Theoretical Computer Science 609 (2016) 32–48

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Range queries on uncertain data ☆

Jian Li [a], Haitao Wang [b,*]

[a] Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing 100084, China
[b] Department of Computer Science, Utah State University, Logan, UT 84322, USA

ARTICLE INFO

Article history:
Received 7 January 2015
Received in revised form 7 July 2015
Accepted 6 September 2015
Available online 10 September 2015
Communicated by G.F. Italiano

Keywords:
Range queries
Uncertain data
Top-k queries
Threshold queries
Data structures
Algorithms

ABSTRACT

Given a set P of n uncertain points on the real line, each represented by its one-dimensional probability density function, we consider the problem of building data structures on P to answer range queries of the following three types for any query interval I: (1) top-1 query: find a point in P that lies in I with the highest probability, (2) top-k query: given any integer $k \leq n$ as part of the query, return the k points in P that lie in I with the highest probabilities, and (3) threshold query: given any threshold τ as part of the query, return all points of P that lie in I with probabilities at least τ. We present data structures for these range queries with linear or nearly linear space and efficient query time.

1. Introduction

With a rapid increase in the number of application domains, such as data integration, information extraction, sensor networks, scientific measurements, etc., where uncertain data are generated in an unprecedented speed, managing, analyzing and query processing over such data have become a major challenge and have received significant attentions. We study an important problem in this domain, building data structures for uncertain data for efficiently answering certain range queries. The problem has been studied extensively with a wide range of applications [3,14,28,33,36,43,44]. We formally introduce the problem below.

Let $\mathbb{R}$ be any real line (e.g., the x-axis). In the (traditional) deterministic version of this problem, we are given a set P of n deterministic points on $\mathbb{R}$, and the goal is to build a data structure (also called "index" in database) such that given a range, specified by an interval $I \subseteq \mathbb{R}$, one point (or all points) in I can be retrieved efficiently. It is well known that a simple solution for this problem is a binary search tree over all points which is of linear size and can support logarithmic (plus output size) query time. However, in many applications, the location of each point may be uncertain and the uncertainty is represented in the form of probability distributions [4,6,14,43,44]. In particular, an *uncertain point* p is specified by its probability density function (pdf) $f_p : \mathbb{R} \to \mathbb{R}^+ \cup \{0\}$. Let P be the set of n uncertain points in $\mathbb{R}$ (with pdfs specified as input). Our goal is to build data structures to quickly answer range queries on P. In this paper, we consider the following three types of range queries, each of which involves a query interval $I = [x_l, x_r]$. For any point $p \in P$, we use $\Pr[p \in I]$ to denote the probability that p is contained in I.

☆ A preliminary version of this paper appeared in the Proceedings of the 25th International Symposium on Algorithms and Computation (ISAAC 2014).

* Corresponding author.

E-mail addresses: lijian83@mail.tsinghua.edu.cn (J. Li), haitao.wang@usu.edu (H. Wang).

http://dx.doi.org/10.1016/j.tcs.2015.09.005

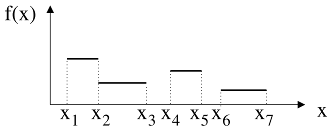

Fig. 1. The pdf of an uncertain point.

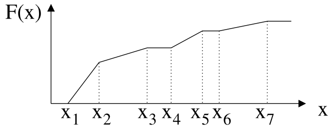

Fig. 2. The cdf of the uncertain point in Fig. 1.

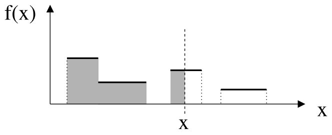

Fig. 3. Geometrically, $F_p(x)$ is equal to the sum of the areas of the shaded rectangles.

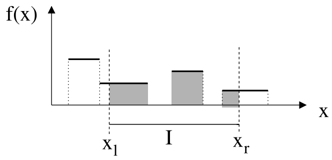

Fig. 4. Geometrically, the probability $\Pr[p \in I]$ is equal to the sum of the areas of the shaded rectangles.

Top-1 query: Return a point p of P such that $\Pr[p \in I]$ is the largest.
Top-k query: Given any integer k, $1 \le k \le n$, as part of the query, return the k points p of P such that $\Pr[p \in I]$ are the largest.
Threshold query: Given a threshold τ, as part of the query, return all points p of P such that $\Pr[p \in I] \ge \tau$.

Note that for each top-1 query, if P has more than one point p such that $\Pr[p \in I]$ is the largest, then an arbitrary such point is considered as a correct answer to the query. Similarly, we break ties arbitrarily for the top-k queries.

We assume f_p is a step function, i.e., a *histogram* consisting of at most c pieces (or intervals) for some integer $c \ge 1$ (e.g., see Fig. 1). More specifically, $f_p(x) = y_i$ for $x_{i-1} \le x < x_i$, $i = 1, \dots, c$, with $x_0 = -\infty$, $x_c = \infty$, and $y_1 = y_c = 0$. Throughout the paper, we assume c is a constant. The cumulative distribution function (cdf) $F_p(x) = \int_{-\infty}^{x} f_p(t)dt$ is a monotone piecewise-linear function consisting of c pieces (e.g., see Fig. 2). Note that $F_p(+\infty) = 1$, and for any interval $I = [x_l, x_r]$ the probability $\Pr[p \in I]$ is $F_p(x_r) - F_p(x_l)$. From a geometric point of view, each interval of f_p defines a rectangle with the x-axis, and the sum of the areas of all these rectangles of f_p is exactly one. Further, the cdf value $F_p(x)$ is the sum of the areas of the subsets of these rectangles to the left of the vertical line through x (e.g., see Fig. 3), and the probability $\Pr[p \in I]$ is the sum of the areas of the subsets of these rectangles between the two vertical lines through x_l and x_r, respectively (e.g., see Fig. 4).

As discussed in [3], the histogram model can be used to approximate most pdfs with arbitrary precision in practice. In addition, the *discrete* pdf where each uncertain point can appear in a few locations, each with a certain probability, can be viewed as a special case of the histogram model because we can use infinitesimal pieces around these locations.

We also study an important special case where the pdf f_p is a uniform distribution function, i.e., f is associated with an interval $[x_l(p), x_r(p)]$ such that $f_p(x) = 1/(x_r(p) - x_l(p))$ if $x \in [x_l(p), x_r(p)]$ and $f_p(x) = 0$ otherwise. Clearly, the cdf $F_p(x) = (x - x_l(p))/(x_r(p) - x_l(p))$ if $x \in [x_l(p), x_r(p)]$, $F_p(x) = 0$ if $x \in (-\infty, x_l(p))$, and $F_p(x) = 1$ if $x \in (x_r(p), +\infty)$. Uniform distributions have been used as a major representation of uncertainty in some previous work (e.g., [12,14,30]). We refer to this special case as the *uniform case* and the more general case where f_p is a histogram distribution function as the *histogram* case.

Throughout the paper, we will always use $I = [x_l, x_r]$ to denote the query interval. The query interval I is *unbounded* if either $x_l = -\infty$ or $x_r = +\infty$ (otherwise, I is *bounded*). For the threshold query, we will always use m to denote the output size of the query, i.e., the number of points p of P such that $\Pr[p \in I] \ge \tau$.

Range reporting on uncertain data has many applications [3,14,28,36,43,44], As shown in [3], our problems are also useful even in some applications that involve only deterministic data. For example, consider the movie rating system in IMDB where each reviewer gives a rating from 1 to 10. A top-k query on $I = [7, +\infty)$ would find "the k movies such that the percentages of the ratings they receive at least 7 are the largest"; a threshold query on $I = [7, +\infty)$ and $\tau = 0.85$ would find "all the movies such that at least 85% of the ratings they receive are larger than or equal to 7". Note that in the above examples the interval I is unbounded, and thus, it would also be interesting to have data structures particularly for quickly answering queries with unbounded query intervals.

1.1. *Previous work*

The threshold query was first introduced by Cheng et al. [14]. Using R-trees, they [14] gave heuristic algorithms for the histogram case, without any theoretical performance guarantees. For the uniform case, if τ is fixed for any query, they proposed a data structure of $O(n\tau^{-1})$ size with $O(\tau^{-1}\log n + m)$ query time [14]. These bounds depend on τ^{-1}, which can be arbitrarily large.

Agarwal et al. [3] made a significant theoretical step on solving the threshold queries for the histogram case: If the threshold τ is fixed, their approach can build an $O(n)$ size data structure in $O(n\log n)$ time, with $O(m + \log n)$ query time; if τ is not fixed, they built an $O(n\log^2 n)$ size data structure in $O(n\log^3 n)$ expected time that can answer each query in $O(m + \log^3 n)$ time. Tao et al. [43,44] considered the threshold queries in two and higher dimensions. They provided heuristic results and a query takes $O(n)$ time in the worst case. Heuristic solutions were also given elsewhere, e.g. [28,36,40]. Recently, Abdullah et al. [1] extended the notion of *geometric coresets* to uncertain data for range queries in order to obtain efficient approximate solutions.

Our work falls into the broad area of managing and analyzing uncertain data which has attracted significant attentions recently in database community. This line of work has spanned a range of issues from theoretical foundation of data models and data languages, algorithmic problems for efficiently answering various queries, to system implementation issues. Probabilistic database systems have emerged as a major platform for this purpose and several prototype systems have been built to address different aspects/challenges in managing probabilistic data, e.g. MYSTIQ [18], Trio [6], ORION [13], MayBMS [29], PrDB [39], MCDB [25]. Besides of the range queries we mentioned above, there has also been much work on efficiently answering different types of queries over probabilistic data, such as conjunctive queries (or the union of conjunctive queries) [18,19], aggregates [26,38], top-k and ranking [16,30,31,37,41], clustering [17,24], nearest neighbors [7,12,35], and so on. We refer interested readers to the recent book [42] for more information.

As discussed in [3], our uncertain model is an analogue of the *attribute-level uncertainty model* in the probabilistic database literature. Another popular model is the *tuple-level uncertainty model* [6,18,45], where a tuple has fixed attribute values but its existence is uncertain. The range query under the latter model is much easier since a d-dimensional range searching over uncertain data can be transformed to a $(d+1)$-dimensional range searching problem over certain data [3,45]. In contrast, the problem under the former model is more challenging, partly because it is unclear how to transform it to an instance on certain data.

1.2. *Our results*

Based on our above discussion, the problem has four variations: the *uniform unbounded* case where each pdf f_p is a uniform distribution function and each query interval I is unbounded, the *uniform bounded* case where each pdf f_p is a uniform distribution function and each query interval I is bounded, the *histogram unbounded* case where each pdf f_p is a general histogram distribution function and each query interval I is unbounded, and the *uniform bounded* case where each pdf f_p is a general histogram distribution function and each query interval I is bounded. Refer to Table 1 for a summary of our results on the four cases.

Note that we also present solutions to the most general case (i.e., the histogram bounded case), which were originally left as open problems in the preliminary version of this paper [32].

We say the *complexity* of a data structure is $O(A, B)$ if it can be built in $O(A)$ time and its size is $O(B)$.

