

# Efficient top- $k$ queries for orthogonal ranges

Saladi Rahul<sup>1</sup>, Prosenjit Gupta<sup>2</sup>, Ravi Janardan<sup>3</sup>, and K. S. Rajan<sup>1</sup>

<sup>1</sup> Lab for Spatial Informatics, IIT-Hyderabad

srahul@research.iiit.ac.in, rajan@iiit.ac.in

<sup>2</sup> Yahoo! Research and Development, Bangalore 560093, India

prosenjit\_gupta@acm.org

<sup>3</sup> Dept. of Computer Science & Engineering, University of Minnesota

janardan@cs.umn.edu

**Abstract.** Advances in sensing and data gathering technologies have resulted in an explosion in the volume of data that is being generated, processed, and archived. In particular, this information overload calls for new methods for querying large spatial datasets, since users are often not interested in merely retrieving a list of all data items satisfying a query, but would, instead, like a more informative “summary” of the retrieved items. An example is the so-called top- $k$  problem, where the goal is to retrieve from a set of  $n$  weighted points in  $\mathbb{R}^d$  the  $k$  most significant points, ranked by their weights, that lie in an orthogonal query box in  $\mathbb{R}^d$  (rather than get a list of all points lying in the query box). In this paper, efficient and output-sensitive solutions are presented for this problem in two settings. In the first setting, the  $k$  points are reported in arbitrary order and the underlying set can be updated dynamically through insertions and deletions of points. In the second setting, the  $k$  points are reported in sorted order of their weights.

## 1 Introduction

In recent years, advances in computing, communication, and sensing technologies have led to an explosion in the quantity of data that is generated on a daily basis. (For instance, according to a recent study (<http://tinyurl.com/2frbdeu>), the total volume of electronic data stored worldwide reached 1.2 billion terabytes (a growth rate of 62% from 2009) and is projected to reach 35 billion terabytes by the year 2020.) This information overload calls for new methods to query large datasets since users are often interested not in merely retrieving a list of all the data items that satisfy a query but would instead like a more informative and manageable “summary” of the query results. Generating such a summary requires applying some kind of aggregation function to the query results and doing so efficiently. One of the simplest and most useful aggregation functions is reporting the  $k$  best items among the ones that satisfy the query, where the notion of “best” is based on a weight associated with each data item. This has led to the formulation of the so-called *top- $k$  problem*, where the goal is to retrieve from a set,  $S$ , of  $n$  weighted points in  $\mathbb{R}^d$  the  $k$  most significant points, ranked by their weights, that lie in a user-specified query range  $q$ , for

instance an orthogonal box in  $\mathbb{R}^d$ . The top- $k$  problem has been studied intensively in the context of information retrieval [5], multimedia similarity search [7], text and data integration [14], business analytics [3], preference queries over product catalogs and Internet-based recommendation sources [15], distributed aggregation of network logs and sensor data [6] and so on. In these domains, the end-users are most interested in the important (i.e. top- $k$ ) query answers in the potentially huge answer space. The top- $k$  problem is a non-trivial generalization of the well-studied range search problem in computational geometry, where the goal is to preprocess  $S$  so that the points that lie in  $q$  can be reported or counted efficiently. (See [2] for a survey of geometric range search.)

In this paper, we present efficient geometric algorithms and data structures for two versions of the top- $k$  problem. In the first version, the top  $k$  points can be reported in any order and the underlying set  $S$  can be updated through insertion and deletion of points. In the second case, the top  $k$  points are required to be reported in sorted order by weight. Besides being efficient in the input size,  $n$ , our solutions are also efficient in terms of the output size,  $k$ , i.e., the query time is a function of  $k$  rather than the actual number of items satisfying the query, which can be much larger. In other words, our solutions are *output-sensitive*.

Before proceeding to the formal problem statement and solution technique, we begin with two motivating examples for the top- $k$  problem and also discuss briefly prior related work.

- Modern GIS systems allow users to query the underlying spatial dataset for useful information. For instance, consider a GIS for rural India which contains location and population information for the villages in a large state. For census or resource allocation purposes, a demographer might wish to identify the  $k$  most populated villages in a rectangular query region of interest. If we model each village as a point in the  $(x, y)$ -plane and associate with it an integer weight equal to its population, then the problem can be framed as an orthogonal top- $k$  range query.
- Consider a financial database of stocks. Each stock is represented by a point in  $\mathbb{R}^d$ , where each of the  $d$  dimensions encodes an attribute of the stock (e.g., price, earnings, dividends, trading volume, market capitalization, etc.) Also associated with the stock is a weight equal to (say) its 5-year performance. An investor might be interested in identifying the ten best-performing stocks over the past five years that also satisfy requirements specified as a range of price, earnings, dividends, etc. This can be accomplished by performing an orthogonal range query to determine all stocks satisfying the  $d$  ranges specified by the query and then sifting through this potentially large output for the ten best stocks. However, a more informative and efficient approach is to issue a top- $k$  ( $k = 10$ ) query on the database to get exactly the desired stocks.

