Theoretical Computer Science 407 (2008) 310–317

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A dynamic data structure for top-k queries on uncertain data

Jiang Chen[a,1], Ke Yi[b,*]

[a] Yahoo! Inc., 701 First Avenue, Sunnyvale, CA 94089, USA

[b] Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

ARTICLE INFO

Article history:
Received 12 December 2007
Received in revised form 15 May 2008
Accepted 11 June 2008
Communicated by G. Ausiello

Keywords:
Uncertain data
Dynamic data structures

ABSTRACT

In an *uncertain data set* $\mathcal{S} = (S, p, f)$ where S is the *ground set* consisting of n elements, $p : S \to [0, 1]$ a probability function, and $f : S \to \mathbb{R}$ a score function, each element $i \in S$ with score $f(i)$ appears independently with probability $p(i)$. The top-k query on $\mathcal{S}$ asks for the set of k elements that has the maximum probability of appearing to be the k elements with the highest scores in a random instance of $\mathcal{S}$. Computing the top-k answer on a fixed $\mathcal{S}$ is known to be easy. In this paper, we consider the dynamic problem, that is, how to maintain the top-k query answer when $\mathcal{S}$ changes, including element insertions and deletions in the ground set S, changes in the probability function p and in the score function f. We present a fully dynamic data structure that handles an update in $O(k \log n)$ time, and answers a top-j query in $O(\log n + j)$ time for any $j \le k$. The structure has $O(n)$ size and can be constructed in $O(n \log k)$ time. As a building block of our dynamic structure, we present an algorithm for the *all-top-k* problem, that is, computing the top-j answers for all $j = 1, \dots, k$, which may be of independent interest.

1. Introduction

Uncertain data naturally arises in a number of modern applications, e.g. imprecise measurement in mobile and sensor data [8], fuzzy duplicates in data warehouse [3], data integration [11], data cleaning [10,5], etc. These applications have called for a lot of research activities in modeling and querying uncertain data in recent years. An uncertain data model represents a probability distribution of all the possible instances of the data set. For example, in the basic uncertain data model [7,2], an *uncertain data set* $\mathcal{S} = (S, p)$ consists of a *ground set* of elements $S = \{1, \dots, n\}$ and a probability function $p : S \to [0, 1]$. It is assumed that each element i appears independently with probability $p(i)$, i.e. the probability that $\mathcal{S}$ instantiates into $I \subseteq S$ is

$$\Pr[I \mid \mathcal{S}] = \prod_{i \in I} p(i) \prod_{i \in S \setminus I} (1 - p(i)).$$

This basic model, in spite of its simplicity, has often been used to approximate the uncertain nature of the underlying data set. We will also adopt this model in this paper. In what follows, we use $I \sim \mathcal{S}$ to denote that I is a random instance generated from $\mathcal{S}$.

Top-k queries are perhaps the most common type of queries in such applications, and have attracted much attention recently. However, all the existing works can only handle one-time top-k computations [12,14,15]. When the underlying data changes, i.e. when the associated probabilities change, or elements are inserted or deleted, the algorithm has to recompute

* Corresponding author. Tel.: +852 23588770.
E-mail addresses: criver@gmail.com (J. Chen), yike@cse.ust.hk (K. Yi).

1 The work was done while the author was at Columbia University.

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.06.016

the answer to the query. This is often unacceptable due to the inherent dynamic nature of the uncertain data in many applications. For instance in data integration, the probability $p(i)$ represents the confidence of its existence, as more data becomes available from different sources, it is conceivable that the confidence levels might experience frequent changes. In this paper, we are interested in designing dynamic data structures that can be used to efficiently maintain the correct top-k answer as the uncertain data set undergoes a series of updates, including probability updates, element insertions and deletions.

Problem definition. There exist a few definitions for top-k queries on an uncertain data set in the literature. We adopt the following natural definition [14], which also requires a score function $f : S \to \mathbb{R}$.

