Range-Aggregate Queries Involving Geometric Aggregation Operations

Saladi Rahul, Ananda Swarup Das, K. S. Rajan, and Kannan Srinathan
{srahul,anandaswarup}@research.iiit.ac.in {rajan, srinathan}@iiit.ac.in

International Institute of Information Technology, Hyderabad, India

Abstract. In this paper we consider *range-aggregate* query problems wherein we wish to preprocess a set S of geometric objects such that given a query orthogonal range q, a certain aggregation function on the objects $S' = S \cap q$ can be answered efficiently. Range-aggregate version of point enclosure queries, 1-d segment intersection, 2-d orthogonal segment intersection (with/without distance constraint) are revisited and we *improve* the existing results for these problems. We also provide *semi-dynamic (insertions)* solutions to some of these problems. This paper is the first attempt to provide dynamic solutions to problems involving geometric aggregation operations.

1 Introduction

Range Searching is an extensively studied problem in the field of Computational Geometry and database communities due to its wide range of applications. In this paper we shall consider *range-aggregate query* problems [8] in which we deal with composite queries involving more than just a simple range searching or counting. Formally, we wish to preprocess a set of geometric objects S, such that given a query range q, a certain aggregation function that operates on the objects of $S' = S \cap q$ can be performed efficiently.

As an example consider the following problem: "Preprocess a set S of weighted points in $\mathbb{R}^2$ such that given a query rectangle q, the point in $S \cap q$ with the "maximum" weight can be reported efficiently" [2]. Other example aggregate functions include "minimum", "sum", "count". All these aggregate functions come under the class of Distributive aggregates which can be computed by partitioning the input into disjoint sets, aggregating each set individually and then obtaining the final result by further aggregation of the partial results. Algebraic and holistic aggregates are the other two classes. Algebraic aggregates (e.g. average) can be expressed as a function of distributive aggregates. Holistic aggregates (e.g. median) cannot be computed by dividing the input into parts. Aggregation functions can be geometric in nature like horizontal-vertical segment intersection, convex hull etc. [6, 8, 11].

In on-line analytical processing (OLAP), geographic information systems (GIS) and other applications range aggregate queries play an important role in summarizing information [12]. In a VLSI layout editing environment and in map overlaying processing range-aggregate query problems with geometric functions are very useful. Refer [8] for detailed explanation of these applications.

1.1 Our Contribution

In this paper we revisit the range-aggregate query problems with geometric functions which were previously attempted in [6] and [8]. We come up with improved solutions to these problems. The results are shown and compared with previous results in Table 2. The problems discussed in these papers were static. In this paper we come up with semi-dynamic (insertions) solutions to some of these problems. To the best of our knowledge, this is the first attempt being made at finding dynamic solutions to problems involving geometric aggregation operations. See Table 1 for the results obtained for semi-dynamic case. In Section 2, we consider static d-dimensional range-aggregate point enclosure problem ($d \geq 1$). In Section 3 we show to how to handle insertions efficiently for this problem. In Section 4, the static and the semi-dynamic (insertions) 1-d range-aggregate segment intersection problem is discussed. In Section 5, the static 2-d range-aggregate orthogonal segment intersection problem is discussed. Finally, in Section 6 the static 2-d range-aggregate orthogonal segment intersection with distance constraint is discussed.

Underlying space	Objects in S	Objects in T	Query	Space	Query time	Insert time (amortized)
$\mathbb{R}^1$	points	segments	point enclosure	$O(n \log n)$	$O(\log n + k)$	$O(\log n)$
$\mathbb{R}^d$	points	hyper rectangles	point enclosure	$O(n \log^d n)$	$O(\log^{d-1} n \log\log n + k)$	$O(\log^d n)$
$\mathbb{R}^1$	segments	segments	intersections	$O(n \log n)$	$O(\log n + k)$	$O(\log n)$

Table 1. Summary of results for *semi-dynamic* range-aggregate query problems; the query q is an orthogonal range; k is the output size. Insertion time mentioned in *amortized* time.

2 Static Range-Aggregate Point Enclosure

In this section we shall consider the static d-dimensional range-aggregate version of the point enclosure problem. A set S of points and a set T of orthogonal hyperboxes in $\mathbb{R}^d$ ($d \geq 1$) are given and we need to report all point-hyperbox incidences inside a query orthogonal hyperbox.

Problem 1. Preprocess a set S of points and a set T of orthogonal (axes-parallel) hyperboxes in $\mathbb{R}^d$ ($d \geq 1$) with $|S| + |T| = n$, such that given a query orthogonal hyperbox $q = [a_1, b_1] \times [a_2, b_2] \times \ldots [a_d, b_d]$, all pairs (s, t), $s \in S$, $t \in T$ satisfying $s \in (t \cap q)$ can be reported efficiently.

