# A General Technique for Top-$k$ Geometric Intersection Query Problems

Saladi Rahul and Ravi Janardan

**Abstract**—In a top-$k$ Geometric Intersection Query (top-$k$ GIQ) problem, a set of $n$ weighted, geometric objects in $\mathbb{R}^d$ is to be pre-processed into a compact data structure so that for any query geometric object, $q$, and integer $k > 0$, the $k$ largest-weight objects intersected by $q$ can be reported efficiently. While the top-$k$ problem has been studied extensively for non-geometric problems (e.g., recommender systems), the geometric version has received little attention.
This paper gives a general technique to solve any top-$k$ GIQ problem efficiently. The technique relies only on the availability of an efficient solution for the underlying (non-top-$k$) GIQ problem, which is often the case. Using this, asymptotically efficient solutions are derived for several top-$k$ GIQ problems, including top-$k$ orthogonal and circular range search, point enclosure search, halfspace range search, etc. Implementations of some of these solutions, using practical data structures, show that they are quite efficient in practice.
This paper also does a formal investigation of the hardness of the top-$k$ GIQ problem, which reveals interesting connections between the top-$k$ GIQ problem and the underlying (non-top-$k$) GIQ problem.

**Index Terms**—Range Search, Aggregation, Top-$k$, Query Processing and Optimization, Geometric Algorithms and Data Structures.

## 1 Introduction

### 1.1 Background and motivation

In a *Geometric Intersection Query* (GIQ) problem, we are given a set, $\mathcal{A}$, of $n$ geometric objects (e.g., points, lines, hyper-rectangles line segments, balls, etc.) in some ambient space $\mathbb{R}^d$. The goal is to organize $\mathcal{A}$ into a space-efficient data structure so that the following query can be answered efficiently: Given a query geometric object, $q$ (e.g., a rectangle or ball), report or count the objects of $\mathcal{A}$ that are intersected by $q$. (In the reporting version, the output is a list of all objects of $\mathcal{A}$ that are intersected by $q$; in the counting version, the output is simply the number of objects of $\mathcal{A}$ intersected by $q$.) Typically, the data structure needs to support a very large number of queries (in the hundreds of thousands), so it is worthwhile to pre-process $\mathcal{A}$ beforehand to speed up the queries. The performance measures of primary interest are the space occupied by the data structure and the query time. It is also desirable that the data structure support updates (insertion/deletion) of objects in $\mathcal{A}$. GIQ problems model many real-world query-retrieval problems arising in diverse domains, including GIS, spatial databases, VLSI design, robotics, CAD/CAM, and computer graphics. GIQ problems have been investigated extensively in the computational geometry and database literature and efficient solutions have been designed for many instances of these problems [1]–[3].

In recent years, there has been an explosion in the volume of digital data that is being generated and stored, and this growth promises to continue unabated as computing and storage costs drop. Major sources of voluminous digital data include social networks (e.g., Facebook, Twitter), the world-wide web, mobile devices (e.g., smart phones), GIS and spatial applications, business analytics, remote sensing, video surveillance, genomics, high-energy physics, etc. This proliferation of digital data has made it imperative that new ways be developed to query large datasets and make sense of the results. Traditional query methods that simply report all the data items satisfying a query are no longer sufficient as they place an undue burden on the user to sift through a potentially huge answer space for relevant information. Instead, what is really needed is an appropriate summary of the query results that can provide the user with useful information. Creating such a summary involves applying a suitable aggregation function to the query results. A simple, yet effective, aggregation function is the so-called top-$k$ function which returns a set of $k$ objects that best satisfy a query, where the notion of "best" is based on a real-valued weight assigned by the application of interest to each data object. The top-$k$ problem has been well studied in various domains, including web search, information retrieval, data mining, business analytics, recommender systems, etc.. Ilyas *et al.* [4] provide a comprehensive survey of the top-$k$ problem.

- *Saladi Rahul is with the Dept. of Computer Science & Engg., Univ. of Minnesota–Twin Cities, 4-192 Keller Hall, 200 Union St. S.E., Minneapolis, MN 55455, USA*
*E-mail: sala0198@umn.edu*
- *Ravi Janardan is with the Dept. of Computer Science & Engg., Univ. of Minnesota–Twin Cities, 4-192 Keller Hall, 200 Union St. S.E., Minneapolis, MN 55455, USA*
*E-mail: janardan@umn.edu*

In this paper, we investigate the top-$k$ problem in a geometric setting. Informally, we are given a set, $\mathcal{A}$, of $n$ geometric objects in $\mathbb{R}^d$, where each object has an associated real-valued weight. Our goal is to pre-process $\mathcal{A}$ into a space-efficient data structure so that for any query geometric object, $q$, and positive integer, $k$, the $k$ largest-weight objects of $\mathcal{A}$ intersected by $q$ can be reported efficiently. (We give a formal definition of the problem in Section 2.) We call this problem a *top-$k$ Geometric Intersection Query* (*top-$k$ GIQ*) problem. It is a useful and non-trivial generalization of the traditional GIQ problem described above and has applications in GIS, spatial databases, VLSI design, scheduling, etc.

We now motivate the study of the top-$k$ GIQ problem with specific examples.

- Consider a real-estate database that contains the location of each house for sale in the U.S., along with the asking price for the house. Potential buyers might be interested in knowing, say, the ten most expensive houses in a geographic area of interest, specified as a rectangle of some size centered around a landmark (e.g., a school or workplace). This is an instance of the *top-$k$ orthogonal range search* problem in $\mathbb{R}^2$, where $k = 10$, the weight of each house (point) is its cost, and the query, $q$, is an axes-aligned rectangle. (To identify the $k$ least expensive houses in a query region—which would likely be of greater interest to most buyers—one could use the reciprocal of the asking price as the weight.) If the area of interest is specified as a disc of some radius, then we have an instance of the *top-$k$ disc range search* problem in $\mathbb{R}^2$. (See Figure 1(a) and (b); to avoid clutter, all examples in Figure 1 assume that $k = 3$.)
- Consider a database that records buyers' preferences for some commodity, say, cars. Each buyer specifies the range of age and mileage that s/he is willing to consider for a car and the price that s/he is willing to offer for a car meeting these requirements. This information is recorded as a rectangle in the age-mileage space $\mathbb{R}^2$, with a weight equal to the offered price. A potential seller might be interested in identifying, say, the five most promising buyers (based on offered price) to negotiate with. This is an instance of the *top-$k$ point-enclosure search* problem in $\mathbb{R}^2$, where $k = 5$ and the query, $q$, is a point representing the age and mileage of the seller's car. (Each rectangle "stabbed" by $q$ is a buyer whose age and mileage requirements are satisfied by the seller.) (See Figure 1(c).)
- Finally, consider a financial database storing earnings ($e$) and volatility ($v$) information for a large number of stocks, as well as information on total return ($r$). Thus, each stock, $s$, is represented as a point $(s_e, s_v)$ in $\mathbb{R}^2$, with weight $s_r$. Each investor, $I$, has a different preference for income and risk, specified as percentages $I_e$ and $I_v$. A potential investor might wish to identify, say, the ten highest-return stocks, $s$, for which the weighted score $I_e \cdot s_e + I_v \cdot s_v$ is at least an investor-specified threshold $t_I$. The equation $I_e \cdot s_e + I_v \cdot s_v \geq t_I$ defines a halfplane in $\mathbb{R}^2$ and we are, thus, interested in the ten highest-return stocks in this halfplane. This is an instance of the *top-$k$ halfplane range search* problem in $\mathbb{R}^2$, where $k = 10$ and the query, $q$, is the above halfplane. (See Figure 1(d).)

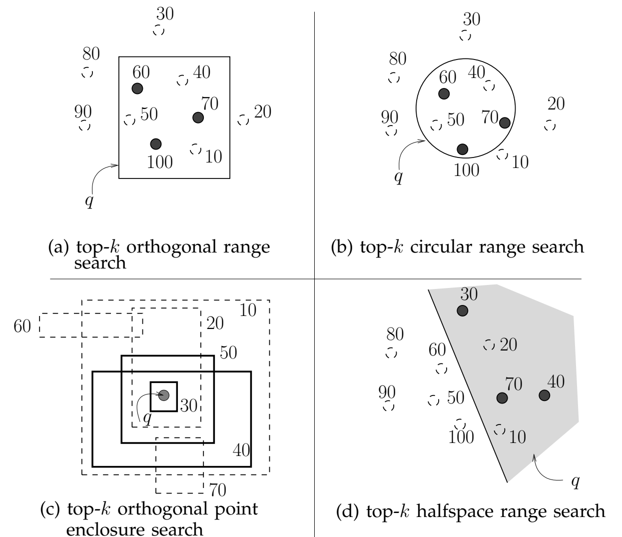



Fig. 1: Various instances of top-$k$ GIQ problems in $\mathbb{R}^2$. (The number next to an object—point or rectangle—is its weight.) In these examples, $k = 3$. Objects shown filled or drawn in heavy lines are the output for each query $q$.

For simplicity, these examples have all been in $\mathbb{R}^2$; however, they extend naturally to higher dimensions. Moreover, similar motivation can be provided for other instances of top-$k$ GIQ problems. For example, the top-$k$ segment intersection search problem discussed later is useful in VLSI design, where the segments model wires. The top-$k$ interval intersection search, also discussed later, is useful in scheduling, where intervals model execution times of jobs.

We note that there is an easy solution to the top-$k$ GIQ problem: Solve the underlying GIQ problem on $\mathcal{A}$ and $q$ (disregarding weights) using a known method and identify the set, $\mathcal{A}(q)$, of objects in $S$ intersected by $q$. Next, run a standard selection algorithm [5] on $\mathcal{A}(q)$ to identify the object with the $k$th-largest weight. Finally, scan $\mathcal{A}(q)$ to output the $k$ largest-weight objects.

Clearly, this approach is not efficient if $|\mathcal{A}(q)| \gg k$, which is often the case. Our goal is to design data structures with the following properties: (a) they have query times that are sensitive to $k$, typically of the form $O(f(n) + k))$ or $O(f(n) + k \cdot g(n))$, where $f(n)$

and $g(n)$ are "small" (e.g., polylogarithmic); (b) they have low storage requirements, typically (but not always) of the form $O(n \cdot \text{polylog}(n))$; and (c) they can support updates in $\mathcal{A}$ efficiently.[1] As we will see, these objectives can be met for a range of top-$k$ GIQ problems.

## 1.2 Contributions

**A general technique:** We present a general technique to solve any top-$k$ GIQ problem efficiently. Our approach only requires that efficient solutions to the reporting and counting versions of the underlying GIQ problem be available, which is often the case. Roughly speaking, our technique incurs only a logarithmic increase in space, query time, and update time over the corresponding bounds for the underlying GIQ problem. Specifically, we establish the following:

*Theorem 1.1:* Let $\mathcal{A}$ be a set of $n$ geometric objects in $\mathbb{R}^d$, each with a real-valued weight. Consider some GIQ problem on $\mathcal{A}$. Suppose that there is a data structure for the reporting (resp. counting) version of this GIQ problem that occupies $S_r(n,d)$ (resp. $S_c(n,d)$) space, answers queries for any query geometric object $q$ in $Q_r(n,d) + O(k)$ (resp. $Q_c(n,d)$) time, and supports updates in $U_r(n,d)$ (resp. $U_c(n,d)$) worst-case or amortized time. Assume that $S_c(n,d)/n$, $S_r(n,d)/n$, $Q_c(n,d)$, $Q_r(n,d)$, $U_c(n,d)$ and $U_r(n,d)$ are non-decreasing functions for non-negative values of $n$.