The top- $k$  version of range searching falls under a class of problems called *range-aggregate* query problems [19] in which many composite queries involving range searching are considered, wherein one needs to compute the aggregate

function of the objects in  $S \cap q$  rather than report all of them as in a range reporting query. In many applications like on-line analytical processing (OLAP), geographic information systems (GIS) and information retrieval (IR), aggregation plays an important role in summarizing information [19] and hence a large number of algorithms and storage schemes have been proposed to support such queries.

### 1.1 Related Work

The following problem (top-1) was considered in Chazelle [8]: Preprocess a set  $S$  of weighted points in  $\mathbb{R}^d$  so that given a query orthogonal box  $q$ , report the point of  $S$  in  $q$  with the maximum weight. On a pointer machine model he obtained a linear space solution which answered queries in  $O(\log^3 n)$  time, for  $d=2$ . In the dynamic setting, the query time increased to  $O(\log^3 n \log \log n)$  and the update time was  $O(\log^3 n \log \log n)$ . In Agarwal et al. [1], this problem is solved on an external memory model. For  $d=2$ , they obtain a static solution which takes up linear space and answers queries in  $O(\log_B^2 n)$  time, where  $n = \lceil N/B \rceil$ . In the dynamic setting the query time increases to  $O(\log_B^3 n)$  and the update time is  $O(\log_B^2 n)$ . The problem being attempted in this paper is a generalized version of this problem wherein we want to report the  $k$  most weighted points. In Gabow et al. [11] they solve the range minimum query by transforming it into a lowest common ancestor problem using the technique of Cartesian trees.

Recently Brodal et al. [4] studied the following problem: Preprocess a one-dimensional array  $A$  having  $n$  elements, so that given two indices  $i$  and  $j$  and an integer  $k$ , report the  $k$  smallest elements in the subarray  $A[i \dots j]$  in sorted order. They assume a RAM model and obtain linear space solution and answer queries in  $O(k)$  time. In the field of databases, top- $k$  query processing is an active area of research. [13] is a good survey paper on the recent results of top- $k$  query processing techniques in relational database systems.

### 1.2 Formal Problem Statement

Formally, the problem we consider is the following.

- Let  $S$  be a set of  $n$  weighted points in  $\mathbb{R}^d$ , where each point  $p$  has a real-valued weight  $w(p)$ . We would like to preprocess  $S$  into a suitable data structure so that for any given orthogonal query box  $q = \prod_{i=1}^d [a_i, b_i]$  and an integer  $k \in [1, n]$ , the top- $k$  points in  $S \cap q$ , ranked by their weights can be reported efficiently and output-sensitively.

We note that the top- $k$  problem is decomposable: Given two disjoint point sets  $A$  and  $B$ , the solution to the top- $k$  problem on  $A \cup B$  can be obtained from the solutions to the top- $k$  problems on  $A$  and  $B$  respectively in  $O(k)$  time by using the linear-time selection algorithm [9].

A straightforward approach to the top- $k$  problem uses a traditional range search method to retrieve the points in  $S \cap q$  and then applies a linear-time

selection algorithm to  $S \cap q$  to identify the  $k$ th largest element. A subsequent linear scan of  $S \cap q$  then identifies the desired top- $k$  elements. However, this approach is expensive when  $|S \cap q| \gg k$ , i.e., it is not sensitive to the output size  $k$ . We seek an efficient, output-sensitive solution to the problem. In the following sections, we describe such solutions for this problem in two settings. In the first one (Section 2) the top- $k$  points may be output in arbitrary order and another (Section 3) where they are requested in sorted order by weight. (The first solution also supports updates in the underlying set  $S$ ). The results obtained have been summarized in Table 1.

Reporting points in	Underlying Space	Space occupied	Query time	Update time (amortized)
arbitrary order	$\mathbb{R}^d$ ( $d \geq 1$ )	$O(n \log^d n)$	$O(\log^d n \log \log n + k)$	$O(\log^d n \log \log n)$
		$O(n \log^d n)$	$O(\log^d n + k)$	————
sorted order	$\mathbb{R}^1$	$O(n \log n)$	$O(\log n + k)$	————
	$\mathbb{R}^d$ ( $d \geq 2$ )	$O(n \log^d n)$	$O(\log^{d-1} n + k \log \log n)$	————

**Table 1.** Summary of results obtained for top- $k$  queries for orthogonal ranges.  $k$  is also part of the query. Two settings are considered. The first one reports the top- $k$  points in arbitrary order and the other in sorted order by weight. All bounds are worst-case results.

