figure 3.5a) is indeed a "simplex" shape, i.e., the intersection of three half planes. Applying the partition tree in this case, however, requires solving the following problem: since the goal is to find only one point in the parallelogram, we should avoid paying the extra cost K/B of reporting the other points.

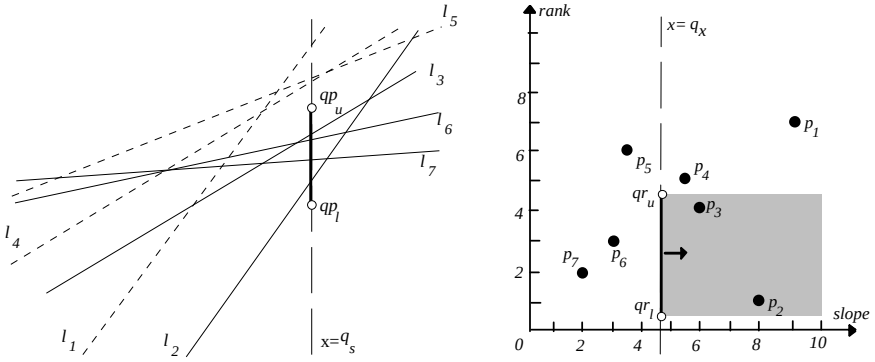
We solve this problem with a slight modification to the partition tree. In each non-leaf node Λ of the tree, we store, using $O(1)$ space for each child Λ_i (which, as described in Section 2, corresponds to a triangular simplicial partition Δ_i), the point with the smallest x-coordinate in its subtree. Figure 3.5b shows an example where the data points are divided into three simplicial partitions Δ_1, Δ_2, Δ_3, whose extents are stored in the root of the tree. The points stored for these partitions are p_7, p_3, p_5, respectively. This modification does not affect the space and building time complexities of the tree, which, however, can now solve the dragging query in $O((N/B)^{1/2+\varepsilon})$ I/Os. Specifically, the search starts from the root, and visits the subtrees whose simplicial partitions partially intersect the 3-sided parallelogram decided by the query. In Figure 3.5b, for example, Δ_1 is not accessed because it lies completely in the parallelogram, in which case the data point p_7 with the smallest x-coordinate is obtained directly from the corresponding root entry. Δ_2, on the other hand, must be visited because it partially intersects the shaded region (even though it does not contain any data point inside the parallelogram). Following an analysis similar to that in [AAE+00], we can derive the following theorem.

Theorem 3.4: Given a dataset S consisting of N 2D points, we can pre-process S into a modified partition tree that consumes $O(N/B)$ space such that a validity RS query can be answered in $O((N/B)^{1/2+\varepsilon})$ I/Os, for arbitrarily small $\varepsilon>0$. The partition tree can be constructed in $O((N/B)\log_B(N/B))$ expected I/Os. ■

- **Using the dual transformation**

We present another solution that reduces general dragging queries (with arbitrary tilting angles) to those with axis-parallel movements, using the dual transformation which converts (i) a point (p_x,p_y) in the original space to a line $y=p_x \cdot x-p_y$ in the dual space, and (ii) a line $y=a \cdot x-b$ to a dual point (a,b). Figure 3.6a illustrates the *dual data lines* for the points in Figure 3.5a. Given a dragging query q, let ql_l and ql_u be the lines representing the movements of the query segment's upper and lower end-points, respectively (see Figure 3.5a). Since they have the same slope q_s, their dual points qp_l and qp_u have the same x-coordinate q_s. The vertical line segment (we also refer to it as the *query segment*) in the dual space q_s:$[qp_{ly},qp_{uy}]$ connecting qp_l and qp_u has the following important property: it intersects a dual data line (in Figure 3.6a, the intersected lines are l_2, l_3, l_6, l_7) *if and only if* the corresponding data point (i.e., p_2, p_3, p_6, p_7) lies between ql_l and ql_u in the original space.