Then there is a data structure for the corresponding top-$k$ GIQ problem that uses $O(S(n,d)\log n)$ space, answers queries for any query pair $(q,k)$ in $O(Q(n,d)\log n + k)$ time, and supports updates in $O(U(n,d)\log n)$ amortized time, where
$S(n,d) = \max\{S_r(n,d), S_c(n,d)\}$,
$Q(n,d) = \max\{Q_r(n,d), Q_c(n,d)\}$ and
$U(n,d) = \max\{U_r(n,d), U_c(n,d)\}$.

By using Theorem 1.1, we obtain efficient solutions to a variety of top-$k$ GIQ problems, with the asymptotic performance bounds given in Table 1. To the best of our knowledge, these are the first known results for such a broad range of top-$k$ GIQ problems.

**Hardness of top-$k$ GIQ:** Theorem 1.1 implies that the complexity of a top-$k$ GIQ problem is upper-bounded by the complexity of the underlying GIQ problem times a logarithmic factor. This motivates the question of how hard a top-$k$ GIQ problem really is? That is, is its complexity lower than that of the underlying GIQ problem? Towards this end, we formally investigate the hardness of the top-$k$ GIQ problem and demonstrate interesting connections between it and the underlying GIQ problem. Informally, the key result is that a top-$k$ GIQ problem is at least as hard as its underlying GIQ problem. For specific instances of the top-$k$ GIQ problem, such as top-$k$ orthogonal range search and top-$k$ orthogonal point enclosure search, we are able to show an even stronger hardness result.

**Experimental results:** We complement the theoretical results in Table 1 by presenting experimental results for two of the problems in the table. Our implementations incorporate the key ideas in the general method but make use of more practical data structures than the ones used to obtain the theoretical results. Specifically, the results in Table 1 show the bounds that are achievable asymptotically (i.e., as $n \to \infty$). However, the underlying data structures for counting and reporting (e.g., range trees) can be inefficient for modest values of $n$—even for small $d$—as noted in [1]. For example, a $d$-dimensional range tree on $n$ points occupies $O(n \log^{d-1} n)$ space. This can be viewed as modestly superlinear as $n \to \infty$. However, for typical real-world datasets the term $\log^{d-1} n$ can actually be much larger than the dataset size $n$. (For example, if $n = 10^6$ and $d = 7$, then $\log^{d-1} n \approx 63 \times 10^6 \gg n$.) For this reason, in our implementation we have used $R$-trees [6] for the counting and reporting structures, as the storage requirement of an $R$-tree is linear in $n$ and the dependence on $d$ is multiplicative rather than exponential. Specifically, we build practical solutions for top-$k$ orthogonal range search and top-$k$ orthogonal point enclosure search. Our experiments show that our solutions for the implemented top-$k$ GIQ problems are quite efficient in practice, in terms of storage and query time. (The implemented solutions do not currently support updates, however.)

## 1.3 A sampling of related prior work

As mentioned earlier, the top-$k$ problem has been well-studied in many domains, including, for example, web search, information retrieval, recommender systems, etc. We refer the reader to Ilyas *et al.* for an excellent discussion of top-$k$ query processing in relational databases [4].

Surprisingly, the geometric version of the top-$k$ problem has received much less attention and known work on this has focused only on top-$k$ orthogonal range search. For this problem in $\mathbb{R}^1$, an algorithm with $O(n)$ space and $O(k)$ query time has been given by Brodal *et al.* [7] when the input consists of integers. For real inputs, Sheng *et al.* [8] and Afshani *et al.* [9] mention a solution with $O(n)$ space, $O(\log n + k)$ query time, and $O(\log n)$ update time. In $\mathbb{R}^d$, $d > 1$, an efficient solution is provided by Rahul et al. [10]. The present paper generalizes the result in [10], so as to make it applicable to a wide range of top-$k$ GIQ problems, and also demonstrates the practicality of the general technique through experimental evaluation.

Closely related to the top-$k$ problem is the problem of reporting only the point with the $k$th-heaviest weight in the query range. Gagie *et al.* [11] and

1. All logarithms in the paper are base 2.

| Top-$k$ GIQ problem | Objects in $\mathcal{A}$ | Query object $q$ | Dim. $d$ | Space occupied | Query time | Update time (amort.) |
|---|---|---|---|---|---|---|
| Orthogonal range search | points | hyper-rect. | $\geq 1$ | $O(n \log^d n)$ | $O(\log^{d+1} n + k)$ | $O(\log^{d+1} n)$ |
| Orthogonal point enclosure search | hyper-rects. | point | $\geq 2$ | $O(n \log^{d+1} n)$ | $O(\log^{d+1} n + k)$ | $O(\log^{d+1} n)$ |
| Halfspace range search | points | halfspace | $\geq 2$ | $O(m^{1+\varepsilon})$ | $O((\frac{n}{m^{\lfloor d/2 \rfloor}} \log n + k) \log n)$ $n \leq m \leq n^{\lfloor d/2 \rfloor}$ | $O(\frac{m^{1+\varepsilon}}{n})$ |
| Circular range search | points | ball | $\geq 2$ | $O(m^{1+\varepsilon})$ | $O((\frac{n}{m^{\lfloor (d+1)/2 \rfloor}} \log n + k) \log n)$ $n \leq m \leq n^{\lfloor (d+1)/2 \rfloor}$ | $O(\frac{m^{1+\varepsilon}}{n})$ |
| Circular point enclosure search | balls | point | | | | |
| Orthogonal segment intersection search | horizontal segments | vertical segment | 2 | $O(n \log^2 n)$ | $O(\log^3 n + k)$ | $O(\log^3 n)$ |
| Interval intersection search | intervals | interval | 1 | $O(n \log n)$ | $O(\log^3 n + k)$ | $O(\log^3 n)$ |

TABLE 1: Summary of performance bounds obtained for various top-$k$ GIQ problems using the general technique. ($\varepsilon > 0$ is an arbitrarily small constant.) Update time bounds are amortized.

Navarro *et al.* [12] answer this query in $\mathbb{R}^1$ and $\mathbb{R}^2$, respectively. Work has also been done on top-$k$ orthogonal range search in the context of document retrieval [13], [14] and for managing subscriptions in wide-area publish/subscribe networks [15].

Top-$k$ GIQ problems fall under the broad category of range aggregation which is a classical topic in the field of databases. As mentioned in [16] range aggregation "...has been studied in a large variety of contexts: relational [17], temporal [18], [19], spatial databases [20]–[22], OLAP [23]–[25], etc". In a typical range aggregate problem, we are given a set of objects such that, given a query object $q$, the query applies a suitable aggregate function on those objects that are intersected by $q$. Classic examples of aggregate functions include count, sum, max (or top-1), average etc.

### 1.4 Organization of the paper

In Section 2, we describe our general technique and establish Theorem 1.1. In Section 3, we apply Theorem 1.1 to specific top-$k$ GIQ problems and obtain the results stated in Table 1. In Section 4, we do a formal investigation of the hardness of the top-$k$ GIQ problem. In Section 5, we report on our experimental results. Finally, we conclude the paper in Section 6.

## 2 THE GENERAL TECHNIQUE

**Problem statement:** We are given a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of $n$ geometric objects in $\mathbb{R}^d$ ($d \geq 1$), where $a_i$ has a real-valued weight $w_i$, $1 \leq i \leq n$. We wish to organize $\mathcal{A}$ into a space-efficient data structure so that for any query pair $(q, k)$, where $q$ is a geometric object and $k > 0$ is an integer, we can report efficiently the $k$ largest-weight objects of $\mathcal{A}$ that are intersected by $q$. (Two geometric objects in $\mathbb{R}^d$ *intersect* iff they have a point in common.)

More precisely, let $\mathcal{A}(q)$ be the set of objects in $\mathcal{A}$ that are intersected by $q$. Then we wish to find and report the objects in a set $\mathcal{A}_k(q) \subseteq \mathcal{A}(q)$ such that $|\mathcal{A}_k(q)| = k$ and for any $a_i \in \mathcal{A}_k(q)$ and any $a_j \in \mathcal{A}(q) \backslash \mathcal{A}_k(q)$, we have $w_i \geq w_j$. (Note that if $q$ intersects $k$ or fewer objects of $\mathcal{A}$ then we simply report all of them.)

We first outline the key steps in our query algorithm and then discuss each step in detail.

**Key steps:** Given a query pair $(q, k)$, we do the following:

1) **Perform initial check:** Let $\mathcal{A}(q)$ be the set of objects of $\mathcal{A}$ intersected by $q$. If $|\mathcal{A}(q)| \leq k$, then we simply report all the objects in $\mathcal{A}(q)$ and stop. Otherwise, if $\mathcal{A}(q) > k$, we proceed to step 2.
2) **Find a threshold object:** We determine an object $a_t$ in $\mathcal{A}(q)$ that has the $k$th-largest weight and proceed to step 3. We call $a_t$ a *threshold object*.
3) **Report top-$k$ objects:** Given $a_t$, we report all objects in $\mathcal{A}(q)$ whose weights are greater than or equal to $w_t$.

As we will see, steps 1 and 3 are essentially instances of the underlying GIQ counting and reporting problems, respectively. Step 2 will employ a binary search-based approach to quickly identify $a_t$.

### 2.1 Implementation of step 1

Let $\mathcal{D}_C$ (resp. $\mathcal{D}_R$) denote a data structure for the counting (resp. reporting) version of the underlying GIQ problem on $\mathcal{A}$. That is, given a query object $q$, $\mathcal{D}_C$ (resp. $\mathcal{D}_R$) returns the count $|\mathcal{A}(q)|$ (resp. the set $\mathcal{A}(q)$). (For example, for the top-$k$ orthogonal range search problem, $\mathcal{D}_C$ (resp. $\mathcal{D}_R$) is a data structure for the counting (resp. reporting) version of orthogonal range search.) For future use, we assume that $\mathcal{D}_C$ and $\mathcal{D}_R$ support updates.

We do step 1 by querying $\mathcal{D}_C$ with $q$. If $|\mathcal{A}(q)| \leq k$, then we also query $\mathcal{D}_R$ with $q$ and output $\mathcal{A}(q)$.