## 2 Dynamic structure for reporting top-k points in arbitrary order

In this section we report the top- $k$  points in an arbitrary order for a set of  $n$  weighted points in  $\mathbb{R}^d$ . In the preprocessing phase, all the points of  $S$  are sorted based on their weights in non-increasing order. Let the sequence of points obtained be  $w(p_1), w(p_2), \dots, w(p_n)$ . These points are made the leaves of a  $BB(\alpha)$  tree  $\mathcal{T}$  [17] from left to right. At each internal node  $v$  of  $\mathcal{T}$ , we build a data structure  $\mathcal{M}_C^v$  on the set of points stored in the subtree rooted at  $v$ . This data structure supports  $d$ -dimensional orthogonal range counting queries and allows updates (insertion/deletion of points). A dynamic  $d$ -dimensional range tree can be used for supporting these operations. It uses  $O(n \log^{d-1} n)$  space and handles updates in  $O(\log^{d-1} n \log \log n)$  time. It can count points lying inside an orthogonal query box in  $O(\log^{d-1} n \log \log n)$  time [16].

Next we transform each point  $p(p_1, p_2, \dots, p_d) \in S$  having weight  $w(p)$  into a  $(d+1)$ -dimensional point  $p'(p_1, p_2, \dots, p_d, w(p))$ . Call this new set of transformed points  $S'$ . We build a data structure  $\mathcal{M}_R$  on  $S'$ , so that given a query of the form  $q = [a_1, b_1] \times \dots \times [a_d, b_d] \times [w_q, \infty)$ , all the points in  $S'$  lying within  $q$  can get

reported efficiently. This structure also supports updates (insertion/deletion of points). A dynamic  $(d+1)$ -dimensional range tree [16] can be used where the first  $d$  level trees are built using the  $d$  coordinate values of each point and dynamic fractional cascading is applied at the innermost level involving the weights of the points.

In the query, we are given an orthogonal  $d$ -dimensional box and an integer  $k$ . The query algorithm consists of the following steps : (a) Using  $\mathcal{T}$  determine the number of points of  $S$  lying within  $q$ . If  $|S \cap q| \leq k$ , then report all the points in  $S \cap q$ ; (b) If  $|S \cap q| > k$ , search for a *threshold* point  $p \in S$  in  $\mathcal{T}$ , such that, the number of points of  $S'$  in the region  $q' = \prod_{i=1}^d [a_i, b_i] \times [w(p), \infty)$  is exactly  $k$ ; (c) Using  $\mathcal{M}_R$ , report all the  $k$  points lying in the region  $q'$ . Steps (a) and (c) are fairly straightforward. The challenge is to find the *threshold* point in step (b).

---

**Algorithm 1** Query (Node  $v$ , Integer  $k'$ , Query Box  $q$ )

---

```

1:
2: int count= $\mathcal{M}_C^v(q)$ ;
3:
4: if count <  $k'$  then
5:   if  $v$  is the root of  $T$  then
6:      $\mathcal{M}_R(q \times (-\infty, \infty))$ ;
7:   else
8:     Query( $sib(v)$ ,  $k' - \text{count}$ ,  $q$ ); { $sib(v)$  is the sibling of the node  $v$ }
9:   end if
10:
11: else if count >  $k'$  then
12:   Query( $lc(v)$ ,  $k'$ ,  $q$ ); { $lc(v)$  is the left child of the node  $v$ }
13:
14: else if count ==  $k'$  then
15:    $\mathcal{M}_R(q \times [w(p_r), \infty))$ ; { $p_r$  is the rightmost point in the subtree rooted at  $v$ }
16:   STOP the execution of the query algorithm.
17:
18: end if