We shall start by considering the problem in a 1-dimensional scenario. Next, we show how to extend the solution to higher dimensions ($d \geq 2$).

Underlying space	Objects in S	Objects in T	Query	Space	Query time	Source
$\mathbb{R}^1$	points	segments	point enclosure	$O(n \log n)$	$O(\log n + k)$	[8]
				$O(n)$	$O(\log n + k)$	New
$\mathbb{R}^d$	points	hyper rectangles	point enclosure	$O(n \log^d n)$	$O(\log^d n + k)$	[8]
				$O(n \log^{d-1} n)$	$O(\log^{d-1} n + k)$	New
$\mathbb{R}^1$	segments	segments	intersections	$O(n \log n)$	$O(\log n + k)$	[8]
				$O(n)$	$O(\log n + k)$	New
$\mathbb{R}^d$	horizontal segments	vertical segments	intersections	$O(n \log^2 n)$	$O(\log^2 n + k)$	[8]
				$O(n \log n)$	$O(\log n + k)$	New
$\mathbb{R}^1$	points	segments	point enclosure with distance constraint	$O(n \log n)$	$O(\log n + k)$	[6]
				$O(n)$	$O(\log n + k)$	New
$\mathbb{R}^d$	horizontal segments	vertical segments	intersections with distance constraint	$O(n \log^3 n)$	$O(\log^2 n + k)$	[6]
				$O(n \log^2 n)$	$O(\log^2 n + k)$	New

Table 2. Comparison of results of *static* range-aggregate query problems; the query is an orthogonal range; k is the output size.

2.1 One-dimensional scenario.

Preprocess a set S of points and a set T of segments on the x-axis, with $|S|+|T| = n$ such that given a query interval $q = [a_1, b_1]$, all pairs (s, t), $s \in S$, $t \in T$ satisfying $s \in t \cap q$ can be reported efficiently.

We sort the points in S in non-decreasing order and remove any point in S which does not stab (or intersect) any segment in T. The reduced set $S' \subseteq S$ is stored at the leaves of a balanced binary search tree BST. BST is searched with segments $t \in T$ to find if any point in S stabs t. If no such point exists for a segment $t \in T$, then it is removed. Let $T' \subseteq T$ be the reduced set. The segments in T' partition the x-axis into $2|T'| + 1$ *elementary intervals*. It might happen that some of these intervals are empty. Let I be the set of these elementary intervals. With each interval $i \in I$, we maintain a list (L_i) of segments, $t' \in T'$, such that $t' \cap i \neq \emptyset$. Also, with each leaf point in BST, we maintain a pointer to that interval in I which it stabs (each point can stab only one interval in I).

The total size of all the lists L_i, $\forall\ i \in I$ will be $O(n^2)$. Notice that if we add up the total number of changes occurring in every pair of consecutive lists L_i and L_{i+1}, it will turn out to be $O(n)$. The reason being that changes between consecutive lists occur either due to an entry of a new segment $t' \in T'$ or removal of a segment $t' \in T'$ and the number of endpoints in T' are $2|T'| \equiv O(n)$. Hence, we make use of this fact and build a partially persistent data structure D [7], instead of separately storing the lists L_i, $\forall\ i \in I$. We start with an initially empty structure D and by treating the x-axis as time, we store the lists L_i, $\forall\ i \in I$ into D. Since, the number of modifications will be $O(n)$, the total size of D will be $O(n)$.

Given a query interval $q = [a_1, b_1]$, BST is searched and the all the points in S' lying within q are found out. For each reported point p, the pointer stored with it is followed to reach the elementary interval $i \in I$ it is stabbing. Then, the tuples (p, l_i), $\forall\ l_i \in L_i$, are reported. The list L_i is obtained by accessing D.

Theorem 1. *A set S of points and a set T of segments on the x-axis where $|S|$ + $|T| = n$, can be preprocessed into a data structure of size $O(n)$ such that given a query interval $q = [a_1, b_1]$, all pairs (s, t), $s \in S$, $t \in T$ such that $s \in t \cap q$ can be reported in time $O(\log n + k)$ where k is the output size.*

2.2 Extension to higher dimensions ($d \geq 2$).

We begin with $d = 2$ and then generalize that solution to higher dimensions. First, we discard all the points in S which do not stab any rectangle in T. The remaining points in S (say S') are put in a two-dimensional Range Tree [3] RT. One way of solving this problem is to maintain with each point in S' the list of rectangles in T that it stabs. Then given a query rectangle q, we shall query RT with q and for each point inside q, report the rectangles it stabs. However, the space required in this case can blow up to $O(n^2)$. We overcome this issue by maintaining a sparse list with each point in S'.