1. For the uniform unbounded case, the complexities of our data structures for the three types of queries are all $O(n\log n, n)$. The top-1 query time is $O(\log n)$; the top-k query time is $O(\log n + k)$; the threshold query time is $O(\log n + m)$.
2. For the histogram unbounded case, our results are the same as the above uniform unbounded case except that the time for each top-k query is $O(k)$ if $k = \Omega(\log n \log\log n)$ and $O(\log n + k\log k)$ otherwise (i.e., for large k, the algorithm has a better performance).
3. For the uniform bounded case, the complexity of our top-1 data structure is $O(n\log n, n)$, with query time $O(\log n)$. For the other two types of queries, the complexities of our data structures are both $O(n\log^2 n, n\log n)$; the top-k query time is $O(k)$ if $k = \Omega(\log n \log\log n)$ and $O(\log n + k\log k)$ otherwise, and the threshold query time is $O(\log n + m)$.
4. For the histogram bounded case, for threshold queries, Agarwal et al. [3] built a data structure of size $O(n\log^2 n)$ in $O(n\log^3 n)$ expected time, with $O(\log^3 n + m)$ query time. Note that our results on the threshold queries for the two uniform cases and the histogram unbounded case are clearly better than the above solution in [3]. For top-1 queries, we build a data structure of $O(n\log^2 n)$ size in $O(n\log^3 n)$ (deterministic) time, with $O(\log^3 n)$ query time. For top-k queries, we build a data structure of $O(n\log^2 n)$ size in $O(n\log^3 n)$ expected time, with $O(\log^3 n + k)$ query time.

Table 1
Summary of our results (the result for threshold queries of the histogram bounded case is from [3]): T is $O(k)$ if $k = \Omega(\log n \log\log n)$ and $O(\log n + k \log k)$ otherwise. For threshold queries, m is the output size of each query. All time complexities are deterministic except the preprocessing times for top-k and threshold queries of the histogram bounded case (marked with *).

Four problem variations			Top-1 queries	Top-k queries	Threshold queries
Unbounded	Uniform	Preprocessing time	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
		Space	$O(n)$	$O(n)$	$O(n)$
		Query time	$O(\log n)$	$O(\log n + k)$	$O(\log n + m)$
	Histogram	Preprocessing time	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
		Space	$O(n)$	$O(n)$	$O(n)$
		Query time	$O(\log n)$	T	$O(\log n + m)$
Bounded	Uniform	Preprocessing time	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$
		Space	$O(n)$	$O(n \log n)$	$O(n \log n)$
		Query time	$O(\log n)$	T	$O(\log n + m)$
	Histogram	Preprocessing time	$O(n \log^3 n)$	$O(n \log^3 n)$*	$O(n \log^3 n)$* [3]
		Space	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n \log^2 n)$ [3]
		Query time	$O(\log^3 n)$	$O(\log^3 n + k)$	$O(\log^3 n + m)$ [3]

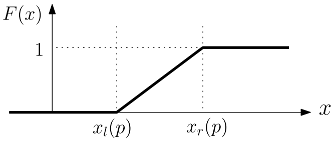

Fig. 5. Illustrating the cdf F_p (the three thick segments).

Note that all above results are based on the assumption that c is a constant; otherwise these results still hold with replacing n by $c \cdot n$ except that for the histogram bounded case the results hold with replacing n by $c^2 n$.

The rest of the paper is organized as follows. We first introduce the notation and some observations in Section 2. We present our results for the uniform case in Section 3. The histogram case is discussed in Section 4. Section 5 concludes the paper.

2. Preliminaries

Recall that an *uncertain point* p is specified by its pdf $f_p : \mathbb{R} \to \mathbb{R}^+ \cup \{0\}$ and the corresponding cdf is $F_p(x) = \int_{-\infty}^{x} f_p(t)dt$ is a monotone piecewise-linear function (with at most c pieces). For each uncertain point p, we call $\Pr[p \in I]$ the I-*probability* of p. Let $\mathcal{F}$ be the set of the cdfs of all points of P. Since each cdf is an increasing piecewise linear function, depending on the context, $\mathcal{F}$ may also refer to the set of the $O(n)$ line segments of all cdfs. Recall that $I = [x_l, x_r]$ is the query interval. We start with an easy observation.

Lemma 1. *If $x_l = -\infty$, then for any uncertain point p, $\Pr[p \in I] = F_p(x_r)$.*

Proof. Due to $x_l = -\infty$, $\Pr[p \in I] = \int_{-\infty}^{x_r} f_p(t)dt$, which is exactly $F_p(x_r)$. □

Let L be the vertical line with x-coordinate x_r. Since each cdf F_p is a monotonically increasing function, there is only one intersection between F_p and L. It is easy to know that for each cdf F_p of $\mathcal{F}$, the y-coordinate of the intersection of F_p and L is $F_p(x_r)$, which is the I-probability of p by Lemma 1. For each point in any cdf of $\mathcal{F}$, we call its y-coordinate the *height* of the point.

In the uniform case, each cdf F_p has three segments (e.g., see Fig. 5): the leftmost one is a horizontal segment with two endpoints $(-\infty, 0)$ and $(x_l(p), 0)$, the middle one, whose slope is $1/(x_r(p) - x_l(p))$, has two endpoints $(x_l(p), 0)$ and $(x_r(p), 1)$, and the rightmost one is a horizontal segment with two endpoints $(x_r(p), 1)$ and $(+\infty, 1)$. We transform each F_p to the line l_p containing the middle segment of F_p. Consider an unbounded interval I with $x_l = -\infty$. We can use l_p to compute $\Pr[p \in I]$ in the following way. Suppose the height of the intersection of L and l_p is y. Then, $\Pr[p \in I] = 0$ if $y < 0$, $\Pr[p \in I] = y$ if $0 \le y \le 1$, $\Pr[p \in I] = 1$ if $y > 1$. Therefore, once we know $l_p \cap L$, we can obtain $\Pr[p \in I]$ in constant time. Hence, we can use l_p instead of F_p to determine the I-probability of p. The advantage of using l_p is that lines are usually easier to deal with than line segments. Below, with a little abuse of notation, for the uniform case we simply use F_p to denote the line l_p for any $p \in P$ and now $\mathcal{F}$ is a set of lines.

Fix the query interval $I = [x_l, x_r]$. For each i, $1 \le i \le n$, denote by p_i the point of P whose I-probability is the i-th largest. Based on the above discussion, we obtain Lemma 2, which holds for both the histogram and uniform cases.

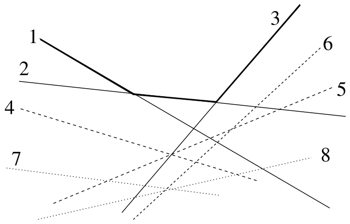

Fig. 6. Partitioning S into three layers: $L_1(S) = \{1, 2, 3\}$, $L_2(S) = \{4, 5, 6\}$, $L_3(S) = \{7, 8\}$. The thick polygonal chain is the upper envelope of S.

Lemma 2. *If $x_l = -\infty$, then for each $1 \le i \le n$, p_i is the point of P such that $L \cap F_{p_i}$ is the i-th highest among the intersections of L and all cdfs of $\mathcal{F}$.* □

Suppose $x_l = -\infty$. Based on Lemma 2, to answer the top-1 query on I, it is sufficient to find the cdf of $\mathcal{F}$ whose intersection with L is the highest; to answer the top-k query, it is sufficient to find the k cdfs of $\mathcal{F}$ whose intersections with L are the highest; to answer the threshold query on I and τ, it is sufficient to find the cdfs of $\mathcal{F}$ whose intersections with L have y-coordinates $\ge \tau$.

Half-plane range reporting: As the half-plane range reporting data structure [11] is important for our later developments, we briefly discuss it in the dual setting. Let S be a set of n lines. Given any point q, the goal is to report all lines of S that are above q. An $O(n)$-size data structure can be built in $O(n \log n)$ time that can answer each query in $O(\log n + m')$ time, where m' is the number of lines above the query point q [11]. The data structure can be built as follows.

Let U_S be the upper envelope of S (e.g., see Fig. 6). We represent U_S as an array of lines $l_1, l_2, \ldots, l_h$ ordered as they appear on U_S from left to right. For each line l_i, l_{i-1} is its *left neighbor* and l_{i+1} is its *right neighbor*. We partition S into a sequence $L_1(S), L_2(S), \ldots,$ of subsets, called *layers* (e.g., see Fig. 6). The first layer $L_1(S) \subseteq S$ consists of the lines that appear on U_S. For $i > 1$, $L_i(S)$ consists of the lines that appear on the upper envelope of the lines in $S \setminus \bigcup_{j=1}^{i-1} L_j(S)$. Each layer $L_i(S)$ is represented in the same way as U_S. To answer a half-plane range reporting query on a point q, let $l(q)$ be the vertical line through q. We first determine the line l_i of $L_1(S)$ whose intersection with $l(q)$ is on the upper envelope of $L_1(S)$, by doing binary search on the array of lines of $L_1(S)$. Then, starting from l_i, we walk on the upper envelope of $L_1(S)$ in both directions to report the lines of $L_1(S)$ above the point q, in linear time with respect to the output size. Next, we find the line of $L_2(S)$ whose intersection with $l(q)$ is on the upper envelope of $L_2(S)$. We use the same procedure as for $L_1(S)$ to report the lines of $L_2(S)$ above q. Similarly, we continue on the layers $L_3(S), L_4(S), \ldots,$ until no line is reported in a certain layer. By using fractional cascading [9], after determining the line l_i of $L_1(S)$ in $O(\log n)$ time by binary search, the data structure [11] can report all lines above q in constant time each.

For any vertical line l, for each layer $L_i(S)$, denote by $l_i(l)$ the line of $L_i(S)$ whose intersection with l is on the upper envelope of $L_i(S)$. By fractional cascading [9], we have the following lemma for the data structure [11].

Lemma 3. *(See [9,11].) For any vertical line l, after the line $l_1(l)$ is known, we can obtain the lines $l_2(l), l_3(l), \ldots$ in this order in $O(1)$ time each.* □

3. The uniform distribution

In this section, we present our results for the uniform case. We first discuss our data structures for the unbounded case in Section 3.1, which will also be needed in our data structures for the bounded case in Section 3.2. Further, the results in Section 3.1 will also be useful in our data structures for the histogram case in Section 4.

Recall that in the uniform case $\mathcal{F}$ is a set of lines.

3.1. Queries with unbounded intervals

We first discuss the unbounded case where $I = [x_l, x_r]$ is unbounded and some techniques introduced here will also be used later for the bounded case. Without loss of generality, we assume $x_l = -\infty$, and the other case where $x_r = +\infty$ can be solved similarly. Recall that L is the vertical line with x-coordinate x_r.

For top-1 queries, by Lemma 2, we only need to maintain the upper envelope of $\mathcal{F}$, which can be computed in $O(n \log n)$ time and $O(n)$ space. For each query, it is sufficient to determine the intersection of L with the upper envelope of $\mathcal{F}$, which can be done in $O(\log n)$ time.

Next, we consider top-k queries.

Given I and k, by Lemma 2, it suffices to find the k lines of $\mathcal{F}$ whose intersections with L are the highest, and we let $\mathcal{F}_k$ denote the set of the above k lines. As preprocessing, we build the half-plane range reporting data structure (see Section 2) on $\mathcal{F}$, in $O(n \log n)$ time and $O(n)$ space. Suppose the layers of $\mathcal{F}$ are $L_1(\mathcal{F}), L_2(\mathcal{F}), \ldots.$ In the sequel, we compute the set $\mathcal{F}_k$. Let the lines in $\mathcal{F}_k$ be $l^1, l^2, \ldots, l^k$ ordered from top to bottom by their intersections with L.