```

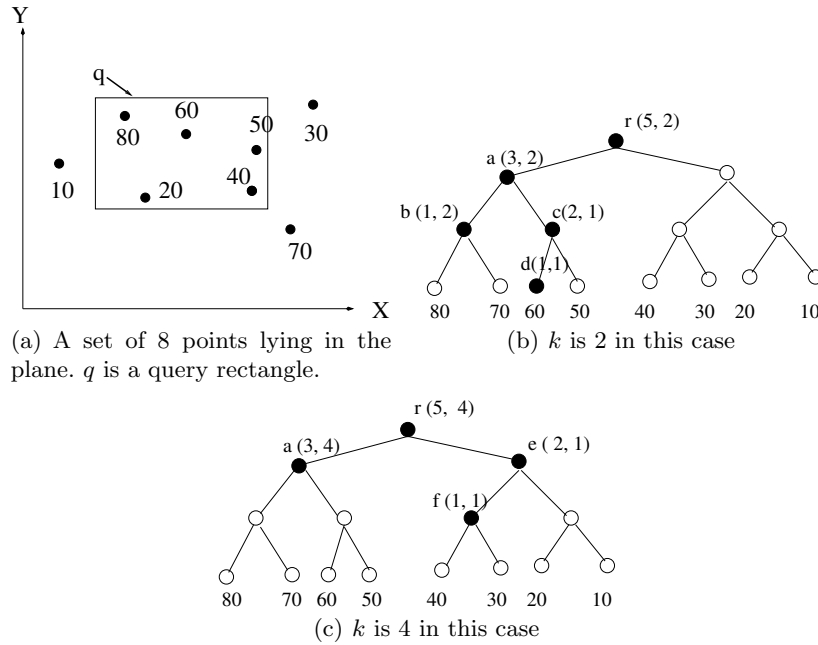
---

The query algorithm is explained in the form of a pseudo-code in Algorithm 1. The query algorithm takes as input a node  $v \in \mathcal{T}$ , integer  $k'$ , and the given query box  $q$ . (Initially,  $v$  is the root  $r$  of  $\mathcal{T}$  and  $k'=k$ ). At the root node, we first query  $\mathcal{M}_C^r$  with  $q$  (line 2) to find out the number of points of  $S$  lying within  $q$ . If  $|S \cap q| \leq k$ , then all the points lying within  $q$  are reported. Line 4–6 and 14–16 handle this case (step a).

If  $|S \cap q| > k$ , then the objective of the algorithm is to find the *threshold* point. We go to the left child (i.e.  $lc(r)$ ) of  $r$  in search of our threshold point (lines 11–12). Let  $S_v$  denote the set of points lying in the subtree rooted at an internal node  $v$ . At node  $lc(r)$ , if  $\text{count} > k$ , i.e.,  $|S_{lc(r)} \cap q| > k$ , which means that the threshold point lies within the subtree of  $lc(r)$ . So, in our search we

will next visit the left child of  $lc(r)$  and query it as  $Query(lc(lc(r)), k, q)$  (again lines 11–12). However, if at node  $lc(r)$ ,  $count < k$ , it means that the threshold point lies in the subtree rooted at the sibling of  $lc(r)$ , i.e.,  $sib(lc(r))$ . Then we query the sibling node of  $lc(r)$  as  $Query(sib(lc(r)), k - count, q)$  (line 8). Note carefully that the second argument is  $k - count$ .

Similarly, at a particular internal node  $v \in T$ , if the threshold point is found to lie inside the subtree of  $v$ , then we descend to the left child of  $v$ ; else we move to the sibling of node  $v$ . Finally at a node  $v'$ , the else condition in line 14 gets satisfied.  $p_r$  is defined as the rightmost point in the subtree rooted at  $v'$ . This assures that  $k = |W' \cap q|$  where  $W' = p_1, p_2, \dots, p_r$ . Now we bring in our range reporting data structure  $\mathcal{M}_R$  and query it with  $\prod_{i=1}^d [a_i, b_i] \times [w(p_r), \infty)$ . The top- $k$  points of  $S$  lying inside  $q$  get reported.



**Fig. 1.** Examples for finding the *threshold* point.

The query algorithm is illustrated via an example shown in Figure 1. In 1(a), eight points lying in a two-dimensional plane are shown along with their weights. The query rectangle  $q$  is also shown in it. Points having weight 20, 40, 50, 60, 80 lie inside  $q$ . Figure 1(b) and 1(c) depict the flow of the query algorithm within the data structure  $\mathcal{T}$  for  $k=2$  and  $k=4$ , respectively. The nodes shown in black are the nodes visited during the execution of the query algorithm. With each black node  $v$  we have associated a label of the form  $v(count, k')$ , where  $count$

denotes the number of points in the subtree of node  $v$  which lie inside  $q$  and  $k'$  is the parameter passed while visiting node  $v$ . The value of *count* is determined by the orthogonal range counting structure stored at  $v$ . For  $k=2$ , we need to report points with weight 80 and 60. In Figure 1(b), at the root we have the label  $r(5, 2)$  which implies that a threshold point exists. Then we visit its left child  $a$ . Since it has the label  $a(3, 2)$ , it means that the threshold point lies in the subtree of  $a$  and hence we visit node  $b$ . However, only one point in the subtree of  $b$  lies inside  $q$  (point with weight 80) which means that the threshold point will lie in its sibling node (i.e. node  $c$ ). From  $c$  we go to its left child  $d$  where both the values match. Then we query our reporting structure with  $q \times [60, \infty)$  to report points with weight 80 and 60.

For  $k=4$  (Figure 1(c)), we need to report points with weights 80, 60, 50 and 40. The subtree under  $a$  contains three of the required points (namely 80, 60 and 50). Now we are in search of one more point under the subtree of  $e$ . At node  $f$  we notice that both the values match ( $f(1, 1)$  as point with weight 40 lies inside  $q$ ). Again we query our reporting structure with  $q \times [30, \infty)$ , since 30 is the rightmost point in the subtree of  $f$ . The appropriate points get reported. An important lemma is stated next.