Next we will have to build a new data structure that solves the standard “point enclosure” problem of reporting all the rectangles stabbed by a query point. Based on the x-projections of the rectangles in T, we build a segment tree ST. Let $I(v)$ be the set of segments allocated to a node v in ST. At each node $v \in ST$, based on the y-projection's of the rectangles whose x-projections are in $I(v)$, we build an instance of data structure D of Theorem 1 for storing the y-projection's. Hence, ST can be used for answering a standard point enclosure problem, for a given query point in $\mathbb{R}^2$. The space occupied by the Range Tree RT and the augmented Segment Tree is $O(n \log n)$.

Consider a point $p \in S$. Do a stabbing query on only the primary structure of ST. Let $\Pi(p)$ be the path from root till the leaf obtained by querying with p. At each node $v \in \Pi(p)$, if $I(v) \neq \emptyset$ query the secondary structure and find out the elementary interval within which point p lies. Make a list $L(p)$ storing the pointer to the elementary interval in which p lies $\forall\ v \in \Pi(p)$. The list $L(p)$ is prepared for each point $p \in S$. The total size of the lists $L(p)$, $\forall\ p \in S'$ will be $O(n \log n)$. In this way we have successfully reduced the space complexity from $O(n^2)$ to $O(n \log n)$.

Given a query rectangle $q = [a_1,\ b_1] \times\ [a_2,\ b_2]$, we query RT with q. For each point $p \in q$, we follow all the pointers in $L(p)$. By following each pointer we reach an elementary interval, say i. Then we start reporting pairs (p, r) where r is the rectangle corresponding to each segment stored in L_i (L_i is the set of segments stored corresponding to the elementary interval i). The time taken to query RT is $O(\log n\ +\ k')$, where k' is the number of points reported by RT and the time taken to report all the k pairs is $O(k)$. Therefore, the total query time is $O(\log n\ +\ k)$.

This solution can be directly extended to higher dimensions ($d > 2$). In a d-dimensional space, RT will be a d-dimensional Range Tree built on points in S. ST will be a d-dimensional Segment tree. However, at the deepest level we shall replace a segment tree by an instance of the structure D of Theorem 1. By doing a stabbing query on ST each point $p \in S$ will create a list $L(p)$ of size $O(\log^{d-1} n)$ size. The size of RT and ST will be $O(n \log^{d-1} n)$. The time taken to answer for a query hyperbox will be $O(\log^{d-1} n + k)$.

Theorem 2. *A set S of points and a set T of axes-parallel hyperboxes in $\mathbb{R}^d$ for $d \geq 2$ with $|S| + |T| = n$, can be preprocessed into a data structure of size $O(n \log^{d-1} n)$ such that given a query hyperbox $q = [a_1, b_1] \times [a_2 , b_2] \times \ldots \times [a_d , b_d]$, all pairs (s, t), $s \in S$, $t \in T$ such that $s \in (t \cap q)$ can be reported in time $O(\log^{d-1} n + k)$ where k is the output size.*

3 Semi-dynamic Range-Aggregate Point Enclosure

In this section we shall build data structures which can handle insertions efficiently for solving the Range-Aggregate Point Enclosure problem. As done previously, the one-dimensional scenario is presented first and then extended to higher dimensions.

3.1 One-dimensional scenario.

The preprocessing steps are as follows. Using the segments in T, a standard segment tree ST is built. Structure ST is equipped to handle both the reporting and the counting queries (query here will be a point). Construction of the segment tree takes $O(n \log n)$ time and space. Point set S is divided into three disjoint subsets S_1, S_2 and S_3. If a point $p \in S$ intersects with none of the segments in T, then p falls into set S_1. If the number of segments of T with which p intersects lies in the range $(0, \log n)$, then it falls into set S_2. $S_3 \subseteq S$ contains the points which intersect with more than "$\log n$ - 1" segments of T, i.e., in the range $[\log n, n]$. Segment tree ST is used for finding the number of segments of T being intersected by each point in S. This partition of point set S into three subsets takes $O(n \log n)$ time.

Based on the x-coordinates of the points in S_i ($\forall\, i$=1,2,3), we build a balanced binary search tree BT_i. We pick BT_2 for further augmentation while BT_1 and BT_3 are not augmented further. The points in S_2 are placed at the leaf nodes of BT_2. With each point $p \in S_2$ present at a leaf node of T_2, we store a list, L_p, of all the segments of T it intersects. Also, the size of each list L_p is maintained. The list L_p can be found by querying ST, which will take time $O(\log n + |L_p|) \equiv O(\log n)$ since $|L_p| \in (0, \log n)$. Therefore, the time taken to augment BT_2 will be $O(n \log n)$. The time taken to build the trees BT_1, BT_2 and BT_3 is $O(n \log n)$. The total preprocessing time is $O(n \log n)$. The space occupied is $O(n \log n)$ since both BT_2 and ST take up $O(n \log n)$ space.