Definition 1 (*[14]*). Let $\mathcal{S} = (S, p, f)$ be an uncertain data set. For any $I \subseteq S$ of size at least k, let $\Psi_k(I)$ be the top-k elements in I according to the score function f; define $\Psi_k(I) = \emptyset$ if $|I| < k$. The answer T^* to a top-k query on $\mathcal{S}$ is

$$T^* = \arg\max_{T\subseteq S} \Pr_{I\sim\mathcal{S}}[\Psi_k(I) = T] = \arg\max_{T\subseteq S} \sum_{\Psi_k(I)=T} \Pr[I \mid \mathcal{S}].$$

Ties can be broken arbitrarily. In other words, T^* is the set of k elements that has the maximum probability of being at the top-k according to the score function in a randomly generated instance.

As a concrete example, S can be a collection of sensors deployed in an environmental study, f represents their precipitation readings, and p measures the probabilities that the sensors are functioning normally. Thus, the top-k result gives us a good idea of where high precipitation occurs. Please see [14] for more potential applications.

As a convention, we assume that all the scores are distinct and $\mathcal{S}$ is given in the decreasing score order, i.e., $f(1) > f(2) > \cdots > f(n)$. Thus the probability of a set T of size k being the top-k elements $\Pr_{I\sim\mathcal{S}}[\Psi_k(I) = T]$ becomes

$$\prod_{j\in T} p(j) \prod_{j<l(T), j\notin T} (1 - p(j)),$$

where $l(T)$ is the last element in T. The problem becomes finding the set of k elements T^* that maximizes the above quantity.

Previous work. Quite a few uncertain data models have been proposed in the database literature [2,4,7,13]. They range from the basic model that we use in this paper, to powerful models that are *complete*, i.e. models that can represent any probability distribution of the data set instances. However, complete models have exponential complexities and are hence uninteresting computationally. Some extensions to the basic model have been introduced to expand the expressiveness of the model while keeping computation tractable. Notably, in the TRIO [2] system, an uncertain data set consists of a number of *x-tuples*, and each x-tuple may include a number of elements associated with probabilities, and represent a discrete probability distribution of these elements being selected. Independence is still assumed among the x-tuples.

Soliman et al. [14] first proposed the problem of top-k query processing in an uncertain data set. Their algorithms have recently been improved by Yi et al. [15], both in the basic uncertain data model and the x-tuple model. In the basic model, if the elements are given in the sorted score order, there is a simple $O(n \log k)$-algorithm to compute the answer of the top-k query in one pass [15]. The process is as follows. We scan the elements one by one, and maintain in a heap the k elements with the highest probabilities seen so far. Every time the heap changes we also incrementally compute the probability that all of these k elements appear while none of the other seen elements appears. In the end we report the k elements that maximizes this probability. It is not difficult to show that these k elements are indeed the top-k answer. However, this simple algorithm is inherently static, it is not clear how to extend it to handle updates without re-computation. In this paper, we develop dynamic algorithms for this problem.

There are a few other top-k query definitions proposed recently. For example, Soliman et al. [14] also proposed the U-kRanks query that concerns with the probability of an element appearing at a particular rank in a randomly generated instance. Another different framework by Ré et al. [12] deals with the problem of finding the k most probable answers for a given certain query, and there the additional scoring dimension is not involved.

Our results. In this paper, we present a dynamic structure of size $O(n)$ that always maintains the correct answer to the top-k query for an uncertain data set $\mathcal{S}$. In fact, our structure answers the top-j query for any $j \le k$ in time $O(\log n + j)$. We conjecture that the problem does not necessarily become easier even if one only requires support for the top-k query. Our structure takes $O(k \log n)$ time to process a probability update, insert a new element into $\mathcal{S}$, or delete an element from $\mathcal{S}$. Note that a score change can be simply accomplished by an element deletion followed by an insertion. Given an uncertain data set whose elements are sorted by score, it takes $O(n \log k)$ time to build the structure. Interestingly, our results show that compared with the $O(n \log k)$ one-time algorithm, it is not asymptotically more expensive to build a data structure that not only supports top-j queries for any $j \le k$, but also allows for efficient updates.

The results presented in this paper also improve upon the bounds in its conference version [6], which gave an update time of $O(k \log k \log n)$ and a construction time of $O(n \log^2 k)$.