### 2.2 Implementation of step 2

W.l.o.g. let $a_1, \ldots, a_n$ be an ordering of the objects of $\mathcal{A}$ by non-increasing weight (ties broken arbitrarily). Our goal is to find the $k$th-leftmost object in this ordering that is intersected by $q$; this is the threshold object $a_t$. Consider object $a_m$, where $m = \lfloor n/2 \rfloor$, and let $\mathcal{A}'$ (resp. $\mathcal{A}''$) be the ordered subset of $\mathcal{A}$ consisting of objects at or to the left of $a_m$ (resp. to the right of $a_m$). We count the number of objects in $\mathcal{A}'$ that are intersected by $q$, i.e., we compute $|\mathcal{A}'(q)|$. If $|\mathcal{A}'(q)| \geq k$, then $a_t$ is in $\mathcal{A}'$ and is the $k$th-leftmost object in $\mathcal{A}'(q)$. Therefore, we search recursively in $\mathcal{A}'$ for the $k$th-leftmost object. However, if $|\mathcal{A}'(q)| < k$, then $a_t$ is in $\mathcal{A}''$ and is the $(k - |\mathcal{A}'(q)|)$th-leftmost object in $\mathcal{A}''(q)$. Therefore, we search recursively in $\mathcal{A}''$ for the $(k - |\mathcal{A}'(q)|)$th-leftmost object.

We implement the above idea as follows: We sort the objects of $\mathcal{A}$ by non-increasing weight (breaking ties arbitrarily) and store them in left-to-right order at the leaves of a balanced binary search tree $\mathcal{T}$. At each node $v$ of $\mathcal{T}$, we store an instance, $\mathcal{D}_C^v$, of the structure $\mathcal{D}_C$ which is built on the objects stored in $v$'s subtree.

Let $r$ be the root of $\mathcal{T}$. At the beginning of this step, our objective is to find the $k$th-leftmost leaf among the leaves of $\mathcal{T}$ that store objects intersected by $q$; this leaf contains $a_t$. However, as the algorithm progresses and reaches some subtree of $\mathcal{T}$, our objective will change in the sense that we will now be seeking the $k'$th-leftmost leaf among the leaves of this subtree that store objects intersected by $q$, for some $k' \leq k$.

Specifically, let $v_{cur}$ denote the root of the subtree of $\mathcal{T}$ that the search is at currently. Initially, we know (from step 1) that $q$ intersects more than $k$ objects among the ones stored in $\mathcal{T}$'s leaves, so we must search in the left subtree of $r$ for the $k$-th leftmost object intersected by $q$. Thus, initially $v_{cur}$ is set to the left child of $r$ and $k'$ is set to $k$. Let $C(v_{cur})$ be the count returned when $\mathcal{D}_C^{v_{cur}}$ is queried with $q$. If $C(v_{cur}) \geq k'$, then the leaf containing $a_t$ is in the left subtree of $v_{cur}$, so the search proceeds to this subtree with $k'$ unchanged. However, if $C(v_{cur}) < k'$, then the leaf containing $a_t$ is in the subtree of the sibling of $v_{cur}$, so the search proceeds to the sibling's subtree with $k'$ set to $k' - C(v_{cur})$. This process repeats iteratively until the leaf, $u$, containing $a_t$ is reached.

### 2.3 Implementation of step 3

We store the objects of $\mathcal{A}$ at the leaves of a balanced binary search tree $\mathcal{T}'$, in the same order in which they appear at the leaves of $\mathcal{T}$. At each node $v$ of $\mathcal{T}'$, we store an instance, $\mathcal{D}_R^v$, of the structure $\mathcal{D}_R$ which is built on the objects stored in $v$'s subtree. Also, if object $a_i$ appears at leaf $u$ of $\mathcal{T}$ and at a leaf $u'$ of $\mathcal{T}'$, then we store a pointer, $ptr$, at $u$ that points to $u'$; i.e., $ptr(u) = u'$. (In fact, we could use $\mathcal{T}$ to store instances of both $\mathcal{D}_C$ and $\mathcal{D}_R$. We use a separate structure $\mathcal{T}'$ only for ease of exposition.)

To report all objects in $\mathcal{A}(q)$ whose weights are greater than or equal to $w_t$, we query $\mathcal{T}'$ with $q$, as follows:

Let $u$ be the leaf of $\mathcal{T}$ that is found to contain the threshold object $a_t$ in step 2. We follow $ptr(u)$ to find the leaf $u'$ of $\mathcal{T}'$ that contains $a_t$. We then walk from $u'$ up to the root of $\mathcal{T}'$ following parent pointers, thereby tracing a path, $\Pi$, in $\mathcal{T}'$. Let $Z$ be the set of nodes, $v$, in $\mathcal{T}'$ such that $v$ is the left child of a node on $\Pi$ but is itself not on $\Pi$. We also include in $Z$ the leaf $u'$. $Z$ consists of both leaves and internal nodes and we call each such node a *canonical node*. Note that $|Z| = O(\log n)$ and, moreover, for each $v \in Z$, the range $[w_t, \infty)$ contains the weights of all the objects stored in $v$'s subtree. For each $v \in Z$, we query $\mathcal{D}_R^v$ with $q$, which causes all objects in $v$'s subtree that are intersected by $q$ to be reported.

This concludes the description of the 3-step query algorithm. The algorithm is presented in pseudocode as Algorithm 1.

### 2.4 An example

We illustrate the query algorithm in Figure 2. Part (a) shows the input objects (points in $\mathbb{R}^2$) and the query object $q$ (a rectangle), part (b) shows the structure $\mathcal{T}$, and part (c) shows the structure $\mathcal{T}'$. To avoid clutter, the structures $\mathcal{D}_C$ and $\mathcal{D}_R$ are not shown at the nodes. For simplicity, we refer to the points by their weights. For $k = 4$, the threshold point is 40 and the top-4 points are 80, 60, 50, and 40. Let $v_1$ be the root of $\mathcal{T}$. Step 1 finds $C(v_1)$ to be 5, corresponding to the points 80, 60, 50, 40, and 20 in $v_1$'s subtree that lie inside $q$. Since $C(v_1) > k$ we proceed to Step 2.

In Step 2 our objective is to find a leaf node $v$ such that among the points stored at $v$ and the leaf nodes to the left of $v$, exactly 4 points lie inside $q$. Initially, $v_{cur} = v_2$ and $k' = k = 4$. Querying $\mathcal{D}_C^{v_2}$ gives $C(v_2) = 3$, corresponding to the points 80, 60, and 50 in $v_2$'s subtree that lie inside $q$. Thus, the threshold point is not in the subtree of $v_2$ but instead is in the subtree of its sibling node $v_3$. Since $k = 4$ and $C(v_2) = 3$, $k'$ is reset to $k - A(v_2) = 1$, and the search proceeds to $v_3$. Querying $\mathcal{D}_C^{v_3}$ gives $C(v_3) = 2$, corresponding to the points 40 and 20 in $v_3$'s subtree that lie inside $q$. Since $C(v_3) > k'$, we proceed to $v_3$'s left child $v_4$. Querying $\mathcal{D}_C^{v_4}$ we find that $C(v_4) = 1$, corresponding to point 40 lying inside $q$. Since $C(v_4) = k'$, we proceed to $v_4$'s left child $v_5$. Querying $\mathcal{D}_C^{v_5}$ yields $C(v_5) = 1$, corresponding to point 40 lying inside $q$. Since $C(v_5) = k'$, we proceed to $v_5$'s left child which happens to be *nil*. At this point we exit the **while**-loop with $u = v_5$ containing the threshold point 40.

Finally, in step 3, we follow $ptr(v_5)$ (not shown) to locate the leaf in $\mathcal{T}'$ storing threshold point 40, identify the path $\Pi$ and the set $Z$ of canonical nodes, and query $\mathcal{D}_R^v$ at each node $v \in Z$ with $q$ to report the top-4 points in $q$.

**Algorithm 1:** Query algorithm for top-$k$ GIQ

**Input**: Data structures $\mathcal{T}$ and $\mathcal{T}'$ storing objects of $\mathcal{A}$ as described in Sections 2.1–2.3, query object $q$, and integer $k > 0$.
**Output**: The $k$ largest-weight objects of $\mathcal{A}$ that are intersected by $q$.

**begin**

```
// Step 1
```

Query $\mathcal{D}_C^r$ with $q$ to compute the number, $C(r)$, of objects in $r$'s subtree that are intersected by $q$, where $r$ is the root of $\mathcal{T}$.
**if** $C(r) \leq k$ **then**
  Query $\mathcal{D}_R^{r'}$ with $q$ to find all the objects in $r'$'s subtree that are intersected by $q$, where $r'$ is the root of $\mathcal{T}'$. Report these objects and exit.

```
// Step 2
```

$v_{cur} \longleftarrow$ left child of $r$
$k' \longleftarrow k$
**while** $v_{cur} \neq nil$ **do**
  Query $\mathcal{D}_C^{v_{cur}}$ with $q$ to compute the number of objects, $C(v_{cur})$, in $v_{cur}$'s subtree that are intersected by $q$.
  $u \longleftarrow v_{cur}$
  **if** $C(v_{cur}) < k'$ **then**
    $k' \longleftarrow k' - C(v_{cur})$
    $v_{cur} \longleftarrow$ sibling of $v_{cur}$
  **else**
    $v_{cur} \longleftarrow$ left child of $v_{cur}$

```
// Step 3
```

$u' \longleftarrow$ leaf of $\mathcal{T}'$ corresponding to $u$
Walk up $\mathcal{T}'$ from $u'$ and identify the set, $Z$, of canonical nodes. For each $v \in Z$, query $\mathcal{D}_R^v$ with $q$ and report all objects returned.

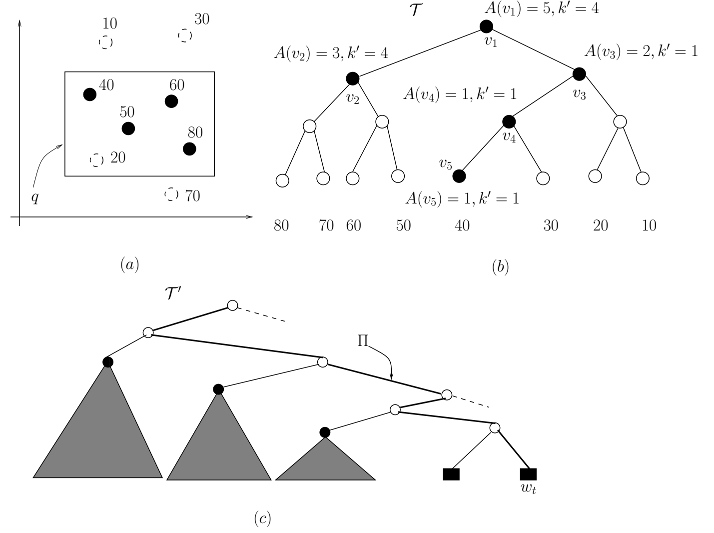



Fig. 2: The general technique illustrated for top-$k$ orthogonal range search in $\mathbb{R}^2$, with $k = 4$. (a) Set $\mathcal{A}$ consisting of 8 weighted points and query rectangle $q$. Points shown filled are the $k$ largest-weight objects intersected by $q$. (b) Finding the threshold point by querying $\mathcal{T}$. (c) Search path, $\Pi$, in $\mathcal{T}'$ (shown in heavy lines) and canonical nodes (shown filled).

### 2.5 Proof of Theorem 1.1

The correctness of the query algorithm follows from the discussion in Sections 2.1–2.3.