Let $l_i(L)$ be the line of $L_i(\mathcal{F})$ which intersects L on the upper envelope of the layer $L_i(\mathcal{F})$, for $i = 1, 2, \ldots$. We first compute $l_1(L)$ in $O(\log n)$ time by binary search on the upper envelope of $L_1(\mathcal{F})$. Clearly, l^1 is $l_1(L)$. Next, we determine l^2. Let the set H consist of the following three lines: $l_2(L)$, the left neighbor (if any) of $l_1(L)$ in $L_1(\mathcal{F})$, and the right neighbor (if any) of $l_1(L)$ in $L_1(\mathcal{F})$.

Lemma 4. *l^2 is the line in H whose intersection with L is the highest.*

Proof. Note that l^2 is the line of $\mathcal{F} \setminus \{l^1\}$ whose intersection with L is the highest. We distinguish two cases:

1. If l^2 is in $L_1(\mathcal{F})$, since the slopes of the lines of $L_1(\mathcal{F})$ from left to right are increasing, l^2 must be a neighbor of l^1. Hence, l^2 must be either the left neighbor or the right neighbor of l^1 in $L_1(\mathcal{F})$.
2. If l^2 is not in $L_1(\mathcal{F})$, then $l_2(L)$ must be the line of $\mathcal{F} \setminus L_1(\mathcal{F})$ whose intersection with L is the highest. According to the definition of the layers of $\mathcal{F}$ the upper envelope of $L_2(\mathcal{F})$ is also the upper envelope of $\mathcal{F} \setminus L_1(\mathcal{F})$. Therefore, $l_2(L)$ is the line of $\mathcal{F} \setminus L_1(\mathcal{F})$ whose intersection with L is the highest. Hence, l^2 must be $l_2(\mathcal{F})$.

The lemma thus follows. □

We refer to H as the *candidate set*. By Lemma 4, we find l^2 in H in $O(1)$ time. We remove l^2 from H. In the following, we insert at most three lines into H such that l^3 must be in H. Specifically, if l^2 is $l_2(L)$, we insert the following three lines into H: $l_3(L)$, the left neighbor of $l_2(L)$, and the right neighbor of $l_2(L)$. If l^2 is the left (resp., right) neighbor l of $l_1(L)$, we insert the left (resp., right) neighbor of l in $L_1(\mathcal{F})$ into H. By generalizing Lemma 4, we can show l^3 must be in H (the details are omitted). We repeat the same algorithm until we find l^k. To facilitate the implementation, we use a heap to store the lines of H whose "keys" in the heap are the heights of the intersections of L and the lines of H.

Lemma 5. *The set $\mathcal{F}_k$ can be found in $O(\log n + k \log k)$ time.*

Proof. According to our algorithm, there are $O(k)$ insertions and "extract-max" operations (i.e., finding the element of H with the largest key and remove the element from H) on the heap H. The size of H is always bounded by $O(k)$ during the algorithm. Hence all operations on H take $O(k \log k)$ time. Further, after finding $l_1(L)$ in $O(\log n)$ time, due to Lemma 3, the lines that are inserted into H can be found in constant time each. Hence, the total time for finding $\mathcal{F}_k$ is $O(k \log k + \log n)$. □

We can improve the algorithm to $O(\log n + k)$ time by using the selection algorithm in [23] for sorted arrays. The key idea is that we can implicitly obtain $2k$ sorted arrays of $O(k)$ size each and $\mathcal{F}_k$ can be computed by finding the largest k elements in these arrays. The details are given in Lemma 6.

Lemma 6. *The set $\mathcal{F}_k$ can be found in $O(\log n + k)$ time.*

Proof. Consider any layer $L_i(\mathcal{F})$. Suppose the array of lines of $L_i(\mathcal{F})$ is $l_1, l_2, \ldots, l_h$ and let l_j be the line $l_i(L)$. The intersections of the lines $l_j, l_{j+1}, \ldots, l_h$ with L are sorted in decreasing order of their heights, and the intersections of the lines $l_{j-1}, l_{j-2}, \ldots, l_1$ with L are also sorted in decreasing order of their heights. Once l_j is known, we can implicitly obtain the following two arrays A_i^r and A_i^l: the t-th element of A_i^r (resp., A_i^l) is the height of the intersection of l_{t-j+1} (resp., l_{j-t}) and L. Since these lines are explicitly maintained in the layer $L_i(\mathcal{F})$, given any index t, we can obtain the t-th element of A_i^r (resp., A_i^l) in $O(1)$ time.

To compute the set $\mathcal{F}_k$, we first find the lines $l_i(\mathcal{F})$ for $i = 1, 2, \ldots, k$, which can be done in $O(\log n)$ time due to Lemma 3. Consequently, we obtain the $2k$ arrays A_i^r and A_i^l for $1 \le i \le k$, implicitly. In fact we only need to consider the first k elements of each such array, and below we let A_i^r and A_i^l denote the arrays only consisting of the first k elements. An easy observation is that the heights of the intersections of L and the lines of $\mathcal{F}_k$ are exactly the largest k elements of $\mathcal{A} = \bigcup_{i=1}^{k} \{A_i^r \cup A_i^l\}$.

In light of the above discussion, to compute $\mathcal{F}_k$, we do the following: (1) find the k-th largest element τ of $\mathcal{A}$; (2) find the lines of $\mathcal{A}$ whose intersections with L have heights at least τ, which can be done in $O(k)$ time by checking the above $2k$ sorted arrays with τ in their index orders. Below, we show that we can compute τ in $O(k)$ time.

Recall that $\mathcal{A}$ contains $2k$ sorted arrays and each array has k elements. Further, for any array, given any index t, we can obtain its t-th element in constant time. Hence, we can find the k-th largest element of $\mathcal{A}$ in $O(k)$ time by using the selection algorithm given in [23] for matrices with sorted columns (each sorted array in our problem can be viewed as a sorted column of a $k \times 2k$ matrix).

The lemma thus follows. □

Hence, we obtain the following result.

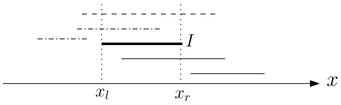

Fig. 7. Illustrating the three types of intervals: the thick segment is I; the two thin solid segments are of L-type; the two dashed dotted segments are of R-type; the dashed segment is of M-type.

Theorem 1. *For the uniform case, we can build in $O(n \log n)$ time an $O(n)$ size data structure on P that can answer each top-k query with an unbounded query interval in $O(k + \log n)$ time.*

For the threshold query, we are given I and a threshold τ. We again build the half-plane range reporting data structure on $\mathcal{F}$. To answer the query, as discussed in Section 2, we only need to find all lines of $\mathcal{F}$ whose intersections with L have y-coordinates larger than or equal to τ. We first determine the line $l_1(L)$ by doing binary search on the upper envelope of $L_1(\mathcal{F})$. Then, by Lemma 3, we find all lines $l_2(L), l_3(L), \dots, l_j(L)$ whose intersections have y-coordinates larger than or equal to τ. For each i with $1 \le i \le j$, we walk on the upper envelope of $L_i(\mathcal{F})$, starting from $l_i(L)$, on both directions in time linear in the output size to find the lines whose intersections have y-coordinates larger than or equal to τ. Hence, the running time for answering the query is $O(\log n + m)$.

3.2. Queries with bounded intervals

Now we assume $I = [x_l, x_r]$ is bounded. Consider any point $p \in P$. Recall that p is associated with an interval $[x_l(p), x_r(p)]$ in the uniform case. Depending on the positions of $I = [x_l, x_r]$ and $[x_l(p), x_r(p)]$, we classify $[x_l(p), x_r(p)]$ and the point p into the following three types with respect to I (e.g. see Fig. 7).

L-type: $[x_l(p), x_r(p)]$ and p are L-type if $x_l \le x_l(p)$.
R-type: $[x_l(p), x_r(p)]$ and p are R-type if $x_r \ge x_r(p)$.
M-type: $[x_l(p), x_r(p)]$ and p are M-type if $I \subset (x_l(p), x_r(p))$.

Denote by P_L, P_R, and P_M the sets of all L-type, R-type, and M-type of points of P, respectively. In the following, for each kind of query, we will build an data structure such that the different types of points will be searched separately (note that we will not explicitly compute the three subsets P_L, P_R, and P_M). For each point $p \in P$, we refer to $x_l(p)$ as the *left endpoint* of the interval $[x_l(p), x_r(p)]$ and refer to $x_r(p)$ as the *right endpoint*. For simplicity of discussion, we assume that no two interval endpoints of the points of P have the same value.

3.2.1. Top-1 queries

For any point $p \in P$, denote by $\mathcal{F}_r(p)$ the set of the cdfs of the points of P whose intervals have left endpoints larger than or equal to $x_l(p)$. Again, as discussed in Section 2 we transform each cdf of $\mathcal{F}_r(p)$ to a line. We aim to maintain the upper envelope of $\mathcal{F}_r(p)$ for each $p \in P$. If we computed the n upper envelopes explicitly, we would have an data structure of size $\Omega(n^2)$. To reduce the space, we choose to use the persistent data structure [21] to maintain them implicitly such that data structure size is $O(n)$. The details are given below.

We sort the points of P by the left endpoints of their intervals from left to right, and let the sorted list be $p'_1, p'_2, \dots, p'_n$. For each i with $2 \le i \le n$, observe that the set $\mathcal{F}_r(p'_{i-1})$ has exactly one more line than $\mathcal{F}_r(p'_i)$. If we maintain the upper envelope of $\mathcal{F}_r(p'_i)$ by a balanced binary search tree (e.g., a red-black tree), then by updating it we can obtain the upper envelope of $\mathcal{F}_r(p'_{i-1})$ by an insertion and a number of deletions on the tree, and each tree operation takes $O(\log n)$ time. An easy observation is that there are $O(n)$ tree operations in total to compute the upper envelopes of all sets $\mathcal{F}_r(p'_1), \mathcal{F}_r(p'_2), \dots, \mathcal{F}_r(p'_n)$. Further, by making the red-black tree persistent [21], we can maintain all upper envelopes in $O(n \log n)$ time and $O(n)$ space. We use $\mathcal{L}$ to denote the above data structure.

We can use $\mathcal{L}$ to find the point of P_L with the largest I-probability in $O(\log n)$ time, as follows. First, we find the point p'_i such that $x_l(p'_{i-1}) < x_l \le x_l(p'_i)$. It is easy to see that $\mathcal{F}_r(p'_i) = P_L$. Consider the unbounded interval $I' = (-\infty, x_r]$. Consider any point p whose cdf is in $\mathcal{F}_r(p'_i)$. Due to $x_l(p) \ge x_l$, we can obtain that $\Pr[p \in I] = \Pr[p \in I']$. Hence, the point p of $\mathcal{F}_r(p'_i)$ with the largest value $\Pr[p \in I]$ also has the largest value $\Pr[p \in I']$. This implies that we can instead use the unbounded interval I' as the query interval on the upper envelope of $\mathcal{F}_r(p'_i)$, in the same way as in Section 3.1. The persistent data structure $\mathcal{L}$ maintains the upper envelope of $\mathcal{F}_r(p'_i)$ such that we can find in $O(\log n)$ time the point p of $\mathcal{F}_r(p'_i)$ with the largest value $\Pr[p \in I']$.