Given a query interval q=$[a_1, b_1]$, BT_3 is queried with q. For each point p in BT_3 lying within q, we query ST with p and report all the pairs (p, t) satisfying

$p \in (t \cap q)$ and $t \in T$. Next, we query BT_2 with q. For each point p in BT_2 lying within q, we shall report the pairs (p, t), $\forall\ t \in L_p$.

The query time of the algorithm is analyzed. The time taken to query BT_2 is $O(\log n + \Sigma|L_p|) \equiv O(\log n + k')$, where k' is the number of pairs formed by points of S_2 lying within q. Let k_p be the number of segments intersected by point $p \in (S_3 \cap q)$. Then the time taken to query BT_3 will be: $O(\Sigma\ (\log n + k_p)) \equiv O(\Sigma k_p) \equiv O(k")$, since $k_p \geq \log n$ and $k"$ be the number pairs formed by points of S_3 lying within q. Therefore, the total query time will be $O(\log n + k' + k") \equiv O(\log n + k)$, where k is the total number of pairs to be reported.

Handling insertions. Suppose a new point p is added to the set S. It is first queried on ST to find out the number of segments of T it intersects with (say k_p). Based on the value of k_p it is kept in one of the subsets S_1, S_2 or S_3 and then inserted appropriately into one of the binary trees BT_1, BT_2 or BT_3. If p is inserted into BT_2, then the list L_p of the segments of T it intersects is also prepared. Thus insertion of a point p can be handled in $O(\log n)$ time.

Now, suppose a new segment t is to be added to the set T. t is first inserted into the segment tree ST. This takes $O(\log n)$ amortized time. Next, BT_2 is queried with t. For each point $p \in BT_2$ which intersects t, t is added to the list L_p. If $|L_p| = \log n$, then p is shifted from set S_2 to S_3. p and its list L_p is deleted from BT_2, and p is inserted into BT_3. Let λ_1 be the number of points in BT_2 which are intersected by p and λ_2 be the no. of points shifting from BT_2 to BT_3. Then the time taken to update the lists in BT_2 and the shifting process from BT_2 to BT_3 takes $O(\log n + \lambda_1 + \lambda_2 \log n)$. Then BT_1 is queried with t. For each point $p \in BT_1$ which intersects t, p is deleted from BT_1 and inserted into BT_2. In BT_2, the list L_p is initialized for point p with t being the only entry in it. Let λ_3 be the number of points shifting from BT_1 to BT_2. This will take $O(\log n + \lambda_3 \log n)$ time. Therefore, the total time for inserting a new segment t is $O(\log n + \lambda_1 + \lambda_2 \log n + \lambda_3 \log n)$.

An amortized analysis is carried out to get an efficient bound. Assume that we insert n segments and points in an arbitrary order. Notice that a point in set S_1 can jump only once into set S_2 and a point in set S_2 can jump only once into set S_3. Therefore, the value of $\Sigma\lambda_2$ and $\Sigma\lambda_3$ where the summation is over n insertions are bounded by $O(n)$. A point in S can remain in the set S_2 till it intersects with less than $\log n$ segments. Also, the number of points in set S after n insertions is still bounded by $O(n)$. Therefore, the value of $\Sigma\lambda_1$ where the summation is over n insertions is $O(n \log n)$. Therefore, the total time taken for insertion of n segments and points is: $O(\Sigma(\log n + \lambda_1 + \lambda_2 \log n + \lambda_3 \log n)) \equiv O(n \log n)$. The amortized time turns out to be $O(\log n)$.

Ater n insertions of points and segments, the whole data structure is deleted and reconstructed. After n insertions, the total number of points and segments become $2n$. Therefore, we shall update the criteria for a point p to enter set S_2 and S_3 from $(0, \log n)$ to $(0, \log 2n)$ and $[\log n, n]$ to $[\log 2n, 2n]$, respectively. Since, the preprocessing time is $O(n \log n)$ when built on n points and segments, the amortized time of insertion does not change.