We now analyze the space bound. Let $v_1, v_2, \ldots, v_t$ be the nodes of $\mathcal{T}$ at a given level (i.e., distance from the root) and let $n_1, n_2, \ldots, n_t$ be, respectively, the number of objects stored at the leaves of their subtrees. The space used by all the secondary structures, $\mathcal{D}_C^{v_i}$, at these nodes is $\sum_{i=1}^t S_c(n_i, d) = \sum_{i=1}^t (S_c(n_i, d)/n_i) \times n_i \leq (S_c(n, d)/n) \sum_{i=1}^t n_i = O(S_c(n, d))$, since $S_c(n_i, d)/n_i$ is non-decreasing, $n_i \leq n$, and $\sum_{i=1}^t n_i \leq n$. Since $\mathcal{T}$ has height $O(\log n)$, the space used by $\mathcal{T}$ is $O(S_c(n, d) \log n)$. Similarly, the space used by $\mathcal{T}'$ is $O(S_r(n, d) \log n)$. Thus, the overall space is $O(\max\{S_c(n, d), S_r(n, d)\} \log n) = O(S(n, d) \log n)$.

Next, we analyze the query time. The time for step 1 is $O(Q_c(n, d) + Q_r(n, d) + k)$. For step 2, consider the path in $\mathcal{T}$ from the root to the leaf node containing $a_t$. At each node $v$ on the path, the secondary structure $\mathcal{D}_C^v$ is queried with $q$. Also, if $v$ is a right child of its parent, then the secondary structure at the left child of $v$'s parent is also queried. So, at each level of $\mathcal{T}$, the secondary structures of at most two nodes, $v_1$ and $v_2$, at that level are queried. Let $n_i$, $i = 1, 2$, be the number of objects stored at the leaves of $v_i$'s subtree. Thus step 2 takes $\sum_{i=1}^2 Q_c(n_i, d)$ time. Since $Q_c(n_i, d)$ is non-decreasing and $n_i \leq n$, $i = 1, 2$, the query time per level is $\sum_{i=1}^2 Q_c(n_i, d) = O(Q_c(n, d))$. Summing over the $O(\log n)$ levels of $\mathcal{T}$ gives an overall query time of $O(Q_c(n, d) \log n)$ time for step 2.

In Step 3, it takes $O(\log n)$ time to identify the set, $Z$, of canonical nodes. For each $v_i \in Z$, let $n_i$ be the number of objects stored at the leaves of $v_i$'s subtree and let $k_i$ be the number of these objects intersected by $q$. Querying $\mathcal{D}_R^{v_i}$ with $q$ at each $v_i \in Z$ takes $O(Q_r(n_i, d) + k_i)$ time. Thus the total query time in step 3 is $O(\sum_{i=1}^{|Z|}(Q_r(n_i, d) + k_i))$. Since $Q_r(n_i, d)$ is non-decreasing, and since $\sum_{i=1}^{|Z|} k_i = k$, the query time for step 3 is $O(Q_r(n, d) \log n + k)$.

Therefore, the time for steps 1–3 is $O(\max\{Q_c(n, d) + Q_r(n, d)\} \log n + k) = O(Q(n, d) \log n + k)$.

Finally, we consider the update time. If $\mathcal{T}$ and $\mathcal{T}'$ are implemented as $BB(\alpha)$ trees, then the technique of Willard *et al.* [26] can be used to keep the trees balanced as updates are performed. As shown in [26], the amortized update time for $\mathcal{T}$ and $\mathcal{T}'$ will be $O(U_c(n, d) \log n)$ and $O(U_r(n, d) \log n)$, respectively. Thus, the overall update time will be

$O(\max\{U_c(n,d) + U_r(n,d)\} \log n) = O(U(n,d) \log n)$ (amortized).

## 3 Applications of the General Technique

We apply the general technique to specific top-$k$ GIQ problems and obtain the bounds given in Table 1.

### 3.1 Top-$k$ orthogonal range search

Here $\mathcal{A}$ is a set of $n$ weighted points in $\mathbb{R}^d$ ($d \geq 2$) and $q$ is an axes-aligned hyper-rectangle. The goal is to report the $k$ largest-weight points intersected by (i.e., lying in) $q$.

For $\mathcal{D}_C$ and $\mathcal{D}_R$ we use a $d$-dimensional range tree [2], [27]. $S_c(n,d) = S_r(n,d) = O(n \log^{d-1} n)$, $Q_c(n,d) = Q_r(n,d) = O(\log^d n)$ and $U_c(n,d) = U_r(n,d) = O(\log^d n)$. See the Appendix for the description of a range tree.

Substituting these bounds into Theorem 1.1 yields a data structure for top-$k$ orthogonal range search that occupies $O(n \log^d n)$ space, answers queries in $O(\log^{d+1} n + k)$ time, and supports updates in $O(\log^{d+1} n)$ amortized time. (These bounds correct the bounds stated in [10].)

### 3.2 Top-$k$ orthogonal point enclosure search

Here $\mathcal{A}$ is a set of $n$ weighted hyper-rectangles in $\mathbb{R}^d$ ($d \geq 1$) and $q$ is a point. The goal is to report the $k$ largest-weight hyper-rectangles intersected by (i.e., containing) $q$.

For both $\mathcal{D}_C$ and $\mathcal{D}_R$, we use a $d$-dimensional segment tree for which $S_c(n,d) = S_r(n,d) = O(n \log^d n)$, $Q_c(n,d) = O(\log^d n)$, $Q_r(n,d) = O(\log^d n)$, and $U_c(n,d) = U_r(n,d) = O(\log^d n)$ (amortized). See the Appendix for a brief description of a segment tree. (See [2] for full details.)

Substituting these bounds into Theorem 1.1 yields a data structure for top-$k$ orthogonal point enclosure search with the bounds stated in Table 1.

### 3.3 Top-$k$ halfspace range searching

Here $\mathcal{A}$ is a set of $n$ weighted points in $\mathbb{R}^d$ ($d \geq 2$) and $q$ is a halfspace. The goal is to report the $k$ largest-weight points intersected by (i.e., lying in) $q$.

For $\mathcal{D}_R$ we use the structure given in [28] for which $S_r(n,d) = O(m^{1+\varepsilon})$, $Q_r(n,d) = O(\frac{n}{m^{\lfloor d/2 \rfloor}} \log n)$, and $U_r(n,d) = O(\frac{m^{1+\varepsilon}}{n})$ (amortized), for any $\varepsilon > 0$ and $n \leq m \leq n^{\lfloor d/2 \rfloor}$.

To the best of our knowledge, there is no efficient dynamic data structure known for halfspace range counting. Instead, we use for $\mathcal{D}_C$ the structure $\mathcal{D}_R$ itself. We observe that $\mathcal{D}_C$ is used in step 2 of the query algorithm to test at each visited node of $\mathcal{T}$ whether $q$ intersects fewer than $k'$ points in the node's subtree, where $k' \leq k$. When using $\mathcal{D}_R$ instead of $\mathcal{D}_C$, we simply keep a count of the objects as they are being output. If the query terminates and the count is less than $k'$, then the outcome of the test is "true"; otherwise, if the count reaches $k'$, then we stop listing objects and register the outcome of the test as "false". Thus, $Q_c(n,d) = O(Q_r(n,d) + k)$. Also, $S_c(n,d) = S_r(n,d)$ and $U_c(n,d) = U_r(n,d)$.

Substituting these into Theorem 1.1 yields a structure for top-$k$ halfspace range search that uses $O(m^{1+\varepsilon})$ space, answers queries in $O((\frac{n}{m^{\lfloor d/2 \rfloor}} \log n + k) \log n)$ time, and allows updates in $O(\frac{m^{1+\varepsilon}}{n})$ amortized time. (In the space and update time bounds, the $\log n$ factor implied by Theorem 1.1 is subsumed by $m^{\varepsilon}$.)

### 3.4 Top-$k$ circular range search

Here $\mathcal{A}$ is a set of $n$ weighted points in $\mathbb{R}^d$ ($d \geq 2$) and $q$ is a (closed) ball. The goal is to report the $k$ largest-weight points intersected by (i.e., lying in) $q$.

We transform this problem to top-$k$ halfspace range search in $\mathbb{R}^{d+1}$ via the well-known *lifting* transformation [29] and apply the result from Section 3.3. The lifting transformation maps each point of $\mathcal{A}$ to a point in $\mathbb{R}^{d+1}$ and maps $q$ to a halfspace in $\mathbb{R}^{d+1}$, as follows:

Let $(x_1, x_2, \ldots, x_{d+1})$ be the system of coordinates in $\mathbb{R}^{d+1}$. Let $\mathcal{U}$ be a paraboloid of revolution in $\mathbb{R}^{d+1}$ given by $x_{d+1} = x_1^2 + x_2^2 + \ldots + x_d^2$. We map each point $a_i = (x_1(a_i), \ldots, x_d(a_i))$ in $\mathcal{A}$ to the point $\hat{a_i} = (x_1(a_i), \ldots, x_d(a_i), x_1(a_i)^2 + \cdots + x_d(a_i)^2)$ in $\mathbb{R}^{d+1}$, i.e., we "lift" $a_i$ in in the positive $x_{d+1}$-direction so that it lies on $\mathcal{U}$. We assign $\hat{a_i}$ a weight equal to that of $a_i$, i.e., $w_i$. Let $\hat{\mathcal{A}}$ be this set of weighted points in $\mathbb{R}^{d+1}$.

We map $q$ as follows: Let $\hat{q}$ be the unique hyperplane in $\mathbb{R}^{d+1}$ whose intersection with $\mathcal{U}$ is precisely the lifted image of the boundary of $q$ onto $\mathcal{U}$. Let $\hat{q}^-$ be the closed halfspace lying on or below $\hat{q}$ (w.r.t. the negative $x_{d+1}$-direction). Then $q$ is mapped to $\hat{q}^-$.

The crucial property of this transformation is that it preserves the relative positions of $a_i$ and $q$, i.e., $a_i$ is in $q$ (interior or boundary) iff $\hat{a_i}$ is in $\hat{q}^-$. (Thus, $a_i$ is not in $q$ iff $\hat{a_i}$ is in the open halfspace above $\hat{q}$.) Figure 3 illustrates the preceding discussion for $d = 2$.

Thus, our problem in $\mathbb{R}^d$ is transformed to top-$k$ halfspace range search in $\mathbb{R}^{d+1}$, on the set $\hat{\mathcal{A}}$ with query halfspace $\hat{q}^-$. The result in Section 3.3 then yields the bounds stated in Table 1.

### 3.5 Top-$k$ circular point enclosure search

Here $\mathcal{A}$ is a set of $n$ weighted (closed) balls in $\mathbb{R}^d$ ($d \geq 2$) and $q$ is a point. The goal is to report the $k$ largest-weight balls intersected by (i.e., containing) $q$.

This problem, too, can be transformed to top-$k$ halfspace range search in $\mathbb{R}^{d+1}$ through the successive application of two transformations: lifting followed by *point-hyperplane duality*—another well-known transformation [29], [30] that we will discuss shortly.