Similarly, we can build a data structure $\mathcal{R}$ of $O(n)$ space in $O(n \log n)$ time that can find the point of P_R with the largest I-probability in $O(\log n)$ time.

To find the point of P_M with the largest I-probability, the approach for P_L and P_R does not work because we cannot reduce the query to another query with an unbounded interval. Instead, we reduce the problem to a "segment dragging query" by dragging a line segment out of a corner in the plane, as follows.

For each point p of P, we define a point $q = (x_l(p), x_r(p))$ in the plane, and we say that p *corresponds to* q. Similar transformation was also used in [14]. Let Q be the set of the n points defined by the points of P. For the query interval

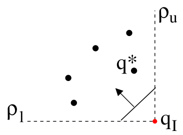

Fig. 8. Dragging a segment of slope 1 out of the corner at q_I: q^* is the first point that will be hit by the segment.

$I = [x_l, x_r]$, we also define a point $q_I = (x_l, x_r)$ (this is different from [14], where I defines a point (x_r, x_l)). If we partition the plane into four quadrants with respect to q_I, then we have the following lemma.

Lemma 7. *The points of P_M correspond to the points of Q that strictly lie in the second quadrant (i.e., the northwest quadrant) of q_I.*

Proof. Consider any point $p \in P$. Let $q = (x_l(p), x_r(p))$ be the point defined by p. On the one hand, p is in P_M if and only if $I \subset (x_l(p), x_r(p))$, i.e., $x_l > x_l(p)$ and $x_r < x_r(p)$. On the other hand, $x_l > x_l(p)$ and $x_r < x_r(p)$ if and only if q is in the second quarter of $q_I = (x_l, x_r)$. The lemma thus follows. □

Let ρ_u be the upwards ray originating from q_I and let ρ_l be the leftwards ray originating from q_I. Imagine that starting from the point q_I and towards northwest, we drag a segment of slope 1 with two endpoints on ρ_u and ρ_l respectively, and let q^* be the point of Q hit first by the segment (e.g., see Fig. 8).

Lemma 8. *The point of P that defines q^* is in P_M and has the largest I-probability among all points in P_M.*

Proof. First of all, by Lemma 7, q^* must be in P_M.

Consider any point q in the second quadrant of q_I, and let p be the point of P that defines q. Since the interval of p contains the interval I, we have $\Pr[p \in I] = \frac{x_r - x_l}{x_r(p) - x_l(p)}$.

Based on the definition of q^*, q^* is the point q of Q in the second quadrant of q_I that has the smallest value $x_r(p) - x_l(p)$. Therefore, q^* is the point q of Q in the second quadrant of q_I that has the largest value $\frac{x_r - x_l}{x_r(p) - x_l(p)}$. The lemma thus follows. □

Based on Lemma 8, to determine the point of P_M with the largest I-probability, we only need to solve the above query on Q by dragging a segment out of a corner. More specifically, we need to build a data structure on Q to answer the following *out-of-corner segment-dragging queries*: Given a point q, find the first point of Q hit by dragging a segment of slope 1 from q and towards the northwest direction with the two endpoints on the two rays $\rho_u(q)$ and $\rho_l(q)$, respectively, where $\rho_u(q)$ is the upwards ray originating from q and $\rho_l(q)$ is the leftwards ray originating from q. By using Mitchell's result in [34] (reducing the problem to a point location problem), we can build an $O(n)$ size data structure on Q in $O(n \log n)$ time that can answer each such query in $O(\log n)$ time.

Theorem 2. *For the uniform case, we can build in $O(n \log n)$ time an $O(n)$ size data structure on P that can answer each top-1 query in $O(\log n)$ time.* □

3.2.2. Top-k queries

To answer a top-k query, we will do the following. First, we find the top-k points in P_L (i.e., the k points of P_L whose I-probabilities are the largest), the top-k points in P_R, and the top-k points in P_M. Then, we find the top-k points of P from the above $3k$ points. Below we build three data structures for computing the top-k points in P_L, P_R, and P_M, respectively.

We first build the data structure for P_L. Again, let $p'_1, p'_2, \ldots, p'_n$ be the list of the points of P sorted by the left endpoints of their intervals from left to right. We construct a complete binary search tree T_L whose leaves from left to right store the n intervals of the points $p'_1, p'_2, \ldots, p'_n$. For each internal node v, let P_v denote the set of points whose intervals are stored in the leaves of the subtree rooted at v. We build the half-plane range reporting data structure discussed in Section 2 on P_v, denoted by D_v. Since the size of D_v is $|P_v|$, the total size of the data structure T_L is $O(n \log n)$, and T_L can be built in $O(n \log^2 n)$ time.

We use T_L to compute the top-k points in P_L as follows. By the standard approach and using x_l, we find in $O(\log n)$ time a set V of $O(\log n)$ nodes of T_L such that $P_L = \bigcup_{v \in V} P_v$ and no node of V is an ancestor of another node. Then, we can determine the top-k points of P_L in similarly as in Section 3.1. However, since we now have $O(\log n)$ data structures D_v, we need to maintain the candidate sets for all such D_v's. Specifically, after we find the top-1 point in D_v for each $v \in V$, we use a heap H to maintain them where the "keys" are the I-probabilities of the points. Let p be the point of H with the largest key. Clearly, p is the top-1 point of P_L; assume p is from D_v for some $v \in V$. We remove p from H and insert at most three new points from D_v into H, in a similar way as in Section 3.1. We repeat the same procedure until we find all top-k points of P_L.

To analyze the running time, for each node $v \in V$, we can determine in $O(\log n)$ time the line in the first layer of D_v whose intersection with L is on the upper envelope of the first layer, and subsequent operations on D_v each takes $O(1)$

time due to fractional cascading. Hence, the total time for this step in the entire algorithm is $O(\log^2 n)$. However, we can do better by building a fractional cascading structure [9] on the first layers of D_v for all nodes v of the tree T_L. In this way, the above step only takes $O(\log n)$ time in the entire algorithm, i.e., do binary search only at the root of T_L. In addition, building the heap H initially takes $O(\log n)$ time. Note that the additional fractional cascading structure on T_L does not change the size and construction time of T_L asymptotically [9]. The entire query algorithm has $O(k)$ operations on H in total and the size of H is $O(\log n + k)$. Hence, the total time for finding the top-k points of P_L is $O(\log n + k\log(k + \log n))$, which is $O(\log n + k\log k)$ by Lemma 9.

Lemma 9. $\log n + k\log(k + \log n) = O(\log n + k\log k)$.

Proof. To simplify the notation, let $n' = \log n$, and our goal is to prove $k\log(k+n') + n' = O(k\log k + n')$. Depending on whether $k \geq \frac{n'}{\log n'}$, there are two cases.

1. If $k \geq \frac{n'}{\log n'}$, then $\log k \geq \log n' - \log\log n'$, implying that $\log\log n' = O(\log k)$. Thus,

$$k\log(k+n') \leq k\log(k + k\log n') = O(k\log(k\log n')) = O(k(\log k + \log\log n')) = O(k\log k).$$

   Hence, we obtain that $k\log(k+n') + n' = O(k\log k + n')$.
2. If $k < \frac{n'}{\log n'}$, then $k\log(k+n') \leq \frac{n'}{\log n'}\log(\frac{n'}{\log n'} + n') = O(\frac{n'}{\log n'}\log n') = O(n')$.
   Hence, we obtain that $k\log(k+n') + n' = O(k\log k + n')$.

The lemma thus follows. □

If $k = \Omega(\log n \log\log n)$, we have a better result in Lemma 10. Note that comparing with Lemma 6, we need to use other techniques to obtain Lemma 10 since the problem here involves $O(\log n)$ half-plane range reporting data structures D_v while Lemma 6 only needs to deal with one such data structure.

Lemma 10. *If $k = \Omega(\log n \log\log n)$, we can compute the top-k points in P_L in $O(k)$ time.*

Proof. We assume $k = \Omega(\log n \log\log n)$. Recall that L is the vertical line with x-coordinate x_r. Let V be the set of $O(\log n)$ nodes of T_L as defined before. Consider any node $v \in V$, which is associated with a half-plane range reporting data structure D_v on the cdfs of the point set P_v. Let $\mathcal{F}(v)$ be the cdfs of the points of P_v. Let $\mathcal{F}(V) = \cup_{v\in V}\mathcal{F}(v)$. Our goal is to find the k lines of $\mathcal{F}(V)$ whose intersections with L are the highest, and denote by $\mathcal{F}_k$ the above k lines that we seek.

We let $L_1(v), L_2(v), \ldots$ be the layers of $\mathcal{F}(v)$, and for each layer $L_i(v)$, denote by $l_i(v)$ the line of $L_i(v)$ whose intersection with L is on the upper envelope of the layer $L_i(v)$. For each layer $L_i(v)$, we define two arrays $A_i^r(v)$ and $A_i^l(v)$ in the same way as in the proof of Lemma 6 (we omit the details). For each node v, we define another array $B(v)$ of size k as follows: for each $1 \leq i \leq k$, the i-th element of $B(v)$ is the height of the intersection of $l_i(v)$ and L. Hence, the elements of $B(v)$ are sorted in decreasing order.

Our algorithm for computing $\mathcal{F}_k$ has two main steps. In the first main step, we will find a set B' of the largest k elements in $B(V) = \cup_{v\in V}B(v)$. For each $v \in V$, let j_v be the number of elements of $B(v)$ that are contained in B', i.e., the first j_v elements of $B(v)$ are in B'. An easy observation is that the heights of the intersections of L and the sought lines of $\mathcal{F}_k$ are the largest k elements of $\mathcal{A} = \bigcup_{v\in V}\cup_{i=1}^{j_v}\{A_i^r(v) \cup A_i^l(v)\}$, which contains $2k$ sorted arrays. In the second main step, we will find the largest k elements of $\mathcal{A}$ and thus obtain the set $\mathcal{F}_k$. Below, we show that the above two main steps can be done in $O(k)$ time.

We consider the first main step. For simplicity of discussion, we assume no two elements of $B(V)$ are equal. First of all, as discussed above, in $O(\log n)$ time we can determine the lines $l_1(v)$ for all $v \in V$, and thus the first elements of all arrays $B(v)$ for $v \in V$ are determined. Further, for each array $B(v)$, due to the fractional cascading, we can obtain the next element in constant time each, and in other words, we can obtain the first i elements of $B(v)$ in $O(i)$ time. To compute the set B', i.e., the largest k elements of $B(V)$, since $B(V)$ contains $|V|$ sorted arrays, one may want to use the selection algorithm in [23] again, as in Lemma 6. However, here we cannot use that algorithm because for each array in $B(V)$, given any data structure, we cannot obtain the corresponding element in constant time. Instead, we propose the following approach.

Recall that $k = \Omega(\log n \log\log n)$. For simplicity of discussion, we assume $|V| = \log n$ and $k \geq \log n \log\log n$. Let $h = \log\log n$. Let H be a max-heap. Initially $H = \emptyset$. During the algorithm, we will maintain a set S of elements. Initially $S = \emptyset$, and after the algorithm stops, S contains at most $k + h$ elements.