Theorem 3. *A set S of points and a set T of segments on the x-axis where $|S|$ + $|T| = n$, can be preprocessed into a data structure of size $O(n \log n)$ such that given a query interval $q = [a_1, b_1]$, all pairs (s, t), $s \in S$, $t \in T$ such that $s \in t \cap q$ can be reported in time $O(\log n + k)$ where k is the output size. Also, insertion of a point or a segment can be handled in $O(\log n)$ amortized time.*

3.2 Extending it to higher dimensions.

Extending our solutions to higher dimensions turns out to be a straightforward process. In $\mathbb{R}^d$, we have a set S of d-dimensional points and a set T of d-dimensional orthogonal hyperboxes. A dynamic d-dimensional segment tree ST [4] is built based on the hyperboxes in set T. Set S is again divided into three disjoint sets S_1, S_2 and S_3. If a point intersects no hyperbox of T then it goes into set S_1, if the number of intersections is in the range (0, $\log^d n$) then the point goes into set S_2 and finally if the point intersects with more than "$\log^d n$ - 1" hyperboxes then it goes into set S_3. Balanced binary trees BT_1, BT_2 and BT_3 are replaced by dynamic range trees [4] which can answer range queries in d-dimensional space. A dynamic range tree when built on m points occupies $O(m \log^{d-1} m)$ space and answers queries in $O(\log^d m + k)$ time. It handles insertions and deletions in $O(\log^d m)$ amortized time. With each point $p \in BT_2$ (or S_2) list L_p is prepared. Appropriate pointers are maintained by points in BT_2 to their lists. As done previously, after n insertions of points and hyperboxes, the whole data structure is deleted and reconstructed.

The total space occupied will be $O(n \log^d n)$ as BT_2 and ST occupy $O(n \log^d n)$ space. The query time will be $O(\log^d + k)$ (the analysis done for 1-d case holds here as well). The time taken to insert a point or a hyperbox is $O(\log^d n)$ amortized time. Using fractional cascading [10], the query time can be reduced to $O(\log^{d-1} n \log\log n + k)$.

Theorem 4. *A set S of points and a set T of axes-parallel hyperboxes in $\mathbb{R}^d$ for $d \geq 2$ with $|S|$ + $|T|$ = n, can be preprocessed into a data structure of size $O(n \log^d n)$ such that given a query hyperbox $q = [a_1, b_1] \times [a_2 , b_2] \times \ldots \times [a_d , b_d]$, all pairs (s, t), $s \in S$, $t \in T$ such that $s \in (t \cap q)$ can be reported in time $O(\log^{d-1} n \log\log n + k)$ where k is the output size. Insertion of a new point or a hyperbox takes $O(\log^d n)$ amortized time.*

4 1-d Range-Aggregate Segment Intersection

Problem 2. Preprocess a set S of n segments on the x-axis, such that given a query interval $q = [a_1, b_1]$, all pairs (s, t), $s \in S$, $t \in S$ satisfying $s \cap t \cap q = \emptyset$ can be reported efficiently.

4.1 Static Solution

We need to find out pairwise intersections of the segments in S which overlap with the query interval q. The above problem is characterized in the following lemma [8].

Lemma 1. *A pair of segments $(s,\, t)$ of S satisfies $s \cap t \cap q \neq \emptyset$ iff*
(i) An endpoint of s is in $t \cap q$ or
(ii) An endpoint of t is in $s \cap q$ or
(iii) $q \subseteq (s \cap t)$

In [8], to report pairs satisfying conditions (i) and (ii) of Lemma 1, they use a data structure that takes up $O(n \log n)$ space and $O(\log n + k)$ time. We shall reduce it to $O(n)$ by the following steps: Discard all the segments of S which do not intersect with any other segment of S. Call the reduced set S'. However, we can simply preprocess the segments of S' and the endpoints of the segments of S' into an instance of the data structure of Theorem 1. For a query interval q, we query the data structure and for each reported tuple $(p,\, i)$, we report $(p',\, i)$ where p' is the segment to which p is an endpoint. This will reduce the space requirement to $O(n)$.

To report segment pairs satisfying condition (iii) of Lemma 1, in [8], they map each segment $s[c, d]$ of S' into the point $(c,\, d) \in \mathbb{R}^2$. These points are preprocessed into a data structure D for 2-d quadrant searching. This can be implemented using a priority search tree [9]. The query interval $q = [a_1,\, b_1]$ is mapped into the northwest quadrant $NW(q)$ of the point $(a_1,\, b_1) \in \mathbb{R}^2$. D is queried with $NW(q)$ and the result is stored in a temporary list $L(q)$. For each pair of points $(p_1,\, p_2)$, $p_1 \in L(q)$, $p_2 \in L(q)$, the interval pair $(p'_1,\, p'_2)$ is reported where p'_1 (respectively p'_2) is the segment corresponding to p_1 (respectively p_2).