Before presenting our dynamic data structure, in Section 2 we consider a generalized version of the top-k problem, the so called *all-top-k* problem, in which we want to compute the top-j answers for all $j \le k$. We give an $O(n \log k + k^2)$-time algorithm for this problem. This algorithm is also a building block of our dynamic data structure, which we describe in Section 3.

In the rest of paper, we assume that $p(i) > 0$ for any $1 \le i < n$. When $p(i) = 0$, we can simply remove i from the data set. In all algorithms presented in this paper, this removal will not increase the asymptotic complexity.

2. The all-top-k problem

In this section, we consider a slightly generalized version of the basic top-k problem. Given an uncertain data set $\mathcal{S}$, in the *all-top-k* problem, we want to compute the answers to all the top-j queries, for $j = 1, \dots, k$. Naïvely applying the basic algorithm in [15] for each j would result in a total running time of $O(nk \log k)$. Below we give an $O(n \log k + k^2)$ algorithm, which will also be useful in our dynamic structure presented in Section 3. Note that the k^2 term in the upper bound is necessary because this problem has a total result size of $\Theta(k^2)$.

Let T_j^* denote the top-j answer. We first observe that once we know $l(T_j^*)$, the last element in T_j^*, the other $j-1$ elements of T_j^* are simply the $j-1$ highest-probability elements in $\{1, \dots, l(T_j^*) - 1\}$. In the following, we focus on computing $l(T_j^*)$ for all $j \le k$, and present an algorithm that runs in $O(n \log k)$ time. After we have the $l(T_j^*)$'s, the T_j^*'s can be computed easily in $O(n \log k + k^2)$ time by scanning all the elements again while keeping a binary tree storing the k elements seen so far that have the largest probabilities in decreasing order.

Algorithm outline. For $i \le n$, let $[i]$ denote the set $\{1, \dots, i\}$. If we ask for the top-j query with the addition condition that the last element has to be i, the answer, denoted by $T_j^*([i])$, is simply the $j-1$ highest-probability elements in $[i-1]$ plus i. For $0 \le j \le k$ and $0 \le i \le n$, define the following matrix

$$\pi_{ji} = \begin{cases} 0 & \text{if } i < j, \\ \prod_{h=1}^{j} p(e_{i,h}) \prod_{h=j+1}^{i} (1 - p(e_{i,h})) & \text{otherwise,} \end{cases}$$

where $e_{i,1}, e_{i,2}, \dots, e_{i,i}$ are elements of $[i]$ sorted in the decreasing order of their probabilities. The probability that $T_j^*([i])$ becomes the top-j set of the uncertain data set $\mathcal{S}$ is

$$\pi'_{ji} = p(i) \cdot \pi_{j-1,i-1}.$$

To find $l(T_j^*)$, we just need to compute the maximum of the jth row of the matrix $(\pi'_{ji})_{k \times n}$, that is,

$$l(T_j^*) = \arg \max_{1 \le i \le n} \pi'_{ji}.$$

The following is an example of a probabilistic data set.

Example 1. Given a 5 element data set with the probabilities $\{0.1, 0.2, 0.1, 0.3, 0.4\}$, π'_{ji}'s are computed as follows. (Recall that we assume data sets are given to us in descending score order. Actual scores are ignored since only their order is of interest.)

For this data set, we have that

$$\begin{array}{rlllll} \pi'_{1*} = & 0.1 & (0.9 \cdot 0.2) & (0.9 \cdot 0.8 \cdot 0.1) & (\mathbf{0.9 \cdot 0.8 \cdot 0.9 \cdot 0.3}) & (0.9 \cdot 0.8 \cdot 0.9 \cdot 0.7 \cdot 0.4) \\ = & 0.1 & 0.18 & 0.072 & \mathbf{0.1944} & 0.18144 \\ \pi'_{2*} = & 0 & (0.1 \cdot 0.2) & (0.9 \cdot 0.2 \cdot 0.1) & (0.9 \cdot 0.2 \cdot 0.9 \cdot 0.3) & (\mathbf{0.9 \cdot 0.8 \cdot 0.9 \cdot 0.3 \cdot 0.4}) \\ = & 0 & 0.02 & 0.018 & 0.0486 & \mathbf{0.07776} \\ \pi'_{3*} = & 0 & 0 & (0.1 \cdot 0.2 \cdot 0.1) & (0.9 \cdot 0.2 \cdot 0.1 \cdot 0.3) & (\mathbf{0.9 \cdot 0.2 \cdot 0.9 \cdot 0.3 \cdot 0.4}) \\ = & 0 & 0 & 0.002 & 0.0054 & \mathbf{0.01944} \end{array}$$