First of all, for each array $B(v)$, we compute its first h elements and then insert only the h-th element of $B(v)$ into H. Now H contains $|V| = \log n$ elements. We do an “extract-max” operation on H, i.e., remove the largest element from H. Suppose the element removed above is from $B(v)$ for a node v. Then, we add the first h elements of $B(v)$ into S. If $|S| \geq k$, the algorithm stops; otherwise, we compute the next h elements of $B(v)$ and insert only the $(2h)$-th element of $B(v)$ into H.

In general, suppose we do an "extract-max" operation on H and let the removed element by the operation be the $(i \cdot h)$-th element of the array $B(v)$ for a node v. Then, we add the elements of $B(v)$ with indices from $i \cdot (h-1)+1$ to $i \cdot h$ to S. If $|S| \geq k$, the algorithm stops; otherwise, we compute the next h elements of $B(v)$ and insert the $[i \cdot (h+1)]$-th element of $B(v)$ into H.

After the algorithm stops, we do the following. Consider any element in the current heap H, and suppose it is the $(j \cdot h)$-th element of the array $B(v)$ for a node v. Then, according to our algorithm, the elements of $B(v)$ with indices from $j \cdot (h-1)+1$ to $j \cdot h$ are not in S, and we call these h elements the *red elements*. Since the current H has at most $|V|-1$ elements, there are at most $h \cdot (|V|-1)$ red elements, and we let S' be the union of S and all red elements.

We claim that the largest k elements of $B(V)$ must be in S', i.e., $B' \subseteq S'$. We prove the claim as follows. Let a be the element that is removed from H in the last extract-max operation on H in the above algorithm, i.e., after a is removed, the algorithm stops. Let H be the heap after the algorithm stops. According to our algorithm, since each array $B(v)$ is sorted decreasingly, a is the smallest element in S. Since $|S| \geq k$ and $|B'| = k$, any element of B' must be larger than a. If $B' \subseteq S$, then the claim is proved. Otherwise, suppose an element b is in $B' \setminus S$. Since $b > a$ and all elements of H are smaller than a, b must be a red element, and thus b is in S'. The claim is proved.

In light of the above claim, B' can be easily obtained after we find the k-th largest element of S'.

In the sequel, we analyze the running time of the above algorithm for the first main step. Recall that $h = \log\log n$. First of all, the size of the heap H is at most $|V| = \log n$ at any time during the algorithm. Clearly, the algorithm will stop after $\lceil \frac{k}{h} \rceil$ extract-max operations. The number of insertions is at most $\lceil \frac{k}{h} \rceil + |V|$. Therefore, the running time of all operations on H in the entire algorithm is $O((\lceil \frac{k}{h} \rceil + \log n)\log\log n)$, which is $O(k)$ due to $k = \Omega(\log n \log\log n)$. On the other hand, the number of red elements is at most $h \cdot (\log n - 1)$, and thus $|S'| \leq h \cdot (\log n - 1) + |S| \leq h \cdot (\log n - 1) + k + h$, which is $O(k)$ due to $k = \Omega(\log n \log\log n)$. Notice that the elements of $B(V)$ that have been computed during the entire algorithm are exactly those in S', and thus the time for computing these elements is $O(|S'|) = O(k)$. Finally, since $|S'| = O(k)$, we can find the k-th largest element in S' in $O(k)$ time by using the well-known linear time selection algorithm.

As a summary, the first main step can find the set B' in $O(k)$ time.

The second main step is to compute the largest k elements in $\mathcal{A}$, which contains $2k$ arrays. As in the proof of Lemma 6, we can obtain any arbitrary element of these arrays in constant time, without computing these arrays explicitly. Hence, by using the same approach as in Lemma 6, we can compute the largest k elements of $\mathcal{A}$ in $O(k)$ time. Consequently, the set $\mathcal{F}_k$ can be obtained.

The lemma thus follows. □

To compute the top-k points of P_R, we build a similar data structure T_R, in a symmetric way as T_L, and we omit the details.

Finally, to compute the top-k points in P_M, we do the following transformation. For each point $p \in P$, we define a point $q = (x_l(p), x_r(p), 1/(x_r(p) - x_l(p)))$ in the 3-D space with x-, y-, and z-axes. Let Q be the set of all points in the 3-D space thus defined. Let the query interval I define an unbounded query box (or 3D rectangle) $B_I = (-\infty, x_l) \times (x_r, +\infty) \times (-\infty, +\infty)$. Similar to Lemma 7 in Section 3.1, the points of P_M correspond exactly to the points of $Q \cap B_I$. Further, the top-k points of P_M correspond to the k points of $Q \cap B_I$ whose z-coordinates are the largest. Denote by Q_I the k points of $Q \cap B_I$ whose z-coordinates are the largest. Below we build a data structure on Q for computing the set Q_I for any query interval I and thus finding the top-k points of P_M.

We build a complete binary search tree T_M whose leaves from left to right store all points of Q ordered by the increasing x-coordinate. For each internal node v of T_M, we build an auxiliary data structure D_v as follows. Let Q_v be the set of the points of Q stored in the leaves of the subtree of T_M rooted at v. Suppose all points of Q_v have x-coordinates less than x_l. Let Q'_v be the points of Q_v whose y-coordinates are larger than x_r. The purpose of the auxiliary data structure D_v is to report the points of Q'_v in the decreasing z-coordinate order in constant time each after the point of q_v is found, where q_v is the point of Q'_v with the largest z-coordinate. To achieve this goal, we use the data structure given by Chazelle and Guibas [10] (the one for Subproblem P1 in Section 5), and the data structure is a *hive graph* [8], which can be viewed as the preliminary version of the fractional cascading techniques [9]. By using the result in [10], we can build such a data structure D_v of size $O(|Q_v|)$ in $O(|Q_v| \log |Q_v|)$ time that can first compute q_v in $O(\log |Q_v|)$ time and then report other points of Q'_v in the decreasing z-coordinate order in constant time each. Since the size of D_v is $|Q_v|$, the size of the tree T_M is $O(n \log n)$, and T_M can be built in $O(n \log^2 n)$ time.

Using T_M, we find the set Q_I as follows. We first determine the set V of $O(\log n)$ nodes of T_M such that $\bigcup_{v \in V} Q_v$ consists of all points of Q whose x-coordinates less than x_l and no point of V is an ancestor of another point of V. Then, for each node $v \in V$, by using D_v, we find q_v, i.e., the point of Q_v with the largest z-coordinate, and insert q_v into a heap H, where the key of each point is its z-coordinate. We find the point in H with the largest key and remove it from H; denote the above point by q'_1. Clearly, q'_1 is the point of Q_I with the largest z-coordinate. Suppose q'_1 is in a node $v \in V$. We proceed on D_v to find the point of Q_v with the second largest z-coordinate and insert it into H. Now the point of H with the largest key is the point of Q_I with the second largest z-coordinate. We repeat the above procedure until we find all k points of Q_I.

To analyze the query time, finding the set V takes $O(\log n)$ time. For each node $v \in V$, the search for q_v on D_v takes $O(\log n)$ time plus the time linear in the number of points of D_v in Q_I. Hence, the total time for searching q_v for all vertices $v \in V$ is $O(\log^2 n)$ time. Similarly as before, we can remove a logarithmic factor by building a fractional cascading structure on the nodes of T_M for searching such points q_v's, in exactly the same way as in [8]. With the help of the fractional cascading structure, all these q_v's for $v \in V$ can be found in $O(\log n)$ time. Note that building the fractional cascading structure does not change the construction time and the size of T_M asymptotically [8]. In addition, building the heap H initially takes $O(\log n)$ time. In the entire algorithm there are $O(k)$ operations on H in total and the size of H is always bounded by $O(k + \log n)$. Therefore, the running time of the query algorithm is $O(\log n + k \log(k + \log n))$, which is $O(\log n + k \log k)$ by Lemma 9.

Using similar techniques as in Lemma 10, we obtain the following result.

Lemma 11. *If $k = \Omega(\log n \log\log n)$, we can compute the top-k points in P_M in $O(k)$ time.*

Proof. Consider any point v in V, which is associated with a set Q_v and a data structure D_v. Define an array $B(v)$ of size k as follows: for each $1 \le i \le k$, the i-th element of $B(v)$ is the i-th largest z-coordinate of the points of Q_v. As discussed above, in $O(\log n)$ time we can obtain the first elements of $B(v)$ for all $v \in V$, and after that, we can obtain the next element of each array $B(v)$ in constant time each by using the data structure D_v. Our goal is to find the point set Q_I.

Let $B(V) = \cup_{v \in V} B(v)$. An easy observation is that the z-coordinates of the points of Q_I are exactly the largest k elements in $B(V)$. Since $k = \Omega(\log n \log\log n)$, computing the largest k elements of $B(V)$ can be done in $O(k)$ time in the same way as the first main step of the algorithm in the proof of Lemma 10, and we omit the details.

The lemma is thus proved. □

We summarize our results for the top-k queries below.

Theorem 3. *For the uniform case, we can build in $O(n \log^2 n)$ time an $O(n \log n)$ size data structure on P that can answer each top-k query in $O(k)$ time if $k = \Omega(\log n \log\log n)$ and $O(k \log k + \log n)$ time otherwise.* □

3.2.3. Threshold queries

To answer the threshold queries, we build the same data structure as in Theorem 3, i.e., the three trees T_L, T_M, and T_R. The tree T_L is used for finding the points p of P_L with $\Pr[p \in I] \ge \tau$; T_R is for finding the points p of P_R with $\Pr[p \in I] \ge \tau$; T_M is for finding the points p of P_M with $\Pr[p \in I] \ge \tau$. The three trees T_L, T_R, and T_M are exactly the same as those for Theorem 3. We can compute them in $O(n \log^2 n)$ time and $O(n \log n)$ space.

Below, we discuss the query algorithms on the three trees. Let m_L, m_R, and m_M be the numbers of points in P_L, P_R, and P_M whose I-probabilities are at least τ, respectively. Hence, $m = m_L + m_R + m_M$.

To find the points p of P_L with $\Pr[p \in I] \ge \tau$, we first determine the set V of $O(\log n)$ nodes of T_L such that $\bigcup_{v \in V} P_v = P_L$ and no node of V is an ancestor of another node of V. Recall that each node v of T_L is associated with a half-plane range reporting data structure D_v. For each node $v \in V$, by using D_v, we can find the points p of P_v with $\Pr[p \in I] \ge \tau$ in $O(\log n + m_v)$ time, where m_v is the output size. Note that P_v and P_u are disjoint for any two nodes v and u of V. Hence, $\sum_{v \in V} m_v = m_L$. As there are $O(\log n)$ nodes in V, it takes $O(\log^2 n + m_L)$ time to find all points p of P_L with $\Pr[p \in I] \ge \tau$, and again the $O(\log^2 n)$ time factor can be reduced to $O(\log n)$ by using fractional cascading [9]. Hence, the total query time is $O(\log n + m_L)$.

We can use the similar approach to find all points p of P_R with $\Pr[p \in I] \ge \tau$ in $O(\log n + m_R)$ time, by using T_R. We omit the details.

Finally, we find the points p of P_M with $\Pr[p \in I] \ge \tau$, by using T_M. As in Section 3.2.2, for each point $p \in P$, we define in the 3-D space a point $(x_l(p), x_r(p), 1/(x_r(p) - x_l(p)))$. Let Q be the set of all n points defined above. Let the interval $I = [x_l, x_r]$ and τ together define an unbounded 3-D box query $B_I = (-\infty, x_l) \times (x_r, +\infty) \times [\tau, +\infty)$. Let $Q_I = Q \cap B_I$. Hence, the points p of P_M with $\Pr[p \in I] \ge \tau$ correspond to the points of Q_I, and thus $m_M = |Q_I|$.