As shown in Figure 3.5a, the result (i.e., p_3) of the original dragging query q_x:$[q_{sy},q_{ey}]$ is the point with the smallest x-coordinate among those (i) that lie between lines ql_l and ql_u, and (ii) whose x-coordinates are larger than q_x. In the dual space, this is equivalent to finding the dual data line with the smallest slope among those (i) that intersect the vertical segment q_s:$[qp_{ly}, qp_{uy}]$, and (ii) whose slopes are larger than q_x. For this purpose, we perform yet another transformation, which converts a dual data line to a point in the *slope-rank space*. Specifically, the *ranks* of dual lines are defined according to their topological ordering at certain x-coordinate (lines may have different ranks at different x-coordinates). In Figure 3.6a, for example, the rank of l_2 (with slope 8) at x=q_s is 1, because its intersection with x=q_s is the lowest among the intersection points of all the lines; thus, it is mapped to (8,1) in the slope-rank space. Following the same idea, Figure 3.6b shows the converted points in the slope-rank space of all the lines (with respect to x=q_s) in Figure 3.6a.



(a) Dual space (b) Slope-rank space

Figure 3.6: Using two transformations to solve queries with non-axis-parallel movement

Accordingly, the query segment q_s:$[qp_{ly},qp_{uy}]$ in the dual space is converted into an other vertical line segment q_x:$[qr_l,qr_u]$ in the slope-rank space, where q_x is the x-coordinate of the original dragging query (in Figure 3.5a, q_x=3.5), and qr_l (qr_u) is the rank of point qp_l (qp_u) at x=q_s in the dual space. In Figure 3.6, qr_l is set to 0.5, indicating that qp_l is below all lines at x=q_s; similarly qr_u is assigned to 4.5, meaning that qp_u is between l_4 and l_3 at this x-coordinate. Note that, the values of qr_l and qr_u are not

unique; for example, qr_l can be any number between 4 (the rank of l_4) and 5 (the rank of l_3), while qr_u can be any number below 1 (the rank of l_2). The merit of this transformation is that, now it remains to find, in the slope-rank space, the point first hit by the right-moving vertical dragging query q_s:[qr_l,qr_u] (i.e., p_3, which as in Figure 3.5a is the correct answer to the original dragging query). This problem was solved in Section 3.2; we use the solution based on the simplified external range tree (Figure 3.4).

We have shown that a dragging query whose moving direction has *specific* slope q_s, can be solved using an external range tree at x=q_s in the dual space. In order to support all slopes, we must maintain such a tree at every x-coordinate. Fortunately, since we only care about the topological ordering of the dual lines, a new tree is necessary only when the ranks of two lines change. Motivated by this, we adopt the persistent version of the (simplified) external range tree, where a new tree is created at the x-coordinates where two dual lines cross each other. Towards this, the first step is to obtain the intersection points of all pairs of dual lines, using an I/O optimal algorithm for *line arrangement* computation [GTVV93]. Then the second step constructs the persistent tree in the ascending order of the intersections' x-coordinates. Since there are totally $O(N^2)$ intersections, the space occupied by the persistent tree is $O((N^2/B)\log_B(N/B)/\text{loglog}_B(N/B))$ and its construction incurs $O((N^2/B)\log_B(N/B))$ I/Os.

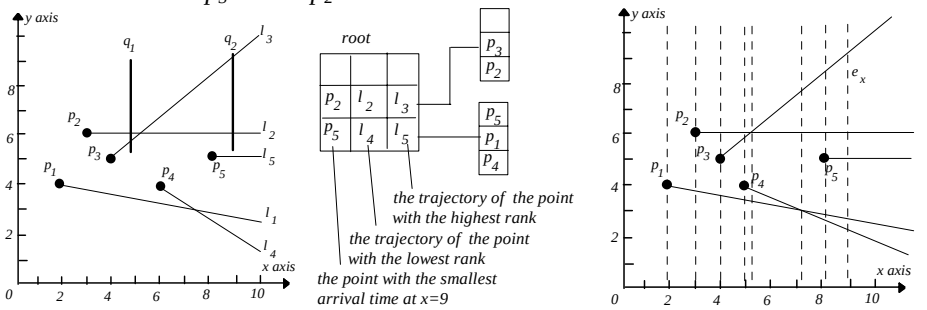
Further, in order to efficiently assign ranks to the end points of the query segment in the dual space (see Figure 3.6a), we maintain another persistent B-tree where each logical B-tree indexes the ranks of the lines at each x-coordinate in the dual space. This tree consumes $O(N^2/B)$ space, and can be built in $O((N^2/B)\log_B(N/B))$ I/Os. The query ranks can be assigned in $O(\log_B(N/B))$ I/Os, by searching the B-tree at x=q_s (i.e., the query slope). The total query cost, however, is dominated by that of searching the external range tree, which, as shown in Theorem 3.3, has complexity $O(\log_B^2(N/B)/\text{loglog}_B(N/B))$.