where π_{j*} denotes $(\pi_{j1}, \pi_{j2}, \pi_{j3}, \pi_{j4}, \pi_{j5})$. The maximum probability of each row is marked in bold font. In this example, we have that $l(T_1^*) = 4$ and $l(T_2^*) = l(T_3^*) = 5$.

The naïve algorithm, going row by row and computing their maximums, will take $O(nk \log k)$ time. As sketched before, to compute π_{ji}, the basic algorithm in [15] scans elements one by one and maintains in a heap the j elements with the highest probabilities. This costs $O(\log k)$ time for computing each π_{ji} and thereby results in a total running time of $O(nk \log k)$. In order to obtain an $O(n \log k)$-algorithm we cannot afford to explicitly compute all the π'_{ji}'s, and need to exploit the interdependence among different rows. Below we first show that the matrix π' is *totally monotone* with an appropriate tie breaking rule, thereby admitting an algorithm, known as the SMAWK algorithm, that finds the maximum in each row by only probing $O(n)$ entries in the matrix [1].

This property, however, does not immediately imply an $O(n \log k)$-algorithm, and we still need to efficiently compute π_{ji} on demand as the SMAWK algorithm requests. Specifically, we design a data structure from which the value of π_{ji} can be extracted in $O(\log k)$ time for any given j, i. This structure can be built in $O(n \log k)$ time, thus combining with the SMAWK algorithm [1], we obtain an $O(n \log k)$-time algorithm for computing all the $l(T_j^*)$'s.

Total monotonicity. The following definition is from [1] (with changes of notations). Let $\prec$ be a total ordering defined on the elements in each row of $(A_{ji})_{k\times n}$. Let l_j be the column at which row j attains its maximum under $\prec$. The matrix A is *monotone* if, for any $1 \le j_1 < j_2 \le k$, $l_{j_1} \le l_{j_2}$. A is *totally monotone* if every submatrix of A is monotone. This is equivalent to having every 2×2 submatrix of A be monotone.

Now we define the total ordering $\prec$ on the matrix π' as follows. For any $\pi'_{ji}, \pi'_{ji'}$ from row j, define $\pi'_{ji} \prec \pi'_{ji'}$ if $\pi'_{ji} < \pi'_{ji'}$, or $\pi'_{ji} = \pi'_{ji'}$ and $i < i'$. Essentially, we break ties by choosing the column with a larger index. Also observe that the row-maxima computed under $\prec$ are valid solutions to $l(T^*_j)$, since the our problem allows ties to be broken arbitrarily.

Lemma 1. *The matrix π' is totally monotone under $\prec$.*

Proof. We will show that for $1 \le j < j' \le k$, and $1 \le i < i' \le n$, the 2×2 submatrix

$$\begin{pmatrix} \pi'_{ji} & \pi'_{ji'} \\ \pi'_{j'i} & \pi'_{j'i'} \end{pmatrix}$$

is monotone. We will show that for $j' = j + 1$. The rest will follow by the transitivity of inequalities.

If $i \le j$, we must have $\pi'_{j+1,i} = 0$. By our tie breaking rule, the maximum of the $(j+1)$th row is at column i'. The submatrix is monotone no matter where the maximum of the jth row is.