By using the tree T_M, we can find Q_I in $O(\log n + m_M)$ time, as follows. We first determine the set V of $O(\log n)$ nodes of T_M such that $\bigcup_{v \in V} Q_v$ consists of all points of Q whose x-coordinates are less than x_l and no node of V is an ancestor of another node of V. Consider any node $v \in V$. Let Q'_v be the points of Q_v whose y-coordinates are larger than x_r, and q_v be the point Q'_v with the largest z-coordinate. Recall that after q_v is found D_v can report other points of Q'_v in the decreasing z-coordinate order in constant time each. Hence after q_v is known we can report the points of Q_v in the query box B_I in time linear in the output size. Again, with the help of fractional cascading, the nodes q_v for all $v \in V$ can be found in $O(\log n)$ time. Therefore, we can find all points of Q_I in $O(\log n + m_M)$ time. In other words, with T_M, we can find the points of P_M whose I-probabilities at least τ in $O(\log n + m_M)$ time.

In summary, we have Theorem 4.

Theorem 4. *For the uniform case, we can build in $O(n \log^2 n)$ time an $O(n \log n)$ size data structure that can answer each threshold query in $O(m + \log n)$ time, where m is the output size of the query.*

4. The histogram distribution

In this section, we present our data structures for the histogram case. In the histogram case, the cdf of each point $p \in P$ has c pieces; recall that we assumed c is a constant, and thus $\mathcal{F}$ is still a set of $O(n)$ line segments.

We first discuss our data structures for the unbounded case in Section 4.1 and then present our results for the bounded case in Section 4.2.

4.1. Queries with unbounded intervals

Again, we assume w.l.o.g. that $x_l = -\infty$. Recall that L is the vertical line with x-coordinate x_r. Note that Lemmas 1 and 2 are still applicable.

4.1.1. Top-1 queries

For the top-1 queries, as in Section 3.1 it is sufficient to maintain the upper envelope of $\mathcal{F}$. Although $\mathcal{F}$ now is a set of line segments, its upper envelope is still of size $O(n)$ and can be computed in $O(n \log n)$ time [5]. Given the query interval I, we can compute in $O(\log n)$ time the cdf of $\mathcal{F}$ whose intersection with L is on the upper envelope of $\mathcal{F}$.

Theorem 5. *In the histogram case, we can build in $O(n \log n)$ time an $O(n)$ size data structure on P that can answer each top-1 query with an unbounded query interval in $O(\log n)$ time.*

4.1.2. Threshold queries

For the threshold query, as discussed in Section 2 we only need to find the cdfs of $\mathcal{F}$ whose intersections with L have y-coordinates at least τ. Let q_I be the point (x_r, τ) on L. A line segment is *vertically above* q_I if the segment intersects L and the intersection is at least as high as q_I. Hence, to answer the threshold query on I, it is sufficient to find the segments of $\mathcal{F}$ that are vertically above q_I. Agarwal et al. [3] gave the following result on the *segment-below-point queries*. For a set S of $O(n)$ line segments in the plane, a data structure of $O(n)$ size can be computed in $O(n \log n)$ time that can report the segments of S vertically *below* a query point q in $O(m' + \log n)$ time, where m' is the output size. In our problem, we need a data structure on $\mathcal{F}$ to solve the *segments-above-point queries*, which can be solved by using the same approach as [3]. Therefore, we can build in $O(n \log n)$ time an $O(n)$ data structure on P that can answer each threshold query with an unbounded query interval in $O(m + \log n)$ time.

Theorem 6. *In the histogram case, we can build in $O(n \log n)$ time an $O(n)$ size data structure on P that can answer each threshold query with an unbounded query interval in $O(m + \log n)$ time, where m is the output size of the query.*

4.1.3. Top-k queries

For the top-k queries, we only need to find the k segments of $\mathcal{F}$ whose intersections with L are the highest. To this end, we can slightly modify the data structure for the segment-below-point queries given in [3].

The data structure in [3] is a binary tree structure that maintains a number of sets of lines (each such line contains a segment of $\mathcal{F}$). For each such set of lines, a half-plane range reporting data structure similar to that in Section 2 is built, where the lower envelopes (instead of the upper envelopes as we discussed in Section 2) of the layers of the lines are maintained. For our purpose, we replace it by our half-plane range reporting data structure in Section 2 (i.e., maintain the upper envelopes). With this modification, we can answer the segments-above-point queries in the following way.

Consider a query point $q = (x_r, -\infty)$ (i.e., the lower infinite endpoint of L), and suppose we want to find the segments of $\mathcal{F}$ vertically above q, which are also the segments intersecting L. By using the data structure [3] modified as above, the query algorithm works as follows. First, with the help of fractional cascading, in $O(\log n)$ time, the query algorithm will find $O(\log n)$ half-plane range reporting data structures such that for each such data structure D the segment intersecting L on the upper envelope of D is known. Second, for each such half-plane range reporting data structure D, from the above known segment intersecting L, by using the fractional cascading and walking on the upper envelopes of the layers of D, we can report all lines of D higher than q in constant time each. The first step takes $O(\log n)$ time, and the second step takes $O(m')$ time, where m' is the total output size.

For our problem, we only need to report the highest k segments of $\mathcal{F}$ that are vertically above q. To this end, we will modify the query algorithm such that the segments of $\mathcal{F}$ vertically above q will be reported in order from top to bottom, and once k segments are reported, we will terminate the algorithm. We use a heap H in a similar way as in Section 3.1 for the uniform case. Specifically, in the first step, we find the $O(\log n)$ half-plane range reporting data structures, and for each such data structure H, the highest segment of D intersecting L is known. In the second step, we build a heap H on these $O(\log n)$ segments where the keys are the y-coordinates of their intersections with L. The segment in H with the largest key must be the highest segment of $\mathcal{F}$ intersecting L. We remove the segment from H, and let D be the half-plane range reporting data structure that contains the segment. As in Section 3.1 for the uniform case, we determine in constant time at most three segments from D and insert them to H. Now the segment of H with the largest key is the second highest segment of $\mathcal{F}$ intersecting L. We repeat the above procedure until we have reported k segments.

To analyze the running time, the first step takes $O(\log n)$ time. In the second step, we have $O(k)$ operations on H and the segments that are inserted to H can be found in constant time each by the range-reporting data structures. The size of the heap H in the entire query algorithm is $O(k + \log n)$. Hence, the running time of the query algorithm is $O(k \log(k + \log n) + \log n)$, which is $O(k \log k + \log n)$ by Lemma 9.

Similarly, if $k = \Omega(\log n \log\log n)$, we can answer the top-$k$ query in $O(k)$ time, as follows. As discussed above, in $O(\log n)$ time we find the $O(\log n)$ half-plane range reporting data structures, and for each such data structure D, the highest segment of D intersecting L is known. The answer to the top-k query is the highest k intersections of L and the lines in these $O(\log n)$ half-plane range reporting data structures. This is exactly the same situation as in Lemma 10, where we also have $O(\log n)$ half-plane range reporting data structures. Hence, the algorithm in Lemma 10 is applicable here, which runs in $O(k)$ time.

In summary, we obtain the following results.

Theorem 7. *We can build in $O(n \log n)$ time an $O(n)$ size data structure on P that can answer each top-k query with an unbounded query interval in $O(k)$ time if $k = \Omega(\log n \log\log n)$ and $O(k \log k + \log n)$ time otherwise.*

4.2. Queries with bounded intervals

In this case, the query interval $I = [x_l, x_r]$ is bounded.

For this case, Agarwal et al. [3] built a data structure of size $O(n \log^2 n)$ in $O(n \log^3 n)$ expected time, which can answer each threshold query in $O(\log^3 n + m)$ time. We first briefly discuss this data structure (refer to Section 4 of [3] for more details) because our data structures for top-1 and top-k queries also use some of their techniques.

Agarwal et al. [3] built a data structure (a binary search tree), denoted by $\mathcal{T}$, which maintains a family of *canonical sets* of planes in 3D (defined by the uncertain points of P). Consider any query interval $I = [x_l, x_r]$ with a threshold value τ. Let $q(I)$ be the point with coordinates (x_l, x_r, τ) in 3D, and let $L(I)$ be the line through q and parallel to the z-axis. Using $\mathcal{T}$, one can determine a family $F(I)$ of $O(\log^2 n)$ canonical sets of $\mathcal{T}$ with the following property: Each uncertain point p defines one and only one plane in $F(I)$ such that the z-coordinate of the intersection of the plane with $L(I)$ is the probability $\Pr[p \in I]$. Note that the canonical sets of $F(I)$ are pairwise disjoint.

To answer the threshold query on I and τ, it is sufficient to report the planes in each canonical set of $F(I)$ that lie above the point $q(I)$. To this end, for each canonical set S of $\mathcal{T}$, Agarwal et al. [3] constructed a halfspace range-reporting data structure given by Afshani and Chan [2] on the planes in S in $O(|S|)$ space and $O(|S| \log |S|)$ expected time, such that given any point q, one can report the planes of S above q in $O(\log |S| + M)$ time, where M is the output size. In this way, because there are $O(\log^2 n)$ canonical sets in $F(I)$, the threshold query can be answered in $O(\log^3 n + m)$ time. The total space of $\mathcal{T}$ including the halfspace range-reporting data structures is $O(n \log^2 n)$ and $\mathcal{T}$ can be built in $O(n \log^3 n)$ expected time.

4.2.1. Top-1 queries

Consider the top-1 query on the above query interval I. To answer the query, it suffices to find the plane in $F(I)$ whose intersection with $L(I)$ is the highest. To this end, it is sufficient to know the intersection of $L(I)$ with the upper envelope of each canonical set of $F(I)$. Therefore, for each canonical set S of $\mathcal{T}$, instead of constructing a halfspace range-reporting data structure, we compute the upper envelope of the planes of S [20] and build a point location data structure [22,27] on the upper envelope, which can be done in $O(|S|)$ space and $O(|S| \log |S|)$ time. In this way, for each canonical set S of $F(I)$, in $O(\log n)$ time we can determine the plane intersecting $L(I)$ in the upper envelope of S. Hence, the top-1 query can be answered in $O(\log^3 n)$ time since $F(I)$ has $O(\log^2 n)$ canonical sets.

Comparing with the original data structure in [3], since we spend $O(|S|)$ space and $O(|S| \log |S|)$ time on each canonical set S of $\mathcal{T}$, the entire data structure can be constructed in $O(n \log^2 n)$ space and $O(n \log^3 n)$ (deterministic) time. We summarize the result for the top-1 queries in the follow theorem.

Theorem 8. *In the histogram case, we can build in $O(n \log^3 n)$ time an $O(n \log^2 n)$ size data structure on P that can answer each top-1 query with a bounded query interval in $O(\log^3 n)$ time.*

4.2.2. Top-k queries

Consider the top-k query on the query interval I. To answer the query, it suffices to find the k planes in $F(I)$ whose intersections with $L(I)$ are the highest. To this end, for each canonical set S of $\mathcal{T}$, we build a *t-highest plane* data structure given by Afshani and Chan [2] on the planes of S in $O(|S|)$ space and $O(|S| \log |S|)$ expected time, such that given any integer t and any query line L parallel to the z-axis, the t highest planes of S at L can be found in $O(\log |S| + t)$ time. Comparing with the original data structure in [3], since we spend asymptotically the same space and time on each canonical set of $\mathcal{T}$, our data structure can be constructed in $O(n \log^2 n)$ space and $O(n \log^3 n)$ expected time.