Theorem 3.5: Given a dataset S of N static 2D points, we can pre-process S into a set of index structures that consume $O((N^2/B)\log_B(N/B)/\text{loglog}_B(N/B))$ space and can be constructed in $O((N^2/B)\log_B(N/B))$ expected I/Os, such that a validity RS query q (with arbitrary window size and moving direction) can be answered in $O(\log_B^2(N/B)/\text{loglog}_B(N/B))$ time. ■

3.4 Dynamic data and static queries

Having solved dragging queries on static objects, in this section, we discuss dynamic data points assuming, however, static queries. As before, the goal is to find the first point that crosses the (static) query segment q=q_x:[q_{sy},q_{ey}]. For this purpose, we separate points moving towards the positive/negative direction of the x-axis, and process them independently. Due to the symmetry, it suffices to elaborate our solution for right-moving points. Figure 3.7a shows an example with 5 points p_1, p_2, ..., p_5 whose trajectories are represented as rays l_1, l_2, ..., l_5. For simplicity, assume that all points move at the same speed. Consider two dragging queries q_1, q_2 in Figure 3.7a, both of which intersect rays l_2 and l_3, but in different order. Specifically, for q_1, the first point hit is p_3, while for q_2 the first point is p_2. Such ordering determines the corresponding query result, and can be described using the concept of "arrival time". Specifically, for each data point p with x-coordinate p_x, its *arrival time* $at_p(x)$ for any $x \geq p_x$ is the future

timestamp such that p_x=x. Then, a dragging query q=q_x:[q_{sy},q_{ey}] is reduced to finding the point p with the smallest arrival time $at_p(q_x)$, among those whose trajectories intersect the query line segment. As an example, q_1 intersects p_2 and p_3, while p_3 is the final result since $at_{p_3}(q_{1x})<at_{p_2}(q_{1x})$.



(a) Object trajectories (b) The agg. B-tree at x=9 (c) x-coord. for new logical tree

Figure 3.7: Handling dynamic data

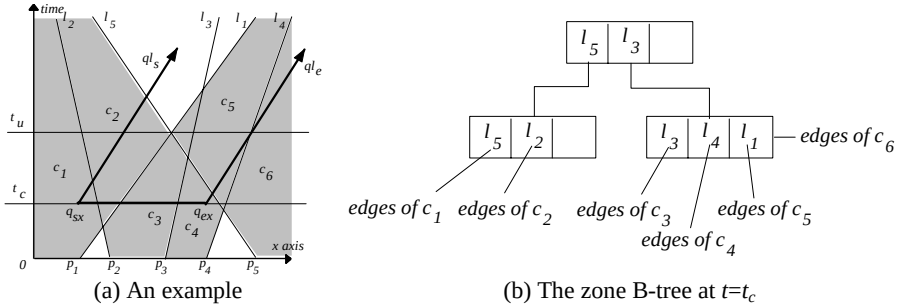
Based on this idea, we solve a dragging query using a persistent aggregate tree in logarithmic query cost. Specifically, the persistent tree maintains a logical aggregate B-tree at every x-coordinate x indexing all the data trajectories spanning x (e.g., in Figure 3.7 the logical tree at x=9 stores all rays), which are sorted according to their ranks (i.e., the topological ordering, similar to Figure 3.6) at x (e.g., at x=9, l_4, l_3 have ranks 1, 5 respectively). Furthermore, in each non-leaf entry (of the aggregate tree at x) we store the ray with the smallest arrival time at x among those in its subtree. In Figure 3.7b, the first root entry <p_5,l_4,l_5>, for example, indicates that at x=9, the point with the lowest (highest) rank in its subtree has trajectory l_4 (l_5), and the point with the minimal arrival time is p_5. Given a query q_x:[q_{sy},q_{ey}], the problem now is reduced to finding the point, in the aggregate tree at x=q_x, with the minimal arrival time among those intersecting the query segment which, using the same algorithm given in Section 3.2, can be solved in $O(\log_B(N/B))$ I/Os.