Theorem 5. *A set S of n segments on the x-axis can be preprocessed into a data structure of size $O(n)$ such that given a query interval $q = [a,\, b]$, all pairs $(s,\, t)$, $s \in S$, $t \in S$ such that $s \cap t \cap q \neq \emptyset$ can be reported in time $O(\log n + k)$ where k is the output size.*

4.2 Semi-dynamic solution

The solution to the semi-dynamic version of the problem is similar to that of the static solution. We shall preprocess the segments of S and the endpoints of the segments of S into an instance of the data structure of Theorem 3. However, one important consideration is taken into account while preparing lists L_p for $p \in BT_2$. An endpoint of a segment t is not considered to be intersecting with the segment it comes from (which in this case is t). This observation has to be incorporated into the data structure of Theorem 3. This makes sense since in conditions (i) and (ii) of Lemma 1 we want an endpoint of a segment t to intersect with a segment other than t. For handling condition (iii), instead of using a static priority search tree (used in the static solution), we shall use a Dynamic priority search tree D [5]. A Dynamic priority search tree when built on m points takes up $O(m)$ space, answers queries in $O(\log m + k)$ time and updates take $O(\log m)$ time. In our case $m \equiv O(n)$. Given a query interval q, the same procedure as followed for the static solution is repeated.

Theorem 6. *A set S of n segments on the x-axis can be preprocessed into a data structure of size $O(n \log n)$ such that given a query interval $q = [a_1,\, b_1]$,* *all pairs (s, t), $s \in S$, $t \in S$ such that $s \cap t \cap q \neq \emptyset$ can be reported in time $O(\log n + k)$ where k is the output size. Also, insertion of a new segment can be done in $O(\log n)$ amortized time.*

5 Range-Aggregate Orthogonal Segment Intersection

Problem 3. Given a set H of horizontal segments and a set V of vertical segments ($|H| + |V| = n$), preprocess them into a data structure such that given query rectangle q=$[a_1, b_1] \times [a_2, b_2]$, we can efficiently report all the pairs of horizontal-vertical segments (h, v) such that $h \in H, v \in V$, and $h \cap v \cap q \neq \emptyset$.

The vertical (resp. horizontal) segments V (resp. H) intersecting q can be categorized into three categories:(a) segments whose both the endpoints are inside q, (b) segments whose one endpoint is inside q, (c) segments which cross q completely. In Figure 1 we show an example of segments in H and V classified into these three categories.

First, we shall build a data structure to report all the intersections involving *vertical* segments of type (a) and (b), for a given query q. Then another data structure is built to help in reporting all the intersections involving *horizontal* segments of type (a) and (b). The data structures for these are described next.

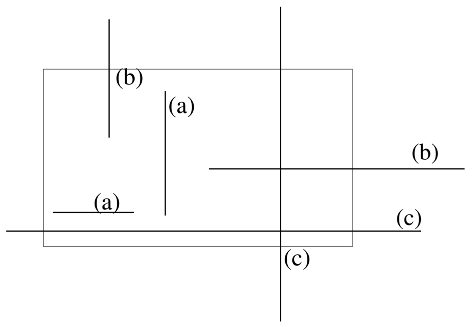

Fig. 1. Different types of vertical and horizontal segments that can intersect q.

We assume that the endpoints of no two horizontal (resp. vertical) segments have the same x and y coordinate. In the preprocessing phase, create a bounding box $\mathcal{B}$ for the segments in the set $H \cup V$. Decompose the bounding box $\mathcal{B}$ into vertical slabs by shooting vertical rays upward and downward from the endpoints of the all the horizontal segments in set H till they hit the walls of $\mathcal{B}$. This divides the plane into many vertical slabs. For each vertical slab $\mathcal{S}_i$, we create a list L_i which stores the y-projection of all the horizontal segments of H passing through the slab $\mathcal{S}_i$ (similar to the technique used earlier in the paper). The list L_i stores the y-projections of the horizontal segments in a sorted order. Note that the list L_i is almost similar to the list L_{i+1} except there is an inclusion or deletion of a value from the list L_i. Hence these lists L_i can be implemented using persistence as done previously.

A vertical segment $v(v_x; v_l, v_u) \in V$ represents a segment with v_x as its vertical projection and $[v_l, v_u]$ as its y-projection. For each vertical segment $v \in V$, we first find the slab $\mathcal{S}_i$ where the x-projection of v (i.e., v_x) lies . Then in the slab $\mathcal{S}_i$, we consider the y-projection of v, i.e., $[v_l, v_u]$ and find out the smallest element h_l and the largest element h_u in the list L_i such that $v_l \leq h_l \leq h_u \leq v_u$. Create two 2-d points (v_x, h_l) and (v_x, h_u). Store these points in a 2-d range searching data structure RT_v. With each point p in RT_v we shall maintain appropriate pointer to the element in L_i from which p got generated. The space occupied by RT_v will be $O(n \log n)$. These structures will help us in reporting intersections involving vertical segments of type (a) and (b).