To answer the top-k query on I, one straightforward way works as follows. For each canonical set S of $F(I)$, by using the t-highest plane data structure with $t = k$, we compute the highest k planes of S at $L(I)$. Since there are $O(\log^2 n)$ canonical sets in $F(I)$, the above computes $O(k \log^2 n)$ planes, and among them the highest k planes at $L(I)$ are the answer to the top-k query. The query time is $O(\log^2 n(\log n + k))$. In the following, we present an improved query algorithm with time $O(\log^3 n + k)$.

In the following discussion, for simplicity, whenever we refer to the relative order the planes (e.g., highest, lowest, higher, lower), we refer to their intersections with the line $L(I)$. For example, by “a plane is higher than another plane”, we mean that the first plane has a higher intersection with $L(I)$ than the second plane. For ease of exposition, we assume the intersection points of the planes of $F(I)$ with $L(I)$ are distinct. Note that $F(I)$ is a family of canonical sets; but by slightly abusing the notation, when we say "a plane of $F(I)$", we really mean that the plane is in a canonical set of $F(I)$.

We make use of some idea from Lemma 10 although the details are quite different. Our algorithm has two steps: a main algorithm and a post-processing algorithm. We discuss the main algorithm first.

Let $f = |F(I)|$, and thus $f = O(\log^2 n)$. Let $S_1, S_2, \dots, S_f$ be the canonical sets of $F(I)$. For each canonical set S_i, let S_i^j denote the set of the highest $2^{j-1} \cdot \log n$ planes of S_i for $j = 1, 2, \dots,$ and we let $S_i^j = \emptyset$ for $j = 0$. For each S_i, our main algorithm maintains a subset $S_i' \subseteq S_i$ and an integer $j(i)$, such that $S_i' = S_i^{j(i)}$. We also maintain a max-heap H that contains the lowest plane in the subset S_i' for each $S_i \in F(I)$. Hence, the size of H is $O(\log^2 n)$. The "keys" of the planes in H are the z-coordinates of their intersections with $L(I)$. In addition, our algorithm maintains an integer r, which is the size of a set R of planes. Before the main algorithm stops, $R = \cup_{i=1}^{f} S_i^{j(i)-1}$ (after the main algorithm stops, the definition of R is slightly different; see the details below). Note that our algorithm does not maintain R explicitly, and we use R only to argue the correctness of the algorithm. During the main algorithm, r will get increased, and the main algorithm stops once $r \geq k$ (at which moment we have identified a set of $O(\log^3 n + k)$ planes, and among them the highest k planes are the answer to our top-k query, which will be found later by the post-processing algorithm).

Initially, for each canonical set S_i of $F(I)$, by using the t-highest plane data structure with $t = \log n$, we compute $S_i' = S_i^1$, and further we find the lowest plane in S_i' and insert it into H; the above can be done in $O(\log n + |S_i'|)$ time, which is $O(|S_i'|)$ time due to $|S_i'| = |S_i^1| = \log n$. Also, initially we set $r = 0$ (R is implicitly set to $\emptyset$), and set $j(i) = 1$ for each i with $1 \leq i \leq f$.

Next, we do an "extract-max" operation on H to find the highest plane in H and remove it from H. Suppose the above plane is from a canonical set S_i for some i. Then, we let $R = R \cup S_i'$ and set $r = r + |S_i'|$. Further, by using the t-highest plane data structure with $t = 2\log n$, we compute $S_i' = S_i^2$, and then we find the lowest plane in S_i' and insert it into H; again, the above can be done in $O(|S_i'|)$ time. Finally, we update $j(i) = 2$.

In general, we do an extract-max operation on the current H and suppose the removed plane is from a canonical set S_i for some i. We let $R = R \cup S_i^{j(i)} \setminus S_i^{j(i)-1}$ (note that $S_i^{j(i)-1} \subseteq S_i^{j(i)}$, and thus this just adds those planes of $S_i^{j(i)}$ that are not in $S_i^{j(i)-1}$ to R), and set $r = r + |S_i^j| - |S_i^{j-1}|$. Again, we do not explicitly maintain R but explicitly maintain r. If $r \geq k$, then we stop the main algorithm. Otherwise, by using the t-highest plane data structure with $t = 2^{j(i)} \cdot \log n$, we compute $S_i' = S_i^{j(i)+1}$ (and the previous S_i' is discarded), and further we find the lowest plane in $S_i^{j(i)+1}$ and insert it into H; again, the above can be done in $O(|S_i'|)$ time. Note that for ease of exposition, we assume $|S_i| \geq 2^{j(i)} \cdot \log n$ (otherwise, we can solve the problem by similar techniques with more tedious discussions). Finally, we increase $j(i)$ by one.

The above finishes the main algorithm. After it stops, let $R' = \cup_{i=1}^{f} S_i^{j(i)}$. Let B be the set of k highest planes in $F(I)$, i.e., B is the answer to our top-k query on I. We have the following lemma.

Lemma 12. *$R \subseteq R'$, $B \subseteq R'$, and $|R'| = O(\log^3 n + k)$.*

Proof. We first show $R \subseteq R'$.

Suppose the last extract-max operation on H in the main algorithm removes a plane from the canonical set S_i for $i = a$. Hence, the algorithm stops after we have $R = R \cup S_a^{j(a)} \setminus S_a^{j(a)-1}$. Thus, $S_a^{j(a)} \subseteq R$. Consider any other canonical set S_i with $i \neq a$. According to our algorithm, it always holds that $S_i^{j(i)-1} \subseteq R$. Therefore, we have the following:

$$R = S_a^{j(a)} \cup \bigcup_{1 \leq i \leq f, i \neq a} S_i^{j(i)-1} \text{ and } R' = \bigcup_{1 \leq i \leq f} S_i^{j(i)}.$$

Since $S_i^{j(i)-1} \subseteq S_i^{j(i)}$ for any i, we obtain that $R \subseteq R'$.

Next, we show that $|R'| = O(\log^3 n + k)$.

Indeed, since the algorithm stops right after $r = r + |S_a^{j(a)}| - |S_a^{j(a)-1}| \geq k$ and $R = R \cup S_a^{j(a)} \setminus S_a^{j(a)-1}$, the original value of r before the above increasing is less than k. In other words, $\sum_{1 \leq i \leq f} |S_i^{j(i)-1}| < k$. For each S_i of $F(I)$, if $j(i) = 1$, then $|S_i^{j(i)}| = \log n$ and $|S_i^{j(i)-1}| = 0$; otherwise, $|S_i^{j(i)}| = 2|S_i^{j(i)-1}|$. Therefore, we obtain $|R'| = \cup_{1 \leq i \leq f} S_i^{j(i)} \leq 2 \cdot \sum_{1 \leq i \leq f} |S_i^{j(i)-1}| + f \cdot \log n = 2k + O(\log^3 n) = O(\log^3 n + k)$ because $f = |F(I)| = O(\log^2 n)$.

Finally, we prove $B \subseteq R'$.

Let σ^* denote the plane removed by the last extract-max operation on H in the main algorithm. We claim that σ^* is the lowest plane in R. We prove the claim below.

According to our algorithm, the planes removed by the extract-max operations on H follow the order from high to low. Consider any plane $\sigma \in R$. To prove the claim, it is sufficient to show that σ^* is not higher than σ. According to our algorithm, the first time σ is added in R must be due to an operation on R: $R = R \cup S_i^{j(i)} \setminus S_i^{j(i)-1}$ after an extract-max operation removes a plane σ' from H and σ' is from a canonical set S_i. This implies that $\sigma \in S_i^{j(i)} \setminus S_i^{j(i)-1}$. According to our algorithm, σ' is the lowest plane in the above $S_i^{j(i)}$, and thus, σ' is not higher than σ. On the other hand, since σ^* is the last plane removed by the extract-max operations, σ^* is not higher than σ'. Therefore, σ^* is not higher than σ, and the above claim is proved.

Consider any plane $\sigma \in B$. To show $B \subseteq R'$, it suffices to prove $\sigma \in R'$. If σ is in R, then since $R \subseteq R'$, $\sigma \in R'$ is true. Below we assume $\sigma \notin R$, and thus $\sigma \neq \sigma^*$.

Note that $\sigma \in B$ implies that there are at most $k-1$ planes of $F(I)$ higher than σ. Since $r = |R| \geq k$ and σ^* is the lowest plane in R, σ must be higher than σ^* since otherwise all planes in R would be higher than σ, contradicting with that there are at most $k-1$ planes higher than σ.

Assume σ is in a canonical set S_i for some i. Recall that σ^* is from the canonical set S_a. Note that all planes of S_a higher than σ^* are in $S_a^{j(a)}$. By our definition of R, $S_a^{j(a)} \subseteq R$. Since σ is higher than σ^* and $\sigma \notin R$, we can obtain $i \neq a$. According to our algorithm, after the algorithm stops, H contains a plane σ_i from S_i, and σ_i is the lowest plane in $S_i^{j(i)}$. Recall that, $S_i^{j(i)} \subseteq R'$. This implies that all planes of S_i higher than σ_i are in R'. Since σ^* is removed by an extract-max operation and after the operation σ_i is still in H, σ^* must be higher than σ_i. Because σ is higher than σ^*, σ is higher than σ_i.

In summary, the above discussion obtains the following: σ is in S_i; σ is higher than σ_i; all planes of S_i higher than σ_i are in R'. Thus, we obtain that σ is in R'.

Therefore, we conclude that $B \subseteq R'$, and the lemma follows. □

Based on Lemma 12, if we have the set R' explicitly, then we can compute B in additional $O(\log^3 n + k)$ time by using the linear time selection algorithm [15]. However, the above main algorithm does not explicitly compute R', but it has maintained $j(i)$ for each $S_i \in F(I)$. Since $R' = \cup_{1 \leq i \leq f} S_i^{j(i)}$, we can compute R' by using a t-highest plane query with $t = 2^{j(i)-1} \cdot \log n$ on each canonical set S_i of $F(I)$.

The following lemma gives the running time of our entire top-k query algorithm.

Lemma 13. *The time complexity of our top-k query algorithm is* $O(\log^3 n + k)$.

Proof. We first analyze the main algorithm, whose running time mainly depends on the time of the t-highest plane queries and the time of the operations on the heap H. We first give a bound on the time of the t-highest plane queries, with the help of Lemma 12.

Note that after each t-highest plane query in the main algorithm, we always find the lowest plane in the output planes of the query, whose time is only linear in the number of output planes and is upper bounded by the above query time. In the following, we focus on analyzing the time of the t-highest plane queries.