Constructing the persistent aggregate tree deserves further discussion. We deploy a plane-sweep that creates the logical trees from left to right. Specifically, a new logical tree is necessary at the x-coordinates where one of the following *events* occurs: (i) a new data point appears, (ii) the rays of two points intersect (i.e., the ranks of the rays change), and (iii) the arrival time of two points becomes equal. Figure 3.7c (i.e., the dashed lines) demonstrates all the events for the example of Figure 3.7a. Notice that the x-coordinate e_x of the last event (i.e., the right-most dashed line) is due to the fact that the arrival time of p_2 equals that of p_3, i.e., $at_{p_2}(e_x)= at_{p_3}(e_x)$. In general, any pair of points (p_i, p_j) defines an event (based on their arrival time), if $p_{ix}>p_{jx}$ and the speed of p_i is larger than that of p_j. It is easy to see that the number of all events is bounded by $O(N^2)$, and they can be obtained using an algorithm similar to the line arrangement computation given in [GTVV93].

Theorem 3.6: Given a dataset S of N moving 2D points (with arbitrary velocity), we can pre-process S into a set of aggregate B-trees that consume $O((N^2/B)\log_B(N/B))$ space and can be constructed in $O((N^2/B)\log_B(N/B))$ expected I/Os, such that a static validity RS query q (with arbitrary window size) is answered in $O(\log_B(N/B))$. ■

3.5 Dynamic data and dynamic queries

We discuss the general case where both the data and query are dynamic in the 1D space, and leave the 2D case for future work. Figure 3.8 shows the trajectories (l_1, l_2, ..., l_5) of 5 points p_1, p_2, ..., p_5 in the x-time space, and an RS query q with interval $[q_{sx},q_{ex}]$ at the current time t_c moving as indicated by the arrows. In this case, the corresponding (1D) dragging queries are two "rays" (i.e., ql_s and ql_e the figure) shooting from the end points of q, and each of them finds the point that is first encountered (p_2 and p_3 respectively). The final answer is the one hit earlier (i.e., p_3).



(a) An example (b) The zone B-tree at $t=t_c$

Figure 3.8: Handling moving data and queries

To solve the 1D dragging query, we only need to maintain the *zone at the current time* t_c, which consists of all the cells, intersecting line $t=t_c$, in the arrangement of the data points' trajectories. Figure 3.8 demonstrates the zone with shaded polygons c_1, c_2, ..., c_5. The zone has complexity $O(N)$ [BKOS97], meaning that the polygons that generate it have $O(N)$ edges (although a single polygon can have $O(N)$ edges in the worst case, the number of such polygons is $O(1)$). To answer a dragging query (e.g., ql_s in Figure 3.8), we first find the cell c_1 of the zone that covers the source (q_{sx}, $q_t=t_c$) of the query ray, which can be achieved in $O(\log_B(N/B))$ I/Os, by searching, with value q_{sx}, a B-tree (called the *zone B-tree* in the sequel) that indexes the topological ordering of the data trajectories at $t=t_c$. In Figure 3.8b, for example, the zone B-tree stores lines l_2, l_1, l_3, l_5, l_4, whose ranks at time t_c are in this order. Each trajectory is associated with a pointer to the cell (of the zone) to its left. In this example, the cell c_1 containing (q_{sx},q_t) is identified through the pointer stored in l_2. Then, a second search is performed to identify the (unique) edge in c_1 that is hit by ql_s. Since c_1 contains $O(N)$ edges, this can be accomplished in $O(\log_B(N/B))$ I/Os through, for example, a binary search. Thus, the total query time is $O(\log_B(N/B))$ I/Os.

The zone at the current time may become useless as the time progresses. Particularly, the zone changes at discrete timestamps (i.e., events) when the trajectories of two data points cross each other. Following the kinetic approach [BGH97], we manage these events in an event queue, and dynamically maintain the zone at the next event removed from the queue. In Figure 3.8, the next event is at timestamp t_u, at which time lines l_5 and l_1 change their ranks (i.e., ordering). To reflect this, we perform two deletions from the zone B-tree, corresponding to the old ranks of l_5 and l_1, and then two insertions for their new ranks. Furthermore, we also need to maintain the cells in the zone, specifically as shown in the figure, removing cell c_3 that is no longer in the zone, and inserting a new one which is enclosed only by l_5 and l_1. Finally, we

must insert two new events in the event queue, which correspond to the (future) intersection time of l_2 and l_5, l_1 and l_3, respectively. In general, a new event is created as a pair of lines become adjacent in the zone B-tree for the first time. Using an external priority queue [A94] as the event queue, handling an event incurs totally $\log_B(N/B)$ I/Os.