Analogously, in order to report horizontal segments of type (a) and (b), we shall build similar data structures as done to handle vertical segments. This will lead to creating lists L_i based on the vertical segments and an analogous tree RT_h.

Now we shall build data structures to handle intersections where both the vertical and horizontal segments are of type (c). A data structure T_v (resp. T_h) shall be built, which for a given query rectangle q shall report all the vertical (resp. horizontal) segments completely crossing q. Convert each vertical segment $v \in V$ into a 3-d point $v'(v_x, v_l, v_u)$. Create a binary search tree T_v whose leaf nodes are sorted in terms of the v_x-values of these 3-d points. At each internal node $\mu \in T_v$, collect all the points of the subtree rooted at μ. Let $X(\mu)$ denote the average of the v_x-value in the rightmost leaf in μ's left subtree and the v_x-value in the leftmost leaf in μ's right subtree. Let μ be a left child of its parent. At the node μ, create a data structure D_μ which is an instance of [1] for handling 3D-dominance reporting queries and handles queries of the form $q' = [a_1, \infty) \times (-\infty, a_2] \times [b_2, \infty)$. If μ is a right child, D_μ will be a 3D-dominance reporting data structure to report points in $q'' = (-\infty, b_1] \times (-\infty, a_2] \times [b_2, \infty)$. The data structure D_μ takes linear storage space and answers queries in $O(\log n$ $+ k)$ query time [1]. Hence, the space occupied by T_v will be $O(n \log n)$. Similarly, based on horizontal segments (H) we build tree T_h but now the primary structure is based on the y-projection of H.

Given a query rectangle $q = [a_1, b_1] \times [a_2, b_2]$, search RT_v and for each point (v_x, h_l) reported, jump to the slab $\mathcal{S}_i$ where the coordinate v_x lies. Here v_x is the x projection of a vertical segment v. Next, starting from the index of the value h_l in the list L_i we descend (ascend) the list L_i until we find a value $h'_y \in L_i$ such that (1) $h'_y < a_2$ (resp $h'_y > b_2$) or (2) $h'_y < v_l$ (resp $h'_y > v_u$). Here $[v_l, v_u]$ is the y -projection of the vertical segment v. This will report all the intersections involving vertical segments of type (a) and (b). Similarly, query RT_h to report intersections involving horizontal segments of type (a) and (b). This will take $O(\log n + k_1)$ time, where k_1 is the number of reported pairs involving segments of type (a) and (b).

Now, all that is left is to report all intersections (h, v) in which both $h \in H$ and $v \in V$ completely cross q. We search T_v with $[a_1, b_1]$ to find the highest node $\pi \in T_v$ such that $X(\pi)$ lies in the interval $[a_1, b_1]$. Let l and r be the left and the right child of π, respectively. Search π_l with q_l=$[a_1, \infty) \times (-\infty, a_2] \times [b_2, \infty]$ and π_r with q_r=$(-\infty, b_1] \times (-\infty, a_2] \times [b_2, \infty)$. If π_l or π_r, report at least one point, then quit the query procedure on T_v. Similarly, query T_h and quit the query procedure if it reports at least one point. If both T_v and T_h report at least one point on being queried with q, then once again query both the structures with q

and this time all the points satisfying the query q are reported. Let H_c^q and V_c^q be the set of segments be reported by querying T_h and T_v, respectively. Then we shall report all the pairs $(h,\ v)$, $\forall\ h \in H_c^q$ and $v \in V_c^q$. This procedure will take $O(\log n\ +\ k_2)$, where k_2 is the number of pairs involving type (c) segments. The overall query time is $O(\log n\ +\ k)$.

Theorem 7. *A set H of horizontal segments and a set V of vertical segments ($|H|\ +\ |V| = n$) in $\mathbb{R}^2$ can be preprocessed into a data structure of size $O(n \log n)$ such that given a query rectangle $q\ =\ [a_1,\ b]\ \times\ [a_2,\ b_2]$, all pairs of horizontal-vertical segments $(h,\ v)$, $h \in H$, $v \in V$ and $h\ \cap\ v\ \cap\ q\ \neq \emptyset$ can be reported in time $O(\log n\ +\ k)$, where k is the output size.*

6 Range-Aggregate Orthogonal Segment Intersection with Distant Constraint

In this section, we target the following problem that has been studied in [6]. The problem is important for finding minimum extension violations in a VLSI circuit. Definitions and more elaborate explanation on the same can be found in [6]. Here $dist(p_1, p_2)$ refers to the euclidean distance between points p_1 and p_2.

Problem 4. Let H be a set of horizontal line segments and V be a set of vertical line segments ($|H|\ +\ |V|\ =\ n$). We need to preprocess them into a data structure such that given a query rectangle q and a parameter δ, all the triplets (h, v, p) where $h \in H$, $v \in V, p$ is an endpoint of either h or v such that $h \cap v \cap q \neq \emptyset$ and $dist(h \cap v, p) \leq \delta$, can be reported efficiently.