**Lemma 1.** *For a given query  $q$  and an integer  $k$ , the number of nodes visited by the query algorithm in the primary structure of  $\mathcal{T}$  is  $O(\log n)$ .*

*Proof.* The height of a  $BB(\alpha)$  tree built on  $n$  points is  $O(\log n)$ . Assume that we are at an internal node  $v \in \mathcal{T}$ . If  $v$  is a left child of its parent node (i.e.  $p(v)$ ), then if there is a next step, we might have to go either to its right child (i.e.  $sib(v)$ ) or to its left child (i.e.  $lc(v)$ ). However, if  $v$  is a right child of its parent node (i.e.  $p(v)$ ), then if there is a next step then we will *always* go its left child (i.e.  $lc(v)$ ). The reason is the following: We reached node  $v$  because at node  $p(v)$  it must have been observed that the threshold point lies in the subtree rooted at  $p(v)$  but not in the subtree rooted at node  $lc(p(v))$  (i.e. left sibling of  $v$ ). Therefore, the threshold point has to lie inside the subtree rooted at node  $v$  and hence we visit  $lc(v)$ . So, after every two steps we go down at least one level in  $\mathcal{T}$ . Hence, the number of nodes visited in the primary structure of  $\mathcal{T}$  is bounded by  $O(\log n)$ .  $\square$

## 2.1 Handling insertions and deletions

As noted earlier, the  $d$ -dimensional range reporting/counting structures can handle insertion and deletion of points of  $S$ . Suppose that we want to insert/delete a point in  $S$ . We first insert/delete  $p$  in  $\mathcal{M}_R$ . Then we search down  $\mathcal{T}$  and insert/delete a leaf and then update the secondary structure (i.e.  $\mathcal{M}_C^v$ ) at each node on the search path. Then  $\mathcal{T}$  might have to be rebalanced via rotations. The rotations will change the set of descendant leaves of certain nodes and thus render obsolete their secondary structures ( $\mathcal{M}_C^v$ ). Then the secondary structures will have to be rebuilt. As shown in Willard et al. [18], the amortized update time for  $\mathcal{T}$  will be  $O(U(n) \log n)$ , where  $U(m)$  is the amortized update time for

$\mathcal{M}_C^c$  built on  $m$  points. The overall update time will be  $O(U(n) \log n + U'(n))$ , where  $U'(n)$  is the amortized update time for  $\mathcal{M}_R$ .

**Theorem 1.** *Suppose that we have a data structure  $\mathcal{M}_C$  which supports  $d$ -dimensional range counting queries. Let the performance of  $\mathcal{M}_C$  be represented by the tuple  $\langle S_c(n), Q_c(n), I_c(n), D_c(n) \rangle$ , where  $S_c(n)$  is the space occupied,  $Q_c(n)$  is the time taken to answer a counting query,  $I_c(n)$  is the amortized time taken to insert a new point and  $D_c(n)$  is the amortized time taken to delete a point. Similarly, suppose that we have a data structure  $\mathcal{M}_R$  which supports orthogonal range reporting for queries of the form  $\prod_{i=1}^d [a_i, b_i] \times [a_{d+1}, \infty)$ . Let its performance be represented by the tuple  $\langle S_r(n), Q_r(n), I_r(n), D_r(n) \rangle$ .*

*Then, a set  $S$  of  $n$  weighted points in  $\mathbb{R}^d$  can be preprocessed into a data structure, so that given an orthogonal  $d$ -dimensional box and an integer  $k \in [1, n]$ , the top- $k$  points in  $S \cap q$ , ranked by their weights can be reported efficiently, with the following bounds:*

- *The space occupied by the data structure is  $O(S_c(n) \log n + S_r(n))$ .*
- *The time taken to answer a query is  $O(Q_c(n) \log n + Q_r(n) + k)$ .*
- *The amortized time taken to insert a new point is  $O(I_c(n) \log n + I_r(n))$ .*
- *The amortized time taken to delete a new point is  $O(D_c(n) \log n + D_r(n))$ .*

In a dynamic setting, a dynamic range tree when built on  $n$   $d$ -dimensional points uses  $O(n \log^{d-1} n)$  space, handles updates in  $O(\log^{d-1} n \log \log n)$  time. It can report and count points lying inside an orthogonal query box in  $O(\log^{d-1} n \log \log n + k)$  and  $O(\log^{d-1} n \log \log n)$  time, respectively [16]. Substituting these values in Theorem 1, we obtain the following theorem.