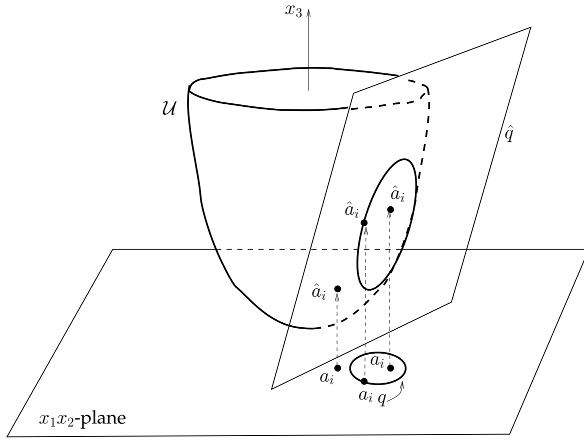



Fig. 3: The lifting transformation for $d = 2$. Point $a_i$ is in the closed disc $q$ if point $\hat{a}_i$ is on or below the plane $\hat{q}$ and $a_i$ is not in $q$ if $\hat{a}_i$ is above $\hat{q}$.

Using the lifting transformation discussed in Section 3.4, we map each ball $a_i \in \mathcal{A}$ to a closed halfspace $\hat{a}_i^-$ and map $q$ to a point $\hat{q}$, all in $\mathbb{R}^{d+1}$. By the properties of the lifting transformation, $q$ is in $a_i$ iff $\hat{q}$ is in $\hat{a}_i^-$.

Next we apply the point-hyperplane duality transformation to the lifted objects in $\mathbb{R}^{d+1}$ and map non-vertical hyperplanes in $\mathbb{R}^{d+1}$ to points in $\mathbb{R}^{d+1}$, and vice versa.

Specifically, a point $u = (u_1, \ldots, u_{d+1})$ in $\mathbb{R}^{d+1}$ is mapped to the non-vertical hyperplane $u' : x_{d+1} = 2u_1x_1 + \cdots + 2u_dx_d - u_{d+1}$ and a non-vertical hyperplane $H : x_{d+1} = h_1x_1 + \cdots + h_dx_d + h_{d+1}$ is mapped to the point $H' = (h_1/2, \ldots, h_d/2, -h_{d+1})$. It is easily verified that point $u$ is above/on/below hyperplane $H$ (w.r.t the negative $x_{d+1}$-direction) iff the dual point $H'$ is above/on/below the dual hyperplane $u'$.

Using this duality transformation, for each $a_i \in \mathcal{A}$, we map the corresponding non-vertical hyperplane $\hat{a}_i$, bounding halfspace $\hat{a}_i^-$, to a point $\hat{a}_i'$ and map point $\hat{q}$ to the non-vertical hyperplane $\hat{q}'$. By the properties of the duality transformation, point $\hat{q}$ is in the closed halfspace $\hat{a}_i^-$ iff the point $\hat{a}_i'$ is in the closed halfspace $\hat{q}'^-$ that lies on or below $\hat{q}'$ .

Thus, our original problem in $\mathbb{R}^d$ has been transformed to top-$k$ halfspace range search in $\mathbb{R}^{d+1}$. The result in Section 3.3 then leads to the bounds given in Table 1.

### 3.6 Top-$k$ orthogonal segment intersection search

Here $\mathcal{A}$ consists of $n$ weighted horizontal line segments in $\mathbb{R}^2$ and $q$ is a vertical line segment. The goal is to report the $k$ largest-weight horizontal segments intersected by $q$.

We are unaware of any previous dynamic data structure $\mathcal{D}_C$ for the counting version of the underlying GIQ problem. Therefore, we provide one below.

We build a segment tree $\mathcal{S}$ [27] on the intervals obtained by vertically projecting the segments of $\mathcal{A}$ onto the horizontal axis. Let $S_v$ be the set of segments assigned to node $v \in \mathcal{S}$. We project the segments of $S_v$ horizontally on the vertical axis to obtain a set of points and build a balanced binary search tree, $\mathcal{B}_v$ on these points. This constitutes the structure $\mathcal{D}_C$.

Given $q$, we search in $\mathcal{S}$ with the vertical projection of $q$ (a point). Let $\Pi$ be the search path. At each node $v \in \Pi$, we query $\mathcal{B}_v$ with the interval that is the horizontal projection of $q$ on the vertical axis and determine the number of points contained in the projection. We sum this count over all $v \in \Pi$ to obtain $|\mathcal{A}(q)|$, i.e., the number of segments of $\mathcal{A}$ intersected by $q$. $\mathcal{D}_C$ occupies $S_c(n, d) = O(n \log n)$ space and answers queries in $Q_c(n, d) = O(\log^2 n)$. It can also be dynamized to handle updates in $U_c(n, d) = O(\log^2 n)$ amortized time [27]

For $\mathcal{D}_R$, we use a structure given in [31], which uses $S_r(n, d) = O(n)$ space, answers queries in $Q_r(n, d) = O(\log^2 n)$, and supports updates in $U_r(n, d) = O(\log n)$ time. Substituting these into Theorem 1.1 yields the bounds stated in Table 1.

### 3.7 Top-$k$ interval intersection search

Here $\mathcal{A}$ consists of $n$ weighted intervals in $\mathbb{R}^1$ and $q$ is also an interval. The goal is to report the $k$ largest-weight intervals intersected by $q$.

Let the intervals in $\mathcal{A}$ be $a_i = [\ell(a_i), r(a_i)]$ and let $q = [\ell(q), r(q)]$. Now, $q$ intersects $a_i$ iff $r(q) \geq \ell(a_i)$ and $\ell(q) \leq r(a_i)$, i.e., iff $-\infty < \ell(a_i) \leq r(q)$ and $\ell(q) \leq r(a_i) < \infty$, i.e., iff the point $a_i' = (\ell(a_i), r(a_i))$ is contained in the quadrant $q' = (-\infty, r(q)] \times [\ell(q), \infty)$.

We map each $a_i \in \mathcal{A}$ to $a_i'$ and assign it a weight equal to that of $a_i$, i.e., $w_i$. We map $q$ to $q'$.

Thus, our problem has been transformed to a special case of top-$k$ orthogonal range search in $\mathbb{R}^2$, where the query $q'$ is a semi-infinite rectangle. This allows us to use for $\mathcal{D}_R$ a priority search tree [2] (instead of a range tree) for which $S_r(n, d) = O(n)$, $Q_r(n, d) = O(\log n)$, and $U_r(n, d) = O(\log n)$, where $d = 2$. For $\mathcal{D}_C$ we use the data structure given in [32], for which $S_c(n, d) = O(n)$, $Q_c(n, d) = O(\log^2 n)$, and $U_c(n, d) = O(\log^2 n)$ (amortized), where $d = 2$. Applying Theorem 1.1 gives the bounds in Table 1. (If we had used range tree, the space would have been higher by a $\log n$ factor.) See the Appendix for the description of a priority search tree.

## 4 Hardness of top-$k$ GIQ

As mentioned in Section 1.2, an interesting question is to characterize the complexity of the top-$k$ GIQ problem relative to that of the underlying GIQ problem. In this section, we perform a formal investigation of this question and show that former is at least as hard as the latter.

### 4.1 Reduction from GIQ in $\mathbb{R}^d$

Informally, the main result of this subsection is that the top-$k$ GIQ problem is at least as hard as the underlying GIQ problem (reporting version). The formal statement is given below.

*Theorem 4.1:* There exists a reduction from a GIQ problem in $\mathbb{R}^d$ to the corresponding top-$k$ GIQ problem in $\mathbb{R}^d$.

*Proof:* Assume there is a data structure $\mathcal{D}$ which can solve the top-$k$ GIQ problem in $\mathbb{R}^d$. Given $n$ objects, let $S(n)$ be the space occupied by the data structure and $Q(n)+O(k)$ be the time taken to answer the top-$k$ query.

Now we show how to solve a GIQ problem in $\mathbb{R}^d$ on $n$ objects using a data structure that uses $O(S(n))$ space and has a query time of $Q(n)+O(t)$, where $t$ is the number of objects reported.

Let $\mathcal{A}$ be the set of $n$ objects in $\mathbb{R}^d$ on which we shall solve the GIQ problem. Each object $a_i \in \mathcal{A}$ is assigned an arbitrary real weight, $w_i$. Next we build an instance of the data structure, $\mathcal{D}$, based on these weighted objects. Clearly the space occupied by the data structure will be $O(S(n))$.

Given a query object, $q$, the query is executed in multiple rounds: In round $j$ (starting from $j=1$), we query $\mathcal{D}$ with $(q, k = 2^{j-1} \cdot Q(n))$. If fewer than $2^{j-1} \cdot Q(n)$ objects are reported, then it means all the objects in $\mathcal{A}(q)$ have been reported and, hence, we stop. Otherwise, we go to round $j+1$. It is clear that this query algorithm will report all the objects in $\mathcal{A}(q)$ and will not report any other object in $\mathcal{A}$.

Now we analyze the time taken to solve the GIQ problem. There are two cases:

1) Only one round is executed: Then the query time will be $O(Q(n))$ since the value of $k = Q(n)$.
2) There are $i > 1$ rounds of execution: In round $j$ the number of objects reported is bounded from above by $2^{j-1} \cdot Q(n)$. Then the query time can be written as $O(\sum_{j=1}^{i}(Q(n) + 2^{j-1}Q(n))) = O(iQ(n) + \sum_{j=1}^{i} 2^{j-1}Q(n))) = O(iQ(n) + 2^iQ(n)) = O(2^i \cdot Q(n))$. The crucial observation is that since the $(i-1)$-th round was executed and we then entered $i$-th round, we have $t \geq 2^{i-2}Q(n)$ which implies that $2^iQ(n) \leq 4t$. Therefore, the query time will be $O(2^i \cdot Q(n)) = O(t)$.

Therefore, the overall query time will be $O(Q(n)+t)$. □

### 4.2 Reduction from GIQ in $\mathbb{R}^{d+1}$

If we restrict our attention to top-$k$ orthogonal range search and top-$k$ orthogonal point enclosure search, then a stronger reduction can be shown. First we consider top-$k$ orthogonal range search and then top-$k$ orthogonal point enclosure search.

In this section, we consider hyper-rectangles in $\mathbb{R}^{d+1}$ for which the $(d+1)$st dimension is semi-infinite (of the form $[x_{d+1}, \infty)$).

*Theorem 4.2:* There exists a reduction from orthogonal range reporting in $\mathbb{R}^{d+1}$ to top-$k$ orthogonal range search in $\mathbb{R}^d$.

*Proof:* Assume there is a data structure, $\mathcal{D}$, which can answer the top-$k$ orthogonal range search query in $\mathbb{R}^d$. Given $n$ points, let $S(n)$ be the space occupied by the data structure and $Q(n) + O(k)$ be the time taken to answer the top-$k$ query.

Suppose we have a set, $\mathcal{A}$, of $n$ points in $\mathbb{R}^{d+1}$ and we want to answer an orthogonal range search query on $\mathcal{A}$ for any given query hyper-rectangle $q = [x_1, y_1] \times [x_2, y_2] \times \ldots [x_d, y_d] \times [x_{d+1}, \infty)$. This can be done using a data structure that uses $O(S(n))$ space and has a query time of $O(Q(n) + t)$, where $t$ is the number of points reported.