Theorem 3.7: Given a dataset S of N moving 1D points, we can pre-process S into an index structure that consumes $O(N/B)$ space and can be constructed in $O((N/B)\log_B(N/B))$ expected I/Os, such that a moving validity RS query q (with arbitrary window sizes and velocities) can be answered in $O(\log_B(N/B))$ I/Os. This structure can be updated in $\log_B(N/B)$ I/Os for each kinetic event. ■

4. Validity Information Retrieval for Nearest Neighbor

If the data points are static, validity information retrieval is simple using the Voronoi diagram defined by the objects. Specifically, given a moving query point q, we first find the Voronoi cell that contains q (i.e., a point location problem), and then identify the edge of the cell hit by the movement of q (discussed in Section 3.5). The point location problem in the secondary memory can be solved in $O(\log_B(N/B))$ I/Os, using a persistent B-tree, which (given that the Voronoi cell has complexity $O(N)$) consumes $O(N/B)$ space, and can be constructed in $O((N/B)\log_B(N/B))$ I/Os [ADT03].

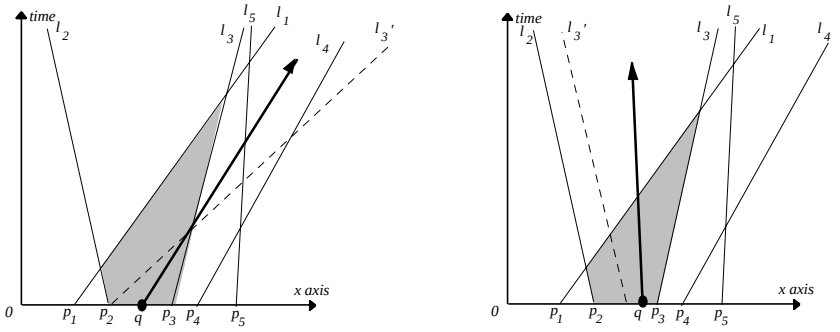
Theorem 4.1: Given a dataset S of N static 2D points and its Voronoi diagram, we can pre-process S into an index structure that consumes $O(N/B)$ space and can be constructed in $O((N/B)\log_B(N/\mathrm{B}))$ expected I/Os, such that a moving validity NN query q (with arbitrary moving direction) can be answered in $O(\log_B(N/B))$ I/Os. ■

Next, we consider the problem with moving objects in the 1D space (the 2D problem is left for future work). Figure 4.1a shows the trajectories l_1, l_2, l_3, l_4, l_5 of 5 points and a moving query q, whose current nearest neighbor is p_3. In order to find the point that will become the next nearest neighbor of q, we *mirror* l_3 to l_3' (i.e., the dotted line) such that the horizontal distance between l_3', q is the same as that between l_3, q for any future time t. In this example, the first points crossed by p_3 and p_3' (after the query time) are p_1 and p_5 respectively, among which p_5 is the next NN of q. Figure 4.1b shows another example, where the trajectories of q and its current NN p_3 do not intersect. We mirror l_3 to l_3' in the same way, and the first point hit by these two lines is p_1, the next NN of q. The following lemma states that the next NN of q always come from the first points hit by its current NN and its mirror.

Lemma 4.1: Given a set S of 1D moving points, let q be the query point whose nearest neighbor is $\rho \in S$. Let ρ' be the *mirror* of ρ with respect to q, such that $\forall t>0$, $dist(\rho(t),q(t))= dist(\rho'(t),q(t))$, where $x(t)$ denotes the position of point x at time t. Assume that p_1, p_2 are the first points in S that are crossed by ρ and ρ' respectively, and let t_1, t_2 be the time when these "crossings" occur. Then the next nearest neighbor of q is p_1 if $t_1<t_2$, or p_2 otherwise. ■

This lemma permits answering a validity NN query using logarithmic time in the same way as we solved the (1D) dragging query on dynamic data. Specifically, we maintain the zone B-tree as in Figure 3.8b so that, given a query point q at the current time, we can efficiently find the cell (in the zone of the current time) that contains q

(the shaded polygons in Figures 4.1a, 4.1b). Then, its current NN is decided from its neighboring points in the zone B-tree (in both examples of Figure 4.1, p_2, p_3 are the neighboring points, among which p_3 is the NN). Then, deciding the first points hit by the NN and its mirror involves inspecting at most two cells, e.g., two (one) cells in Figure 4.1a (4.1b) (note that the cell containing q is always visited). The zone B-tree and the cells of the zone at the current time can be updated using the kinetic approach as discussed in Section 3.5, with the same query time and space complexities.