**Theorem 2.** *A set  $S$  of  $n$  weighted points in  $\mathbb{R}^d$  ( $d \geq 1$ ) can be preprocessed into a data structure of size  $O(n \log^d n)$  so that given a query orthogonal box  $q$  and an integer  $k \in [1, n]$ , the top- $k$  points in  $q$  ranked by their weights can be reported in  $O(\log^d n \log \log n + k)$  time. Additionally points can be inserted and deleted in  $O(\log^d n \log \log n)$  amortized time. The model of computation is assumed to be a pointer machine model.*

If we do not want any updates to happen on our underlying set  $S$ , then we can seek to improve the query time. The dynamic range tree is replaced by a static range tree. A static range tree when built on  $n$   $d$ -dimensional points uses  $O(n \log^{d-1} n)$  space. It can report and count points lying inside the query box in  $O(\log^{d-1} n + k)$  and  $O(\log^{d-1} n)$  time, respectively [2]. Substituting these values in Theorem 1, we obtain the following theorem.

**Theorem 3.** *A set  $S$  of  $n$  weighted points in  $\mathbb{R}^d$  ( $d \geq 1$ ) can be preprocessed into a data structure of size  $O(n \log^d n)$  so that given a query orthogonal box  $q$  and an integer  $k \in [1, n]$ , the top- $k$  points in  $q$  ranked by their weights can be reported in  $O(\log^d n + k)$  amortized time. The model of computation is assumed to be a pointer machine model.*

### 3 Reporting top- $k$ points in sorted order

In this section we propose a static solution that reports the top- $k$  points in sorted order of their weights. First we present a solution for this problem in one-dimensional space. Then we present a separate solution for  $d$ -dimensional space, where  $d \geq 2$ .

#### 3.1 Solution for $\mathbb{R}^1$

First we consider the case wherein the queries are of the form  $q = [a_1, \infty)$ . For a fixed  $a_1$ , we can store all the points of  $S$  that lie in the interval  $[a_1, \infty)$  in a list  $\mathcal{L}(a_1)$  in the non-increasing order of weights. Given  $k$ , we can retrieve the top- $k$  points in  $[a_1, \infty)$ , simply by walking down  $\mathcal{L}(a_1)$  and reporting the top- $k$  points in  $\Theta(k)$  time. We build such a list for each  $a_1 \in S$ .

We sort the points of  $S$  in non-decreasing order of their  $x$  coordinates (ties broken arbitrarily) as  $P_1, P_2, \dots, P_n$  and store them along with the points  $P_0 = -\infty$  and  $P_{n+1} = \infty$  in an array  $A$ . This defines  $n + 1$  intervals. Let  $I_j$  denote the interval  $(P_j, P_{j+1}]$ . For any  $a_1 \in I_j$ , the top- $k$  points in  $[a_1, \infty)$  are the same. With  $I_j$  we can store the list  $\mathcal{L}(P_{j+1})$ . Given a query  $q$ , we search in  $A$  to locate the interval  $I_j$  containing  $q$ . Then we query  $\mathcal{L}(P_{j+1})$  with  $q$  and report the top- $k$  points. The overall space requirement is  $O(n^2)$  and the query time is  $O(\log n + k)$ .

The space requirement can be reduced to  $O(n)$  by observing that  $\mathcal{L}(P_j)$  and  $\mathcal{L}(P_{j+1})$  are only slightly different.  $\mathcal{L}(P_{j+1}) = \mathcal{L}(P_j) \setminus P_j$ . Hence  $\mathcal{L}(P_{j+1})$  can be obtained from  $\mathcal{L}(P_j)$  with  $O(1)$  memory modifications. Treating the  $x$ -coordinate as time, we store all the lists in a partially persistent structure [10]. We start with  $L_0$  containing the points in  $S$  in non-increasing order of their weights. Then we obtain  $L(P_{j+1})$  from  $L(P_j)$  for  $j = 0, 1, \dots, n$ . These operations require  $O(n)$  memory modifications overall and hence the data structure occupies  $O(n)$  space.

**Lemma 2.** *A set,  $S$ , of  $n$  weighted points in  $\mathbb{R}^1$  can be preprocessed into a data structure of size  $O(n)$  so that for any query range  $q = [a_1, \infty)$ , and a given integer  $k$  satisfying  $1 \leq k \leq n$ , the points in  $S \cap q$  with the top- $k$  weights can be reported in  $O(\log n + k)$  time.*