Each point $a_i = (x_1(a_i), \ldots, x_d(a_i), x_{d+1}(a_i)) \in \mathcal{A}$ in $\mathbb{R}^{d+1}$ is mapped to a point $a_i = (x_1(a_i), \ldots, x_d(a_i)) \in \mathbb{R}^d$ and assigned a weight $x_{d+1}(a_i)$. Based on these newly mapped points we build an instance of the data structure $\mathcal{D}$. Clearly the space occupied by $\mathcal{D}$ will be $O(S(n))$.

Given a query region $q = [x_1, y_1] \times [x_2, y_2] \times \ldots [x_d, y_d] \times [x_{d+1}, \infty)$, we map it to a query $q' = [x_1, y_1] \times [x_2, y_2] \times \ldots [x_d, y_d]$ in $\mathbb{R}^d$. Then the query is executed in multiple rounds. In round $j$ (starting with $j = 1$), we query $\mathcal{D}$ with $(q', k = 2^{j-1} \cdot Q(n))$. Two cases arise:

1) If exactly $2^{j-1} \cdot Q(n)$ points are reported, then we scan the reported points to find the point with the smallest weight (say $w_i$). If $w_i < x_{d+1}$, then we do not go to the next round. Otherwise, we go to round $j+1$.
2) If less than $2^{j-1} \cdot Q(n)$ points are reported, then we do not go to the next round.

If the query does not go beyond round $j$, then among the points reported in round $j$, we remove each point whose weight is less than $x_{d+1}$. The remaining points are the output of the orthogonal range search query on $\mathcal{A}$ with $q$.

Now we analyze the time taken to answer the orthogonal range search query. There are two cases:

1) Only one round is executed: Then the query time will be $O(Q(n))$ since the value of $k = Q(n)$.
2) There are $i > 1$ rounds of execution: Then the query time will be $O(\sum_{j=1}^{i}(Q(n)+2^{j-1}Q(n))) = O(2^i \cdot Q(n))$. As in the proof of Theorem 4.1, we have $t \geq 2^{i-2}Q(n)$ which implies that $2^iQ(n) \leq 4t$. Therefore, the query time will be $O(2^i \cdot Q(n)) = O(t)$.

Therefore, the overall query time will be $O(Q(n) + t)$. □

*Theorem 4.3:* There exists a reduction from orthogonal point enclosure reporting in $\mathbb{R}^{d+1}$ to top-$k$ orthogonal point enclosure in $\mathbb{R}^d$.

*Proof:* Assume there is a data structure, $\mathcal{D}$, which can answer the top-$k$ orthogonal point enclosure in $\mathbb{R}^d$. Given $n$ hyper-rectangles, let $S(n)$ be the space occupied by the data structure and $Q(n) + O(k)$ be the time taken to answer the top-$k$ query.

We wish to answer an orthogonal point enclosure query on a set, $\mathcal{A}$, of $n$ hyper-rectangles in $\mathbb{R}^{d+1}$. Each hyper-rectangle in $\mathcal{A}$ is of the form $[x_1, y_1] \times [x_2, y_2] \times \ldots [x_d, y_d] \times [x_{d+1}, \infty)$. This can be done using a data structure that uses $O(S(n))$ space and has a query time of $Q(n) + O(t)$, where $t$ is the number of hyper-rectangles reported.

Each hyper-rectangle in $\mathcal{A}$ which is of the form $[x_1, y_1] \times \ldots [x_d, y_d] \times [x_{d+1}, \infty)$ is mapped to a new hyper-rectangle $[x_1, y_1] \times \ldots \times [x_d, y_d]$ in $\mathbb{R}^d$ and assigned a weight $-x_{d+1}$. Based on these newly mapped hyper-rectangles we build an instance of the data structure $\mathcal{D}$. Clearly the space occupied by $\mathcal{D}$ will be $O(S(n))$.

Given a query point $q = (q^1, \ldots, q^d, q^{d+1})$, we map it to $q' = (q^1, \ldots, q^d)$ in $\mathbb{R}^d$. Then the query is executed in multiple rounds. In round $j$, query $\mathcal{D}$ with $(q', k = 2^{j-1} \cdot Q(n))$. Two cases arise:

1) If exactly $2^{j-1} \cdot Q(n)$ hyper-rectangles are reported, then we scan the reported hyper-rectangles to find the hyper-rectangle with the smallest weight (say $w_i$). If $w_i < -q^{d+1}$, then we do not go to the next round. Otherwise, we go to round $j + 1$.
2) If less than $2^{j-1} \cdot Q(n)$ hyper-rectangles are reported, then we do not go to the next round.

If the query does not go beyond round $j$, then among the hyper-rectangles reported in round $j$, we remove each hyper-rectangle whose weight is less than $-q^{d+1}$. The remaining hyper-rectangles are the output of the orthogonal range search query on $\mathcal{A}$ with $q$. By repeating the analysis done for Theorem 4.2 the query time can be seen to be $O(Q(n) + t)$. □

# 5 Experimental results

We report on some experiments conducted with practical versions of the top-$k$ orthogonal range search and top-$k$ orthogonal point enclosure search algorithms discussed in Section 3. For the reasons given at the end of Section 1.2, our implementations use $R$-trees [6] instead of the asymptotically efficient counting and reporting structures described in Section 3. $R$-trees are well known in the database literature for answering orthogonal range search and orthogonal point enclosure search efficiently. We shall name our implementation of top-$k$ orthogonal range search and top-$k$ orthogonal point enclosure search *Top-k ORS* and *Top-k OPES*, respectively. In our implementation, the entire data structure resides in main memory, rather than on disk.

For the sake of comparison, for both the problems we also implemented the following two naive solutions:

1) First, we built a solution based on the idea discussed in Section 1. Build an $R$-tree on the objects in $\mathcal{A}$ (disregarding their weights). Given the query pair $(q, k)$, query the $R$-tree with $q$ to report all the objects of $\mathcal{A}$ which intersect $q$. Next, run a standard selection algorithm on $\mathcal{A}(q)$ to identify the object with the $k$th-largest weight. Finally, scan $\mathcal{A}(q)$ to output the $k$ largest-weight objects. For top-$k$ orthogonal range search and top-$k$ orthogonal point enclosure search, this naive solution is henceforth referred to as *Naive ORS* and *Naive OPES*, respectively. (The qualifier "Top-$k$" is omitted for brevity.)
2) Second, we built a solution which will be referred to as *Naive Scan*: In the pre-processing phase, the *Naive Scan* technique sorts all the objects in $\mathcal{A}$ in non-increasing order of their weights and keeps them in an array. Given a query pair $(q, k)$, the array is scanned from the beginning till either (i) $k$ objects of $\mathcal{A}$ intersecting $q$ are found, or (ii) the end of the array is reached.

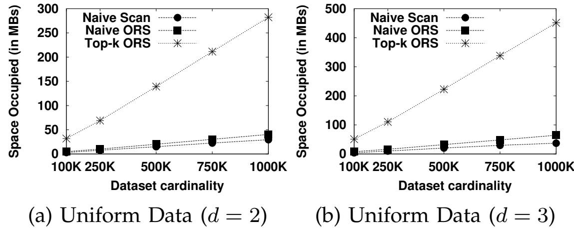
We make an additional remark here. An aggregate $R$-tree [33] is an augmented $R$-tree in which at every intermediate node we store the maximum weight among the objects stored in its subtree. An aggregate $R$-tree is used to solve the top-1 GIQ problem. However, it is not obvious how one can trivially modify this structure to handle top-$k$ GIQ, for any value of $k > 1$. The same issue also arises for $aP$-tree [18] which also answers the top-1 GIQ (for low dimensional data). Therefore, we do not compare our techniques with these spatial structures.

Our implementation was in C++ on a Celeron(R) dual-core (2.10GHz $\times 2$) Linux machine with 3GB RAM.

## 5.1 Top-$k$ orthogonal range search

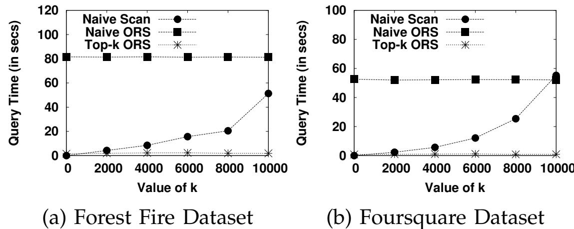
The following datasets were used to evaluate solutions for the top-$k$ orthogonal range search problem:

1) **Uniform data:** For each point, we generated each of its coordinates and its weight uniformly at random in the interval $[0, 10^6]$. For this dataset, we varied $n$ from $10^5$ to $10^6$ and varied $d$ from 2 to 5.
2) **Forest fire data:** We obtained a dataset for the state of California, which is represented by a $1200 \times 1200$ pixel map, where each pixel corresponds to a 1km $\times$ 1km region on the ground. The Computer Science department at the University of Minnesota has devised a technique for identifying forest fires, which gives a confidence value (a real number) for a forest fire at a pixel. Higher confidence values indicate higher likelihood of a forest fire having occurred at the pixel. Our goal was to use this information to identify, within an area of interest in the pixel




(a) Uniform Data ($d = 2$) (b) Uniform Data ($d = 3$)

Fig. 4: Size of structure versus dataset cardinality.




(a) Forest Fire Dataset (b) Foursquare Dataset

Fig. 5: Query time versus $k$. 'Query time' is the total time to execute 100 queries.

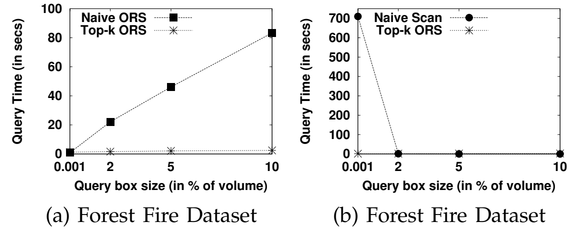
map, those pixels where the confidence levels of the technique were highest, hence where the likelihood of a fire having occurred was greatest. Towards this end, we treated the pixels as the data points, the lat-long information of each pixel as its coordinates, and the confidence value of the technique as the weight of each pixel. Thus, a top-$k$ orthogonal range search would yield the desired answer. For this dataset, we had $n = 1200 \times 1200 = 1.44 \times 10^6$ and $d = 2$.

3) **Foursquare data:** This is a real-world dataset obtained from Foursquare user histories [34]. It contains a database of users and the locations visited by these users. A user provides a rating to the locations s/he visits based on her/his experience at that location. We used this data to obtain an *average rating* for each location. The lat-long of each location was used as the coordinates and the average rating was the weight associated with each location. For this dataset, we had $n = 500,000$ and $d = 2$. This dataset has been previously used in [35], [36].

Unless mentioned otherwise, we used a query hyper-rectangle, $q$, that occupied 10% of the volume of the universe (The shape of $q$ was fixed and its location was varied.)