Consider any canonical set $S_i \in F(I)$. According to our algorithm, for each j with $1 \leq j \leq j(i)$, the main algorithm performs a t-highest plane query on S_i with $t = 2^{j-1} \cdot \log n$ to compute S_i^j, which takes $O(|S_i^j|)$ time (we ignore the $\log n$ factor in the query time because $\log n \leq |S_i^j|$). Hence, the total time of the t-highest plane queries on S_i in the main algorithm is $O(\sum_{j=1}^{j(i)} |S_i^j|)$. Note that

$$\sum_{j=1}^{j(i)} |S_i^j| = \log n \cdot \sum_{j=1}^{j(i)} 2^{j-1} \leq \log n \cdot 2^{j(i)} = 2 \cdot |S_i^{j(i)}|.$$

Recall that $|R'| = \sum_{1 \leq i \leq f} |S_i^{j(i)}|$. Hence, the total time on the t-highest plane queries in the entire main algorithm is $O(|R'|)$, which is $O(\log^3 n + k)$ by Lemma 12.

Next, we analyze the time we spent on the heap H. Recall that the size of H is $O(\log^2 n)$. Initially, we build H on $O(\log^2 n)$ planes, which can be done in $O(\log^2 n)$ time. Later in the algorithm, the operations on H include the extract-max and insertion operations. We need to figure out how many operations were performed on H in the main algorithm.

Consider any extract-max operation on H, and suppose the removed plane is from set S_i. Then, after the operation, we have $R = R \cup S_i^{j(i)} \setminus S_i^{j(i)-1}$, and since $|S_i^{j(i)}| - |S_i^{j(i)-1}| \geq \log n$, the above increases R by at least $\log n$ planes. After that, there is at most one insertion operation on H. Since the main step stops once $|R| \geq k$, the total number of extract-max operations is at most $\frac{k}{\log n}$. The number of insertion operations is also at most $\frac{k}{\log n}$. Since $|H| = O(\log^2 n)$, each operation on H takes $O(\log\log n)$ time. The total time on H is $O(\log^2 n + \frac{k}{\log n} \cdot \log\log n) = O(\log^2 n + k)$.

Therefore, the total time of the main algorithm is $O(\log^3 n + k)$.

Finally, we analyze the running time of the post-processing step, which computes R' and finds the highest k planes in R'. Computing R' is done by doing a t-highest plane query with $t = j(i)$ on each set S_i. Therefore, as above, the total time is at most $|R'|$, which is $O(\log^3 n + k)$. Finding the highest k planes in R' takes $O(|R'|)$ time by using the linear time selection algorithm [15].

Thus, the total time of our top-k query algorithm is $O(\log^3 n + k)$. □

The above discussion leads to the following theorem.

Theorem 9. *We can build in $O(n\log^3 n)$ expected time an $O(n\log^2 n)$ size data structure on P that can answer each top-k query with a bounded query interval in $O(\log^3 n + k)$ time.*

Note that the planes reported by our top-k query algorithm are not in any sorted order.

5. Conclusions

In this paper we present a number of data structures for answering a variety of range queries over uncertain data in one-dimensional space. In general, our data structures have linear or nearly linear sizes and can support efficient queries. While it would be interesting to develop better solutions, an interesting but challenging open problem is whether we can generalize our techniques to solve the corresponding problems in higher dimensions, for which only heuristic results have been proposed [43,44].

Acknowledgements

The authors would like to thank Sariel Har-Peled for several insightful comments, and for suggesting us the canonical set idea (somewhat similar to that in [3]), which leads us to the solutions for the histogram bounded case of the problem. Jian Li was supported in part by the National Basic Research Program of China Grant 2015CB358700, 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61202009, 61033001, 61361136003. Haitao Wang was supported in part by NSF under Grant CCF-1317143.

References

[1] A. Abdullah, S. Daruki, J.M. Phillips, Range counting coresets for uncertain data, in: The 29th Annual Symposium on Computational Geometry, 2013, pp. 223–232.
[2] P. Afshani, T.M. Chan, Optimal halfspace range reporting in three dimensions, in: Proc. of the 20th Annual ACM–SIAM Symposium on Discrete Algorithms, SODA, 2009, pp. 180–186.
[3] P.K. Agarwal, S.-W. Cheng, Y. Tao, K. Yi, Indexing uncertain data, in: Proc. of the 28th Symposium on Principles of Database Systems, 2009, pp. 137–146.
[4] P.K. Agarwal, A. Efrat, S. Sankararaman, W. Zhang, Nearest-neighbor searching under uncertainty, in: Proc. of the 31st Symposium on Principles of Database Systems, PODS, 2012, pp. 225–236.
[5] P.K. Agarwal, M. Sharir, Red-blue intersection detection algorithms, with applications to motion planning and collision detection, SIAM J. Comput. 19 (1990) 297–321.
[6] P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. Nabar, T. Sugihara, J. Widom, Trio: a system for data, uncertainty, and lineage, in: Proc. of the 32nd International Conference on Very Large Data Bases, 2006, pp. 1151–1154.
[7] G. Beskales, M.A. Soliman, I.F. Ilyas, Efficient search for the top-k probable nearest neighbors in uncertain databases, in: Proc. of the VLDB Endowment, 2008, pp. 326–339.
[8] B. Chazelle, Filtering search: a new approach to query-answering, SIAM J. Comput. 15 (3) (1986) 703–724.
[9] B. Chazelle, L.J. Guibas, Fractional cascading: I. A data structuring technique, Algorithmica 1 (1) (1986) 133–162.
[10] B. Chazelle, L.J. Guibas, Fractional cascading: II. Applications, Algorithmica 1 (1) (1986) 163–191.
[11] B. Chazelle, L.J. Guibas, D.T. Lee, The power of geometric duality, BIT 25 (1985) 76–90.
[12] R. Cheng, J. Chen, M. Mokbel, C. Chow, Probabilistic verifiers: evaluating constrained nearest-neighbor queries over uncertain data, in: IEEE International Conference on Data Engineering, ICDE, 2008, pp. 973–982.
[13] R. Cheng, D. Kalashnikov, S. Prabhakar, Evaluating probabilistic queries over imprecise data, in: ACM SIGMOD International Conference on Management of Data, SIGMOD, 2003, pp. 551–562.
[14] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, J.S. Vitter, Efficient indexing methods for probabilistic threshold queries over uncertain data, in: Proc. of the 30th International Conference on Very Large Data Bases, 2004, pp. 876–887.
[15] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.
[16] G. Cormode, F. Li, K. Yi, Semantics of ranking queries for probabilistic data and expected ranks, in: IEEE International Conference on Data Engineering, ICDE, 2009, pp. 305–316.
[17] G. Cormode, A. McGregor, Approximation algorithms for clustering uncertain data, in: ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS, ACM, 2008, pp. 191–200.
[18] N. Dalvi, D. Suciu, Efficient query evaluation on probabilistic databases, VLDB J. (2006) 523–544.
[19] N. Dalvi, D. Suciu, The dichotomy of probabilistic inference for unions of conjunctive queries, J. ACM 59 (6) (2012) 30.
[20] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry—Algorithms and Applications, 3rd edition, Springer-Verlag, Berlin, 2008.
[21] J. Driscoll, N. Sarnak, D. Sleator, R.E. Tarjan, Making data structures persistent, J. Comput. System Sci. 38 (1) (1989) 86–124.
[22] H. Edelsbrunner, L. Guibas, J. Stolfi, Optimal point location in a monotone subdivision, SIAM J. Comput. 15 (2) (1986) 317–340.
[23] G. Frederickson, D. Johnson, The complexity of selection and ranking in $X + Y$ and matrices with sorted columns, J. Comput. System Sci. 24 (2) (1982) 197–208.
[24] S. Guha, K. Munagala, Exceeding expectations and clustering uncertain data, in: ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS, ACM, 2009, pp. 269–278.
[25] R. Jampani, F. Xu, M. Wu, L.L. Perez, C. Jermaine, P.J. Haas, MCDB: a Monte Carlo approach to managing uncertain data, in: ACM SIGMOD International Conference on Management of Data, SIGMOD, ACM, 2008, pp. 687–700.
[26] T.S. Jayram, A. McGregor, S. Muthukrishnan, E. Vee, Estimating statistical aggregates on probabilistic data streams, in: ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS, 2007, pp. 243–252.
[27] D. Kirkpatrick, Optimal search in planar subdivisions, SIAM J. Comput. 12 (1) (1983) 28–35.
[28] A. Knight, Q. Yu, M. Rege, Efficient range query processing on uncertain data, in: Proc. of IEEE International Conference on Information Reuse and Integration, 2011, pp. 263–268.
[29] C. Koch, MAYBMS: a system for managing large probabilistic databases, in: Managing and Mining Uncertain Data, 2009, pp. 149–183.
[30] J. Li, A. Deshpande, Ranking continuous probabilistic datasets, Proc. VLDB Endow. 3 (1), 638–649.

[31] J. Li, B. Saha, A. Deshpande, A unified approach to ranking in probabilistic databases, VLDB J. 20 (2) (2011) 249–275.

[32] J. Li, H. Wang, Range queries on uncertain data, in: Proc. of the 25th International Symposium on Algorithms and Computation, ISAAC, 2014, pp. 326–337.

[33] V. Ljosa, A.K. Singh, APLA: indexing arbitrary probability distributions, in: IEEE International Conference on Data Engineering, ICDE, 2007, pp. 946–955.

[34] J.S.B. Mitchell, L_1 shortest paths among polygonal obstacles in the plane, Algorithmica 8 (1) (1992) 55–88.

[35] M. Potamias, F. Bonchi, A. Gionis, G. Kollios, k-Nearest neighbors in uncertain graphs, Proc. VLDB Endow. 3 (1) (2010) 997–1008.

[36] Y. Qi, R. Jain, S. Singh, S. Prabhakar, Threshold query optimization for uncertain data, in: ACM SIGMOD International Conference on Management of Data, SIGMOD, 2010, pp. 315–326.

[37] C. Re, N. Dalvi, D. Suciu, Efficient top-k query evaluation on probabilistic data, in: IEEE International Conference on Data Engineering, ICDE, 2007, pp. 886–895.

[38] C. Ré, D. Suciu, Efficient evaluation of HAVING queries on a probabilistic database, in: Proc. of the 11th International Conference on Database Programming Languages, 2007, pp. 186–200.

[39] P. Sen, A. Deshpande, L. Getoor, Prdb: managing and exploiting rich correlations in probabilistic databases, VLDB J. 18 (5) (2009) 1065–1090.

[40] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, S.E. Hambrusch, Indexing uncertain categorical data, in: IEEE International Conference on Data Engineering, ICDE, 2007, pp. 616–625.

[41] M.A. Soliman, I.F. Ilyas, C.C. Chang, Top-k query processing in uncertain databases, in: IEEE International Conference on Data Engineering, ICDE, IEEE, 2007, pp. 896–905.

[42] D. Suciu, D. Olteanu, C. Ré, C. Koch, Probabilistic databases, Synth. Lect. Data Manag. 3 (2) (2011) 1–180.

[43] Y. Tao, R. Cheng, X. Xiao, W.K. Ngai, B. Kao, S. Prabhakar, Indexing multi-dimensional uncertain data with arbitrary probability density functions, in: Proc. of the 31st International Conference on Very Large Data Bases, VLDB, 2005, pp. 922–933.

[44] Y. Tao, X. Xiao, R. Cheng, Range search on multidimensional uncertain data, ACM Trans. Database Syst. 32 (2007).

[45] M.L. Yiu, N. Mamoulis, X. Dai, Y. Tao, M. Vaitis, Efficient evaluation of probabilistic advanced spatial queries on existentially uncertain data, IEEE Trans. Knowl. Data Eng. 21 (2009) 108–122.