In the other case we have $i' > i \ge j + 1$, consider the ratio[2] between $\pi'_{j+1,i}$ and π'_{ji}:

$$\frac{\pi'_{j+1,i}}{\pi'_{ji}} = \frac{p(i)\cdot p(e_{i,1})\cdots p(e_{i,j})\cdot(1-p(e_{i,j+1}))\cdots(1-p(e_{i,i-1}))}{p(i)\cdot p(e_{i,1})\cdots p(e_{i,j-1})\cdot(1-p(e_{i,j}))\cdots(1-p(e_{i,i-1}))} = \frac{p(e_{i,j})}{1-p(e_{i,j})}.$$

Similarly, we have

$$\frac{\pi'_{j+1,i'}}{\pi'_{ji'}} = \frac{p(e_{i',j})}{1-p(e_{i',j})}.$$

Since $i' > i$, by definition we have $p(e_{i',j}) \ge p(e_{i,j})$, hence

$$\frac{\pi'_{j+1,i'}}{\pi'_{ji'}} \ge \frac{\pi'_{j+1,i}}{\pi'_{ji}}. \tag{1}$$

When $\pi'_{ji'} < \pi'_{ji}$, the maximum of the jth row is at column i. The submatrix is monotone no matter where the maximum of the $(j + 1)$th row is. Otherwise we have $\pi'_{ji'} \ge \pi'_{ji}$. Combining with (1), we have $\pi'_{j+1,i'} \ge \pi'_{j+1,i}$. Thus, the submatrix is monotone. □

Since $n \ge k$, we can now apply the SMAWK algorithm [1] on π' to find all the maximums in the columns by probing $O(n)$ entries in the matrix.

Computing π_{ji}. It remains to specify how to compute π_{ji} and thereby π'_{ji} for any j, i, as required by the SMAWK algorithm. We first rewrite π_{ji} as

$$\pi_{ji} = \prod_{h=1}^{j} \frac{p(e_{i,h})}{1-p(e_{i,h})} \prod_{h=1}^{i}(1-p(e_{i,h})) = \prod_{h=1}^{j} \frac{p(e_{i,h})}{1-p(e_{i,h})} \prod_{h=1}^{i}(1-p(h)). \tag{2}$$

The second factor of (2) does not depend on j and is simply a prefix-product of the $(1-p(h))$'s. We can easily pre-compute these prefix-products for all i and store them in an array of size n. It takes $O(n)$ time to build this array, and then we can retrieve the prefix-product for any i in constant time. Thus in the following we focus on computing the first factor of (2).

To compute the first factor of (2) for any given j, i, we build a data structure that supports such a query in $O(\log k)$ time and can be constructed in $O(n \log k)$ time. We process the n elements one by one, and maintain a dynamic binary tree (say a red-black tree) of k elements, storing the highest-probability elements among the elements that have been processed, sorted by their probabilities. At the leaf of the tree storing an element e, we maintain the value $p(e)/(1 - p(e))$, and in each internal node u the product of all $p(e)/(1-p(e))$'s in the subtree rooted at u. It is clear that this binary tree can be updated in $O(\log k)$ time per element. The binary tree built after having processed the first i elements can be used to compute the first factor of (2) for any j in $O(\log k)$ time. However, this tree is only useful for a fixed i; after we have processed the $(i + 1)$-th element, the tree for the first i elements is lost.

To support queries for all i, i.e. to be able to query all binary trees that ever appear, we make the data structure *partially persistent*. More precisely, the structure has multiple versions, one corresponding to each binary tree ever built, and allows

[2] In this paper, we use the following convention to handle the multiplication and division of zeros. We keep a counter on how many zeroes have been applied to a product: incrementing the counter for each multiplication by 0 and decrementing for each division by 0. We interpret the final result as 0 if the counter is positive, or ∞ if negative.

queries on any version, but only allows updates to the current version. That is, when we process i, we produce a new binary tree of version i without altering any of the previous versions. Since the binary tree clearly has bounded in-degree, we can use the generic technique of Driscoll et al. [9] to make it partially persistent, without increasing the asymptotic query and update costs. Thus, this persistent structure can be built in $O(n \log k)$ time and supports a query on any version of the binary tree in time $O(\log k)$.

Therefore we have

Lemma 2. *We can spend $O(n \log k)$ preprocessing time to build a data structure that allows us to extract π_{ji} for any j, i in $O(\log k)$ time.*

Plugging Lemma 2 into the SMAWK algorithm yields the desired result.