Now we extend our solution to finite range (i.e.  $[a_1, b_1]$ ). We store the points of  $S$  at the leaves of a balanced binary search tree  $T$  in non-decreasing order of their  $x$ -coordinates. At each internal node  $v$ , we store an instance  $DL(v)$  of the data structure of Lemma 2 built on  $S(Left(v))$ , the set of points stored in the leaves of the left subtree of  $v$ . Similarly we store another data structure  $DR(v)$  built on  $S(Right(v))$ , the set of points stored in the leaves of the right subtree of  $v$  supporting top- $k$  queries of the form  $(-\infty, b_1]$ . To answer a query  $q = [a_1, b_1]$  we search with  $a_1$  and  $b_1$  in  $T$ . This generates paths  $l$  and  $r$  in  $T$  that possibly diverge at some non-leaf node  $u$  of  $T$ . We query  $DL(u)$  (respectively  $DR(u)$ ) with  $[a_1, \infty)$  (respectively  $(-\infty, b]$ ) to retrieve  $S_L$  (respectively  $S_R$ ), the top- $k$  points in the range. We can retrieve the overall top- $k$  points in  $[a_1, b_1]$  from  $S_L$

and  $S_R$  in  $O(k)$  time due to the fact that the top- $k$  problem is decomposable, as discussed in Section 1.2.

**Theorem 4.** *A set,  $S$ , of  $n$  weighted points in  $\mathbb{R}^1$  can be preprocessed into a data structure of size  $O(n \log n)$  so that for any query range  $q = [a_1, b_1]$ , and a given integer  $k$  satisfying  $1 \leq k \leq n$ , the points in  $S \cap q$  with the top- $k$  weights can be reported in  $O(\log n + k)$  time.*

### 3.2 Solution for $\mathbb{R}^d$ , $d \geq 2$

We first handle queries of the form  $q = [a_1, b_1] \times \dots \times [a_d, \infty]$ . Let the points in  $S$  be lying in a  $d$ -dimensional space. A  $d$ -dimensional range tree  $T$  is built based on the points in  $S$ . At the innermost level of  $T$  we employ fractional cascading [2]. Fractional cascading does not cause the space to increase but saves a log factor in the query time when a range search query is performed. Since fractional cascading has been applied, at the innermost level we will have arrays sorted based on their increasing  $x_d$ -coordinate values. At each such array (say  $M$ ) we do the following: Take each point  $p$  in  $M$  and associate a list  $L_p$  with it. The points which have  $x_d$ -coordinate value  $\geq$  than  $p$  in  $M$  are sorted in decreasing order based on their weights and put in the list  $L_p$ .

If  $M$  had  $m$  elements in it, then the total size of all the lists  $L_p$ , for all  $p \in M$  will be  $O(m^2)$ . However, notice that given two consecutive points  $p$  and  $q$  in  $M$ ,  $L_p$  can be obtained from  $L_q$  by making  $O(1)$  changes. Now treating the  $x_d$ -coordinate as time, we store all the lists  $L_p, \forall p \in M$ , in a partially persistent structure [10]. The overall memory modifications will be  $O(m)$  and hence the total size of all the lists reduces from  $O(m^2)$  to  $O(m)$ . Therefore, the overall size of the  $d$ -dimensional range tree remains  $O(n \log^{d-1} n)$ .

To answer a query, we do a standard search on our range tree with  $q$ . Let  $C$  be the set of  $O(\log^{d-1} n)$  canonical arrays selected. At each canonical array  $c \in C$ , we consider a list  $L_p$ , where  $p$  is the point having the least  $x_d$ -coordinate value among all the points in  $c$  having their  $x_d$ -coordinate value  $\geq a_d$ . A max-heap  $H$  is initialized with the first point from each of these lists  $L_p$ , using the point's weight as key. Then we repeat the following  $k$  times or until  $H$  is empty: We extract and report the point with maximum key in  $H$  and then access the list  $L_p$  that the reported point belongs to and insert the next point from this list into  $H$ .

To see the correctness of the query algorithm, note that the points of  $S$  that are in  $q$  are stored in the canonical arrays. The points of  $S$  not lying in  $q$  are never considered at any stage. When  $H$  is initialized, its root contains an unreported point from  $q$  with the largest weight and this property is maintained while  $H$  is queried and updated.

The time taken to search in  $T$  is  $O(\log^{d-1} n)$ . The time to initialize  $H$  is  $O(|C|)$ . The time to query and update  $H$  is  $O(\log |C|)$ . Since there are at most  $k$  queries and  $k$  updates on  $H$ , the total query time is  $O(\log^{d-1} n + k \log C) = O(\log^{d-1} n + k \log n)$ , since  $|C| = O(\log^{d-1} n)$ .