In this section we study the preliminary version of the problem 4 which we define next.

Problem 5. Given a set S of points and a set T of line segments ($|S|\ +\ |T|$ $=\ n$), on the real line $\mathbb{R}^1$, preprocess them into a data structure such that given a query interval $[a_1, b_1]$ and a parameter δ, we can efficiently report all the pairs of (s, t) where $s \in S$, $t \in T$ such that $p \cap t \cap q \neq \emptyset$ and $dist(s, e) \leq \delta$, where e is one of the endpoint of the segment t.

Problem 5 was solved in [6] in $O(\log n + k)$ time using a storage space of $O(n \log n)$. This solution was used to solve problem 4 in $O(\log^2 n + k)$ time using a storage space of $O(n \log^3 n)$ in [6]. We will now show that problem 5 can be solved in $O(\log n + k)$ time using a storage space $O(n)$ which results in problem 4 being solved in $O(\log^2 n + k)$ time using a storage space of $O(n \log^2 n)$.

Problem 5 can be solved in the following way. Partition the real line into $2|T| + 1$ *elementary intervals* by drawing vertical line from the end points of the line segments of T. Discard points of S which do not stab any segment of T resulting in set S'. For every elementary interval I_j we maintain two arrays A_j and B_j. The array A_j (resp B_j) stores the left endpoints (resp. right endpoints) of the segments passing through elementary interval I_j in decreasing (resp. increasing) order of their x-coordinates. Since consecutive arrays A_j's are almost same they can be implemented using $O(n)$ storage space by using persistence [7]. For a point $p \in S'$ find out the elementary interval in which it lies. Let the point

p lie in the elementary interval I_j. Find the distance d' between the point p and the first point of the array A_j. Similarly find the distance d'' between the point p and the first point of the array B_j. Create two 2-d points (p, d') and (p, d''). Repeat the process for all the points in the set S'. Create a priority search tree [9], T_{PST} on the 2-d points thus created.

Given a query interval $[a_1,\ b_1]$ and the parameter δ, we search T_{PST} with $[a_1, b_1] \times (-\infty, \delta]$ and for each point reported we jump to the corresponding elementary interval I_j and traverse the array A_j (or B_j) and report the pairs (s, t) as long as $dist(s, e) \leq \delta$ where e is an end point of t or till the end of the array is reached.

Theorem 8. *There exists a data structure which solves problem 5 using linear space and $O(\log n + k)$ query time.*

Using the result of the Theorem 8 and the techniques of [6], problem 4 solution can be improved to $O(\log^2 n + k)$ query time and a storage space of $O(n \log^2 n)$.

Theorem 9. *There exists a data structure which solves problem 4 using $O(n \log^2 n)$ space and $O(\log^2 n + k)$ query time.*

References

1. P. Afshani. *On Dominance Reporting in 3D* Proceedings of 16thEuropean Symposium on Algorithms (ESA), 2008, 41–51.
2. Pankaj K. Agarwal, Lars Arge, Jun Yang, Ke Yi. I/O-Efficient Structures for Orthogonal Range-Max and Stabbing-Max Queries. *11th European Symposium on Algorithms*, 7–18, 2003.
3. M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Computational Geometry, Springer Verlag, 2nd ed., 2000.
4. Y. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry", Proceedings of the IEEE, Special Issue on Computational Geometry, G. Toussaint (Ed.), vol. 80(9), pp. 1412-1434, 1992.
5. Alok N. Choudhary. Dynamic Priority Search Trees, November 1987. http://www.cs.brown.edu/courses/cs252/misc/proj/src/Spr96-97/mjr/doc/dyn-pst.ps
6. A. S. Das, P. Gupta, K. Srinathan. Data Structures for Reporting Extension Violations in a Query Range. *Proceedings, 21st Canadian Conference on Computational Geometry (CCCG)*, 2009, 129–132
7. J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
8. P. Gupta. Algorithms for Range-Aggregate Query Problems Involving Geometric Aggregation Operations. *ISAAC*, 2005: 892-901
9. E.M. McCreight. Priority search trees, *SIAM Journal of Computing*, 14(2), 257–276, 1985.
10. K. Mehlhorn and S. Naher, "Dynamic Fractional Cascading", Algorithmica 5(1990), 215-241.
11. S. Shekhar, and S. Chawla. Spatial Databases: A Tour, Prentice Hall, 2002.
12. Y. Tao and D. Papadias. Range aggregate processing in spatial databases. IEEE Transactions on Knowledge and Data Engineering, 16(12), 2004, 15551570.