Theorem 1. *There is an algorithm that computes $l(T_1^*), \ldots, l(T_k^*)$ in $O(n \log k)$ time.*

Corollary 1. *There is an algorithm that solves the all-top-k problem in $O(n \log k + k^2)$ time.*

Proof. We first compute $l(T_1^*), \ldots, l(T_k^*)$ in $O(n \log k)$ time. Next, we scan all the elements again while maintaining the k elements with the largest probabilities seen so far in decreasing order. This can be done in $O(n \log k)$ time by using a binary tree. As soon as we have scanned the first $l(T_j^*)$ elements, for $j = 1, \ldots, k$, we output the k elements maintained. By the definition of $l(T_j^*)$, T_j^* exactly consists of these k elements. The bound in the corollary then follows. □

It is also obvious that if only the probabilities of these T_j^*'s, namely $\pi'_{j,l(T^*,j)}$, are required, $O(n \log k)$ time suffices.

3. The dynamic data structure

We present our dynamic data structure in this section. In Section 3.1, we first discuss how to handle probability updates, and assume that the ground set S is static. In Section 3.2, we talk about how to handle element insertions and deletions.

3.1. The data structure

The structure. We build a balanced binary tree $\mathcal{T}$ on $\{1, \ldots, n\}$. Each leaf of $\mathcal{T}$ stores between k and $2k$ elements. Thus there are a total of $O(n/k)$ leaves, and hence a total number of $O(n/k)$ nodes in $\mathcal{T}$. For any node $u \in \mathcal{T}$, let S^u be the set of elements stored in the leaves of the subtree rooted at u, and $\mathcal{S}^u$ be the corresponding uncertain data set.

For each node u, we solve the all-top-k problem for $\mathcal{S}^u$, except that we do not list or store the all-top-k sets (which takes time and space of $\Omega(k^2)$). Instead, we only store the corresponding probabilities of the sets. More precisely, let $T_j^*(\mathcal{S}^u)$ be the top-j answer for $\mathcal{S}^u$. We compute and store $\rho_j^u = \Pr_{I \sim \mathcal{S}^u}[\Psi_j(I) = T_j^*(\mathcal{S}^u)]$ for all $j = 1, \ldots, k$. Thus the all-top-k solutions for the whole set $\mathcal{S}$ can be found at the root of the whole binary tree.

At each node u, we also compute $k+1$ auxiliary variables π_j^u, for $j = 0 \ldots, k$. If we sort the elements in $\mathcal{S}^u$ by their probabilities in descending order, and suppose that $e_1^u, e_2^u, \ldots, e_{|S^u|}^u$ is such an order, then similar to Section 2, π_j^u is defined as

$$\pi_j^u = \prod_{h=1}^{j} p(e_h^u) \prod_{h=j+1}^{|S^u|} (1 - p(e_h^u)). \tag{3}$$

In other words, π_j^u is the maximum probability for any j-set generated from $\mathcal{S}^u$. Note that $\pi_0^u = \prod_{e \in S^u} (1 - p(e))$ is just the probability that none of S^u appears.