**Lemma 3.** *A set,  $S$ , of  $n$  weighted points in  $\mathbb{R}^d$  can be preprocessed into a data structure of size  $O(n \log^{d-1} n)$  so that given a query  $q=[a_1, b_1] \times \dots \times [a_d, \infty)$  and an integer  $k \in [1, n]$ , the top- $k$  points in  $S \cap q$  can be reported in sorted order of their weights in  $O(\log^{d-1} n + k \log \log n)$  time.*

Next we show how to handle queries of the form  $q=[a_1, b_1] \times \dots \times [a_d, b_d]$ . We store the points of  $S$  at the leaves of a balanced binary search tree  $BST$  in non-decreasing order of their  $x_d$  coordinates. At each internal node  $v$ , we store an instance  $DL(v)$  of the data structure of Lemma 3 built on  $S(left)$ , the set of points stored in the leaves of the left subtree of  $v$ . Similarly we store another data structure  $DR(v)$  built on  $S(right(v))$ , the set of points stored in the leaves of the right subtree of  $v$  supporting top- $k$  queries of the form  $q=[a_1, b_1] \times \dots \times (-\infty, b_d]$ .

Let  $q'=[a_1, b_1] \times \dots \times [a_{d-1}, b_{d-1}]$ . To answer a query  $q = q' \times [a_d, b_d]$  we search with  $a_d$  and  $b_d$  in  $BST$ . This generates paths  $l$  and  $r$  in  $BST$  that possibly diverge at some non-leaf node  $u$  of  $BST$ . We query  $DL(u)$  (respectively  $DR(u)$ ) with  $q' \times [a_d, \infty)$  (respectively  $q' \times (-\infty, b_d]$ ) to retrieve  $S_L$  (respectively  $S_R$ ), the top- $k$  points in the range. We can retrieve the overall top- $k$  points in  $q$  from  $S_L$  and  $S_R$  in  $O(k)$  time due to the fact that the top- $k$  problem is decomposable, as discussed in Section 1.2.

**Theorem 5.** *A set,  $S$ , of  $n$  weighted points in  $\mathbb{R}^d$  can be preprocessed into a data structure of size  $O(n \log^d n)$  so that given a query  $q=[a_1, b_1] \times \dots \times [a_d, b_d]$  and an integer  $k \in [1, n]$ , the top- $k$  points in  $S \cap q$  can be reported in sorted order of their weights in  $O(\log^{d-1} n + k \log \log n)$  time.*

## 4 Conclusions and Future Work

We have given efficient solutions for finding the top- $k$  points inside an orthogonal query box. Removing the  $O(\log \log n)$  penalty per reported point in the query time for the version wherein points are reported in sorted order of weights remains an important open problem.

## References

1. P. K. Agarwal, L. Arge, J. Yang, Ke Yi. I/O-Efficient Structures for Orthogonal Range-Max and Stabbing-Max Queries. *11th European Symposium on Algorithms*, 7–18, 2003.
2. P.K. Agarwal, and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, Contemporary Mathematics, 23, 1999, 1–56, American Mathematical Society Press.
3. S. Agarwal, S. Chaudhuri, G. Das, A. Gionis. Automated ranking of database query results. *First Biennial Conference on Innovative Data Systems Research*, 2003.
4. G. S. Brodal, R. Fagerberg, M. Greve, A. Lopez-Ortiz. Online Sorted Range Reporting. *20th International Symposium on Algorithms and Computation*, 173–182, 2009.

5. C. Buckley, G. Salton, J. Allan. The effect of adding relevance information in a relevance feedback environment. *17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, 292–300, 1994.
6. P. Cao, A. Wang. Efficient top- $k$  query calculation in distributed networks. *23rd Annual Symposium on Principles of Distributed Computing*, 206–215, 2004.
7. S. Chaudhuri, L. Gravano, A. Marian. Optimizing top- $k$  selection queries over multimedia repositories. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):992–1009, 2004.
8. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computing*, 17(3):427–462, June 1988.
9. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd edition, MIT Press, 2000.
10. 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.
11. H.N. Gabow, J.L. Bentley, R.E. Tarjan. Scaling and Related Techniques for Geometry Problems. *16th Annual ACM symposium on Theory of Computing*, 135–143, 1984.
12. P. Gupta, R. Janardan, and M. Smid. A technique for adding range restrictions to generalized searching problems. *Information Processing Letters*, 64 (1997), pp. 263–269.
13. I.F. Ilyas, G. Beskales, M.A. Soliman. A survey of top- $k$  query processing techniques in relational database systems. *ACM Computing Surveys* 40(4): (2008).
14. R. Kaushik, R. Krishnamurthy, J. F. Naughton, R. Ramakrishnan. On the integration of structure indexes and inverted lists. *International Conference on Management of Data (SIGMOD 2004)*, 779–790, 2004.
15. A. Marian, N. Bruno, L. Gravano. Evaluating top- $k$  queries over web-accessible databases. *ACM Transactions On Database Systems*, 29(2):319–362, 2004.
16. K. Mehlhorn and S. Naher. Dynamic Fractional Cascading, *Algorithmica*, 5:215–241, 1990.
17. J. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, (2):33–43, 1973.
18. D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, (32):597–617, 1982.
19. Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(12), 2004, 1555–1570.