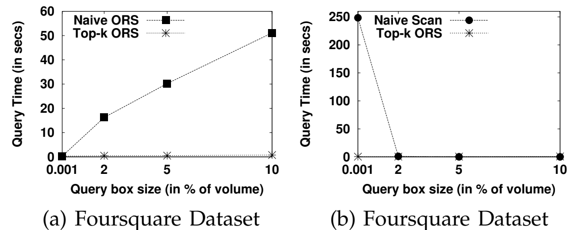
### 5.1.1 Comparison of space used

Here we discuss the space used by *Top-k ORS*, *Naive ORS* and *Naive Scan*. Figure 4 depicts the performance of the three solutions for different uniform datasets. With $R$-trees, the space used by *Top-k ORS* is $O(n \log n)$ rather than $O(n \log^d n)$, with the dependence on $d$ being multiplicative not exponential. This




(a) Forest Fire Dataset (b) Forest Fire Dataset

Fig. 6: Query time versus query box size. 'Query time' is the total time to execute 100 queries.



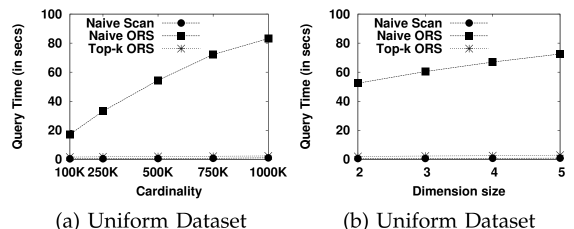

(a) Foursquare Dataset (b) Foursquare Dataset

Fig. 7: Query time versus query box size. 'Query time' is the total time to execute 100 queries.

manifests itself in the slow rate of growth of the space used as a function of $n$. *Naive ORS* and *Naive Scan* used $O(n)$ space and as expected, used less space than *Top-k ORS*. However, given the significant benefits realized by *Top-k ORS* in query efficiency (discussed in the following subsections), we believe that this space overhead is reasonable.

### 5.1.2 Effect of $k$

Figure 5 shows the query time of the three solutions as a function of $k$ on the forest fire and foursquare datasets. All the algorithms used the same set of 100 query boxes to query the dataset. We note that the query times for *Naive ORS* and *Top-k ORS* do not vary much with the value of $k$. In the former case this might be attributable to the fact that the number of points in the query range dominates the value of $k$, while in the latter case the time to taken to identify the threshold point and the canonical nodes in $\mathcal{T}'$ and to query the GIQ data structure at the canonical nodes




(a) Uniform Dataset (b) Uniform Dataset

Fig. 8: (a) Query time versus cardinality. (b) Query time versus dimension size. 'Query time' is the total time to execute 100 queries.

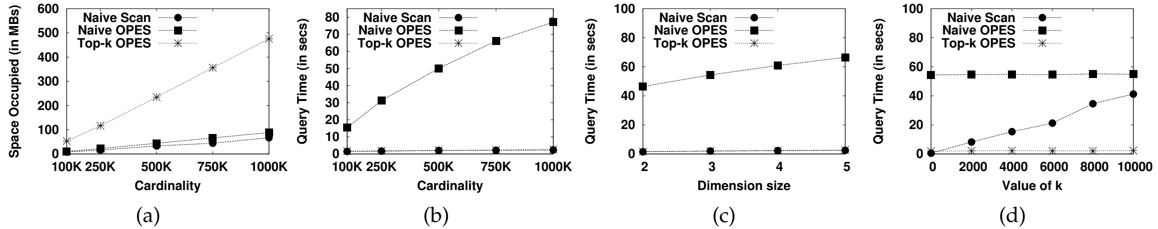



Fig. 9: Results for top-$k$ orthogonal point enclosure. 'Query time' is the total time to execute $100$ queries.

dominates the time to actually report the top-$k$ points. However, the query time for *Naive Scan* increased rapidly with the increase in the value of $k$. This is to be expected since the larger the value of $k$, the greater the number of points that *Naive Scan* has to scan.

#### 5.1.3 Effect of query box size

Figure 6 and 7 shows the query time of the three solutions as a function of the query box size on the forest fire and foursquare datasets, respectively, with $k = 5$. Each query box was a square whose center was generated uniformly at random in the interval $[0, 10^6]$ along each dimension. The performance of *Naive ORS* was directly proportional to the number of points inside the query box. As shown in Figures 6(a) and 7(a), when the query box size was very small, *Naive ORS* performed well but its query time increased drastically as the query box size increased. The query time of *Top-k ORS* also increased with query box size but at a much slower rate. On the other hand, the performance of *Naive ORS* was inversely proportional to the number of points inside the query box. As shown in Figures 6(b) and 7(b), when the query box size was very small, *Naive Scan* performed poorly (as it had to scan a large portion of the dataset) but its query time decreased dramatically as the query box size increased.

#### 5.1.4 Effect of cardinality and dimensionality

Finally we performed a set of experiments to test the effect of dataset cardinality and dimensionality on the query time of the two solutions. Figure 8(a) shows the query time versus cardinality, on the uniform dataset with $d = 2$ and $k = 5$. Figure 8(b) shows the query time versus dimension size, on the uniform dataset with $n = 500,000$ and $k = 5$.

As can be seen in Figure 8, the query time of *Top-k ORS* and *Naive Scan* increased with dataset cardinality or dimension, but at a rather slow rate. By comparison, the query time of *Naive ORS* grew at a much faster rate. We also observe that the query time of *Naive ORS* degraded more rapidly with increasing dataset cardinality than with increasing dimension. One possible reason is that the performance of *Naive ORS* is directly proportional to the number of points inside the query box, which increases with increase in cardinality; on the other hand since an $R$-tree is being used to retrieve all the points inside the query box efficiently, dataset dimension has less of an impact on query time.

The conclusion to be drawn from these experiments is that our implementation of *Top-k ORS* is quite practical and is robust to changes in dataset cardinality and dimension, as well as changes in the query parameters (location and size of $q$, value of $k$).

### 5.2 Top-$k$ orthogonal point enclosure search

We used synthetic datasets to compare the performance of *Top-k OPES*, *Naive OPES* and *Naive Scan*. Each input hyper-rectangle was created by generating the coordinates of its bottom-left and top-right corners uniformly and at random in $[0, 10^6]$ (taking care that the top-right coordinates were larger than their bottom-left counterparts). Unless mentioned otherwise, the default values of $d = 3$, $n = 500,000$ and $k = 5$ were used.

With $R$-trees, the space used by *Top-k OPES* is $O(n \log n)$. Figure 9 compares the performance of the two solutions against various parameters as in Section 5.1.3. Again, the implementation of *Top-k OPES* is seen to be quite efficient in practice.

## 6 Conclusion

We have given a general technique that solves any top-$k$ GIQ problem efficiently, provided efficient solutions are available for the counting and reporting versions of the underlying GIQ problem. We have shown how to use this to obtain asymptotically efficient solutions for several specific instances of top-$k$ GIQ problems (Table 1). We have also investigated the computational hardness of the top-$k$ GIQ problem. We have implemented our solution for some of the top-$k$ GIQ problems discussed, using practical data structures ($R$-trees), and have found them to be quite efficient.

Our results assume a computational model where all the data resides in internal memory. It is also possible to extend our results to the external memory model where the data resides primarily on disk, in blocks of some size $B$. A result analogous to Theorem 1.1 can be obtained except that space is measured

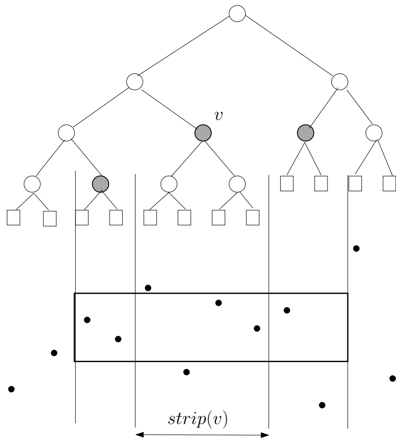



Fig. 10: Querying a two-dimensional range tree built on 12 points. The canonical nodes are shaded in grey.

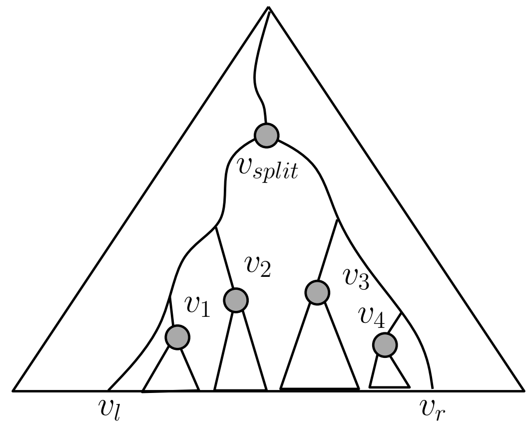



Fig. 11: Querying the priority search tree. $C_l = \{v_1, v_2\}$ and $C_r = \{v_3, v_4\}$.

in blocks, query and update performance are measured in terms of the number of input-output (I/O) operations, and the output-size term in the query time is of the form $k/B$.

# Appendix A
# Some Common Geometric Data Structures

In this appendix we give a brief introduction to some of the common geometric data structures which are used in computational geometry.

## A.1 Range Tree

A Range Tree is a popular data structure in computational geometry to answer orthogonal range reporting and counting queries. For the sake of simplicity, we shall discuss static range trees in $\mathbb{R}^2$ for a reporting query. We are given a set $S$ of $n$ points lying in $\mathbb{R}^2$. The primary structure is a height-balanced binary search tree $T$ in which the points of $S$ are stored in the leaves of $T$ in increasing order of their $x$-coordinate. For each node $v \in T$, define $S(v)$ to be the set of points of $S$ lying in the subtree of $v$. With each node $v \in T$, we associate an array $A(v)$ which stores the points of $S(v)$ in increasing order of their $y$-coordinate values. Also, define $strip(v)$ to be the range on the $x$-axis which contains all the points in $S(v)$, i.e., $strip(v) = [x_l, x_r]$, where $x_l$ (resp. $x_r$) is the $x$-coordinate of the leftmost (resp. rightmost) point in the subtree of $v$ (see Figure 10). Each point $p$ of $S$ is stored in the array associated with all the $O(\log n)$ nodes on the path from the root till the leaf containing $p$. Therefore, the space occupied by $T$ will be $O(n \log n)$.

Let $q = [x_1, x_2] \times [y_1, y_2]$ be the query rectangle. Given $q$, a node $v \in T$ is a *canonical node* if $[x_1, x_2] \subseteq strip(v)$ but $[x_1, x_2] \not\subseteq strip(p(v))$, where $p(v)$ is the parent of $v$. It can be shown that the size of the set, $C_q$, of canonical nodes is bounded by $O(\log n)$ (at most two per each level of $T$). Given any two nodes $u, v \in C_q$, $strip(u)$ and $strip(v)$ are disjoint; and the union of the strips of the nodes in $C_q$ exactly cover $[x_1, x_2]$ (see Figure 10). The crucial observation is the following: The points of $S$ inside $q$ can be reported by performing one-dimensional range reporting with $[y_1, y_2]$ on each of the arrays associated with the canonical nodes. Performing a one-dimensional range search on an array $A(v)$ takes $O(\log n + k_v)$ time, where $O(\log n)$ is the time taken to perform a binary search on the array and $k_v$ is the number of points reported. Therefore, the total query time will be $O(\sum_{v \in C_q}(\log n + k_v)) = O(\log^2 n + \sum_{v \in C_q} k_v) = O(\log^2 n + k)$, where $k = |S \cap q|$. By making suitable changes to the query algorithm, one can also answer the counting query in $O(\log^2 n)$ time. For further details, see the book by de Berg *et al.* [2].