(a) q intersects its NN (b) q does not intersect its NN

Figure 4.1: Validity NN query

Theorem 4.2: Given a dataset S of N moving 1D points, we can pre-process S into an index structure that consumes $O(N/B)$ space and can be constructed in $O((N/\mathrm{B})\log_B(N/\mathrm{B}))$ I/Os, such that a moving validity RS query q (with arbitrary moving direction) can be answered in $O(\log_B(N/B))$ I/Os. This structure can be updated in $\log_B(N/B)$ I/Os for each kinetic event. ■

5. Summary and Open Problems

Validity information retrieval aims at accompanying with expiry information the result of a traditional spatial query in a dynamic environment. Therefore, a validity query is usually executed together with the corresponding spatial query and the overall complexity is dominated by the more expensive of these two queries. Table 6.1 summarizes the results in this paper and illustrates the best performance for the corresponding traditional queries. Each cell includes the space complexity, followed by the query time. Using this table, we identify the following open problems:

- For validity RS queries on static data, is there a method which consumes linear space and answers a query in $O((N/B)^{1/2})$ I/Os? This would eliminate the extra $O((N/B)^{\varepsilon})$ query overhead and make the validity query as expensive as the traditional RS, achieving optimal performance using linear space.
- For validity RS queries on static data, is there a method which consumes space $O((N/B)\log_B(N/B)/\log\log_B(N/B))$ and answers an axis-parallel-moving query in $O(\log_B(N/B))$ I/Os? Such a method would achieve logarithmic query time using the minimal space.
- For static validity RS queries on dynamic data, devise a method which achieves logarithmic query cost with less space consumption than the current solution.

- Develop index structures for dynamic validity RS queries on dynamic 2D data with good worst case bounds.
- Design a solution that answers NN, and validity NN queries on dynamic 2D data.
- Determine the lower (space/query cost) bounds for the above problems.

query	traditional queries		validity queries
range search on static data	linear space	$O(N/B)$, $O((N/B)^{1/2}+K/B)$ [KS99, GI99]	$O(N/B)$, $O(\log_B(N/B))$ for constant query size and velocities
			$O(N/B)$, $O((N/B)^{1/2+\varepsilon})$
	non-linear space	$O((N/B)\log_B(N/B)/\log\log_B(N/B))$, $O(\log_B(N/B)+K/B)$ [ASV99]	$O((N/B)\log_B(N/B))$, $O(\log_B(N/B))$ for APM*
			$O((N/B)\log_B(N/B)/\log\log_B(N/B))$, $O(\log_B^2(N/B)/\log\log_B(N/B))$ for APM
			$O((N^2/B)\log_B(N/B)/\log\log_B(N/B))$, $O(\log_B^2(N/B)/\log\log_B(N/B))$
range search on dynamic data	linear space	$O(N/B)$, $O((N/B)^{1/2}+K/B)$ [ASV99]	NA
	non-linear space	$O((N/B)\log_B(N/B)/\log\log_B(N/B))$, $O(\log_B(N/B)+K/B)$ [ASV99]	$O((N^2/B)\log_B(N/B))$, $O(\log_B(N/B))$ for static queries
			$O(N/B)$, $O(\log_B(N/B))$ for 1D data
			NA for 2D data
NN on static data	$O(N/B)$, $O(\log_B(N/B))$ (Voronoi diagram plus point location queries)		$O(N/B)$, $O(\log_B(N/B))$
NN on dynamic data	$O(N/B)$, $O(\log_B(N/B))$ for 1D		$O(N/B)$, $O(\log_B(N/B))$ for 1D
	NA for 2D		NA for 2D

*APM= axis-parallel-moving queries

Table 6.1: Summary of the results

Acknowledgements

This work was supported from grants HKUST 6081/01E, HKUST 6197/02E and HKU 7380/02E from Hong Kong RGC.