## A.2 Priority Search Tree

A Priority Search Tree is used for range reporting when the query rectangle is restricted to be of the form $q = [x_1, x_2] \times [y_1, \infty)$. It is a combination of a heap (on $y$-coordinates) and a binary search tree (on $x$-coordinates). For a set $S$ of $n$ points lying in $\mathbb{R}^2$, the space occupied is $O(n)$ and the query time is $O(\log n + k)$, where $k = |S \cap q|$.

Pick the point in $S$ with the highest $y$-coordinate, say $p_{max}$ and place it at the root of the structure $T$. Let $p_{med}$ be the median point of $S$ based on the $x$-coordinate, which is also stored at the root node. Recursively build the left (resp. right) subtree of $T$ based on the points of $S$ whose $x$-coordinate is smaller (resp. larger) than that of $x_{med}$. Since each point of $S$ gets stored at exactly one node, the space occupied by $T$ is $O(n)$.

Given the restricted query rectangle $q$, find the leaf node $v_l$ (resp. $v_r$) of $T$ which is a successor (resp. predecessor) of $x_1$ (resp. $x_2$). Let $v_{split}$ be the least common ancestor node of $v_l$ and $v_r$ (see Figure 11).

Then the points of $S \cap q$ can be reported by the following steps:

1) Scan the points stored on the path from root till $v_l$ (resp. $v_r$) and report those which lie inside $q$. There will be only $O(\log n)$ points on these two paths.
2) Let $C_l$ (resp. $C_r$) be the set of nodes such that each is not on the path from $v_{split}$ till $v_l$ (resp. $v_r$) but whose parent is on the path. All the points in the subtree of any node in $C_l$ and $C_r$ have the property that their $x$-coordinate is in the range $[x_1, x_2]$, which implies that we only need to report points whose $y$-coordinate is greater than $y_1$ (see Figure 11). To do this, at every node $v \in C_l \cup C_r$, we check if the point stored at $v$ has $y$-coordinate greater than $y_1$. If the answer is no, then we stop; else we then recursively proceed to the subtrees of its two children. It can be shown that the time spent at each node $v \in C_l \cup C_r$ is $O(1 + k_v)$, where $k_v$ is the number of points reported in the subtree of $v$. The total time taken for all the nodes in $C_l \cup C_r$ will be $O(\sum_{v \in C_l \cup C_r}(1 + k_v)) = O(\log n + k)$.

The overall query time will be $O(\log n + k)$. For details on how to handle updates, see the survey paper by Chiang and Tamassia [27].

### A.3 Segment Tree

The segment tree [27] data structure is used for handling queries on geometric objects such as hyper-rectangles, orthogonal/arbitrary segments etc. A fundamental problem which can be solved using a segment tree is the following: Preprocess a set $S$ of $n$ intervals (possibly overlapping) on the real line into a data structure, so that given a query point $q$ on the real line, we can efficiently report or count the intervals of $S$ intersected by $q$. The space occupied by the segment tree will be $O(n \log n)$ and the query time is $O(\log n + k)$, where $k = |S \cap q|$.

Let $p_1, p_2, \ldots, p_m$ be the set of distinct endpoints of the intervals of $S$, sorted from left to right. The real line is partitioned into the following disjoint regions called *elementary intervals*: $(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_{m-1}, p_m)$, $[p_m, p_m], (p_m, \infty)$. A height balanced binary search structure $T$ is built. The leaves of $T$ store the elementary intervals in the order shown above. With each node $v \in T$, we store a variable $strip(v)$. If $v$ is a leaf node, then $strip(v)$ is the elementary interval corresponding to $v$; else it is the union of the elementary intervals stored in the leaf nodes of the subtree of $v$. A node stores an interval (say $I = (x_1, x_2)$) of $S$ if $strip(v) \subseteq I$ but $strip(p(v)) \nsubseteq I$, where $p(v)$ is the parent of $v$. All the intervals stored at node $v$ are maintained in a linked list $S_v$. It can be shown that each interval of $S$ can get stored at $O(\log n)$ nodes of $T$ (at most two nodes per level of $T$). Therefore, the size of the segment tree is $O(n \log n)$.

To answer a reporting query, we follow the path, $\Pi$, from the root of $T$ till a leaf node $u$ such that $q \in strip(u)$. At each node $v \in \Pi$, we report all the intervals stored in $S_v$. This will ensure that all the points in $S \cap q$ are reported. The query time can be shown to be $O(\log n + k)$. For more details see [2].

## Acknowledgment

The authors thank the three reviewers for their constructive comments which helped improve the paper.

## References


[1] P. Agarwal and J. Erickson, "Geometric range searching and its relatives," in *B. Chazelle, J. E. Goodman, and R. Pollack (Eds.)*, Advances in Discrete and Computational Geometry, *AMS Press*, vol. 23, 1999, pp. 1–56.

[2] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry*. Springer–Verlag, 2000.

[3] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[4] I. Ilyas, G. Beskales, and M. Soliman, "A survey of top-$k$ query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, no. 4, 2008.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2001.

[6] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. of ACM Management of Data (SIGMOD)*, 1984, pp. 47–57.

[7] G.Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz, "Online sorted range reporting," in *Proc. of International Symposium on Algorithms and Computation (ISAAC)*, 2009, pp. 173–182.

[8] C. Sheng and Y. Tao, "Dynamic top-$k$ range reporting in external memory," in *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, 2012, pp. 121–130.

[9] P. Afshani, G. Brodal, and N. Zeh, "Ordered and unordered top-$k$ range reporting in large data sets," in *Proc. of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2011, pp. 390–400.

[10] S. Rahul, P. Gupta, R. Janardan, and K. Rajan, "Efficient top-$k$ queries for orthogonal ranges," in *International Workshop on Algorithms and Computation (WALCOM)*, 2011, pp. 110–121.

[11] T. Gagie, S. Puglisi, and A. Turpin, "Range quantile queries: another virtue of wavelet trees," in *Proc. of the International Conference on String Processing and Information Retrieval (SPIRE)*, 2009, pp. 1–6.

[12] G. Navarro and L. Russo, "Space-efficient data-analysis queries on grids," in *Proc. of International Symposium on Algorithms and Computation (ISAAC)*, 2011, pp. 323–332.

[13] M. Karpinski and Y. Nekrich, "Top-$k$ color queries for document retrieval," in *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2011, pp. 401–411.

[14] T. Gagie, G. Navarro, and S. Puglisi, "Colored range reporting and document retrieval," in *Proc. of the International Conference on String Processing and Information Retrieval (SPIRE)*, 2010, pp. 67–81.

[15] A. Yu, P. Agarwal, and J. Yang, "Processing and notifying range top-$k$ subscriptions," in *Proc. of International Conference on Data Engineering (ICDE)*, 2012, pp. 810–821.

[16] Y. Tao, C. Sheng, C.-W. Chung, and J.-R. Lee, "Range aggregation with set selection," *To appear in IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2014.

[17] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," *The VLDB Journal*, vol. 12, no. 3, pp. 262–283, 2003.

[18] Y. Tao and D. Papadias, "Range aggregate processing in spatial databases," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, no. 12, pp. 1555–1570, 2004.

[19] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa, "Keyword search in spatial databases: Towards searching by document," in *Proc. of International Conference on Data Engineering (ICDE)*, 2009, pp. 688–699.
[20] S. Govindarajan, P. K. Agarwal, and L. Arge, "Crb-tree: An efficient indexing scheme for range-aggregate queries." in *Proc. of International Conference on Database Theory (ICDT)*, 2003, pp. 143–157.
[21] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient olap operations in spatial data warehouses," in *Proc. of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, 2001, pp. 443–459.
[22] C. Sheng and Y. Tao, "New results on two-dimensional orthogonal range aggregation in external memory," in *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, 2011, pp. 129–139.
[23] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.
[24] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant, "Range queries in olap data cubes," in *Proc. of ACM Management of Data (SIGMOD)*, 1997, pp. 73–88.
[25] C. K. Poon, "Optimal range max datacube for fixed dimensions," in *Proc. of International Conference on Database Theory (ICDT)*, 2003, pp. 158–172.
[26] D. Willard and G. Lueker, "Adding range restriction capability to dynamic data structures," *Journal of the ACM (JACM)*, vol. 32, pp. 597–617, 1982.
[27] Y.-J. Chiang and R. Tamassia, "Dynamic algorithms in computational geometry," *Proc. of the IEEE, Special Issue on Computational Geometry*, vol. 80, no. 9, pp. 1412–1434, 1992.
[28] P. Agarwal and J. Matousek, "Dynamic half-space range reporting and its applications," *Algorithmica*, vol. 13, no. 4, pp. 325–345, 1995.
[29] J. Matousek, "Geometric range searching," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 421–461, 1994.
[30] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*. Springer–Verlag, 1987.
[31] S.-W. Cheng and R. Janardan, "Efficient dynamic algorithms for some geometric intersection problems," *Information Processing Letters (IPL)*, vol. 36, no. 5, pp. 251–258, 1990.
[32] B. Chazelle, "A functional approach to data structures and its use in multidimensional searching," *SIAM Journal of Computing*, vol. 17, no. 3, pp. 427–462, 1988.
[33] M. Jürgens and H.-J. Lenz, "The $R_a^*$-tree: An improved R*-tree with materialized data for supporting range queries on OLAP-data," in *DEXA Workshop*, 1998, pp. 186–191.
[34] Foursquare:, https://foursquare.com.
[35] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel, "Lars: A location-aware recommender system," in *Proc. of International Conference on Data Engineering (ICDE)*, 2012, pp. 450–461.
[36] M. Sarwat, J. Bao, A. Eldawy, J. J. Levandoski, A. Magdy, and M. F. Mokbel, "Sindbad: a location-based social networking system," in *Proc. of ACM Management of Data (SIGMOD)*, 2012, pp. 649–652.

PLACE PHOTO HERE

**Saladi Rahul** is a third year Ph.D. candidate in the Department of Computer Science & Engineering at the University of Minnesota–Twin Cities. He received his Bachelor's degree and Master's degree in Computer Science from IIIT-Hyderabad, India. His research interests are computational geometry and algorithms for database systems.

PLACE PHOTO HERE

**Ravi Janardan** is a full professor in the Department of Computer Science & Engineering at the University of Minnesota–Twin Cities. He earned a Ph.D. degree in computer science in 1987 from Purdue University, West Lafayette, IN. His research interests are in the design and analysis of geometric algorithms and data structures, and their applications to spatial query retrieval, biomedicine, and computer-aided design and manufacture. He is a Senior Member of IEEE.