References

[A94] Arge, L. The Buffer Tree: A New Technique for Optimal I/O Algorithms. WADS, 1994.

[AAE+00] Agarwal, P., Arge, L., Erickson, J., Franciosa, P., Vitter, J. Efficient Searching with Linear Constraints. Journal of Computer and System Sciences, 61(2): 194-216, 2000.

[AAE00] Agarwal, P., Arge, L., Erickson, J. Indexing Moving Points. ACM PODS, 2000.

[AAV01] Agarwal, P., Arge, L., Vahrenhold, J. A Time Responsive Indexing Scheme for Moving Points. Workshop on Algorithms and Data Structures, 2001.

[ADT03] Arge, L., Danner, A., Teh, S. I/O Efficient Point Location Using Persistent B-Trees. ALENEX, 2003.

[ASV99] Arge, L., Samoladas, V., Vitter, J. On Two-Dimensional Indexability and Optimal Range Search Index. ACM PODS, 1999.

[BGH97] Basch, J., Guibas, L., Hershberger, J. Data Structures for Mobile Data. ACM SODA, 1997.

[BGO+96] Becker, B., Gschwind, S., Ohler, T., Seeger, B. Widmayer, P. An Asymptotically Optimal Multiversion B-trees. VLDB Journal, 5(4): 264-275, 1996.

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. IDEAS, 2002.

[BKOS97] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. Computational Geometry. Springer, 1997.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. ACM SIGMOD, 1990.

[C88] Chazelle, B. An algorithm for segment-dragging and its implementation. Algorithmica, 3: 305-221, 1988.

[CGR95] Callahan, P., Goodrich, G., Ramaiyer, K. Topology B-Trees and Their Applications. Workshop on Algorithms and Data Structures, 1995.

[GBE+00] Guting, R., Böhlen, M., Erwig, M., Jensen, C., Lorentzos, N., Schneider, M., Vazirgiannis, M. A Foundation for Representing and Querying Moving Objects. ACM TODS, 25(1): 1-42, 2000.

[GI99] Grossi, R., Italiano, G. Efficient Splitting and Merging Algorithms for Order Decomposable Problems. Information and Computation, 154(1): 1-33, 1999.

[GLM00] Govindarajan, S., Lukovszki, T., Maheshwari, A., Zeh, N. I/O-Efficient Well-Separated Pair Decomposition and Its Applications. Annual European Symposium on Algorithms, 2000.

[GTVV93] Goodrich, M., Tsay, J., Vengroff, D., Vitter, J. External Memory Computational Geometry. IEEE FOCS, 1993.

[HKP97] Hellerstein, J., Koutsoupias, E., Papadimitriou, C. On the Analysis of Indexing Schemes. ACM PODS, 1997.

[IKO87] Icking, C., Klein, R., Ottmann, T. Priority Search Trees in Secondary Memory. GTCCS, 1987.

[KGT99a] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. ACM PODS, 1999.

[KGT99b] Kollios, G., Gunopulos, D., Tsotras, V. Nearest Neighbor Queries in Mobile Environment. STDBM, 1999.

[KRVV96] Kanellakis, P., Ramaswamy, S., Vengroff, D., Vitter, J. Indexing for Data Models with Constraints and Classes. Journal of Computer and System Sciences, 52(3): 589-612, 1996.

[KS99] Kanth, K., Singh, A. Optimal Dynamic Range Searching in Non-Replicating Index Structures. ICDT, 1999.

[RS94] Ramaswamy, S., Subramanian, S. Path Caching: A Technique for Optimal External Searching. ACM PODS, 1994.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. ACM SIGMOD, 2000.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. SSTD, 2001.

[SR95] Subramanian, S., Ramaswamy, S. The P-Range Tree: A New Data Structure for Range Searching in Secondary Memory. ACM SODA, 1995.

[TPZ02] Tao, Y., Papadias, D., Zhang, J. Aggregate Processing of Planar Points. EDBT, 2002.

[TP02] Tao, Y., Papadias, D. Time-Parameterized Queries for Spatio-Temporal Databases. ACM SIGMOD, 2002.

[TPS02] Tao, Y., Papadias, D., Shen, Q. Continuous Nearest Neighbor Search. VLDB, 2002.