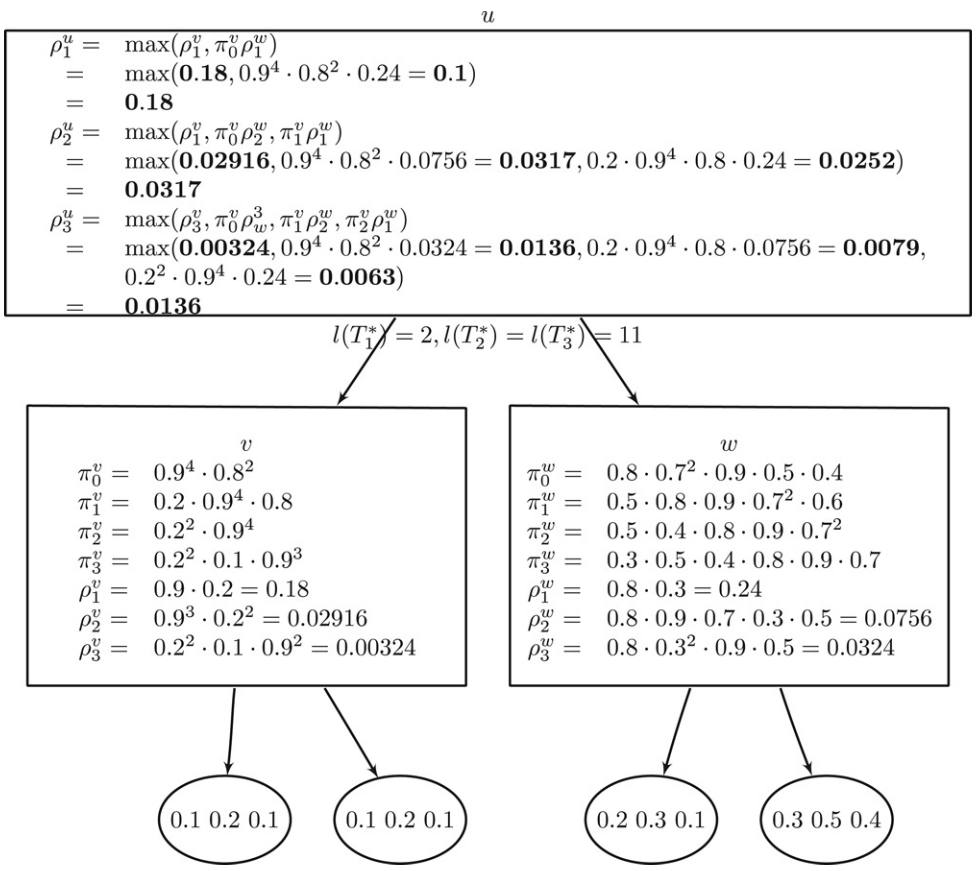
This completes the description of our data structure. See Fig. 1 for the structure built on an example data set of 12 elements. The structure has a size of $O(n)$ since it has $O(n/k)$ nodes and each node takes $O(k)$ space. Below we describe how various operations can be performed on $\mathcal{T}$.

Initializing and updating the π_j^u's. First of all, the π_0^u's are easy to initialize and maintain. For a leave u, we can compute π_0^u in $O(k)$ time by scanning the elements in S^u. Since $\pi_0^u = \pi_0^v \pi_0^w$ for an internal node u with children v and w, we can compute π_0^u for all the internal nodes in a bottom-up fashion in $O(n/k)$ time. So it takes $O(n)$ time to initialize all the π_0^u's. When there is a probability change at one of the leaves, we first update the π_0^u at the leaf; then we update all the π_0^u's along a leaf-to-root path. Therefore, updating the π_0^u's upon each probability change takes $O(\log n)$ time.

Next consider π_j^u for $j = 1, \ldots, k$. Rewriting (3), we get

$$\pi_j^u = \prod_{h=1}^{j} \frac{p(e_h^u)}{1 - p(e_h^u)} \prod_{h=1}^{|S^u|} (1 - p(e_h^u)) = \pi_0^u \cdot \prod_{h=1}^{j} \frac{p(e_h^u)}{1 - p(e_h^u)}. \tag{4}$$

Hence π_j^u is just π_0^u times the product of the first j numbers of the list $\frac{p(e_1^u)}{1-p(e_1^u)}, \frac{p(e_2^u)}{1-p(e_2^u)}, \ldots$. Recall that $e_1^u, e_2^u, \ldots$ are the elements of S^u ordered in decreasing probability, thus if we have $e_1^u, \ldots, e_k^u$ at each u in sorted order, we can compute π_j^u for $j = 1, \ldots, k$ in $O(k)$ time.



u
$\rho_1^u = \max(\rho_1^v, \pi_0^v\rho_1^w) = \max(\mathbf{0.18}, 0.9^4\cdot 0.8^2\cdot 0.24 = \mathbf{0.1}) = \mathbf{0.18}$
$\rho_2^u = \max(\rho_1^v, \pi_0^v\rho_2^w, \pi_1^v\rho_1^w) = \max(\mathbf{0.02916}, 0.9^4\cdot 0.8^2\cdot 0.0756 = \mathbf{0.0317}, 0.2\cdot 0.9^4\cdot 0.8\cdot 0.24 = \mathbf{0.0252}) = \mathbf{0.0317}$
$\rho_3^u = \max(\rho_3^v, \pi_0^v\rho_w^3, \pi_1^v\rho_2^w, \pi_2^v\rho_1^w) = \max(\mathbf{0.00324}, 0.9^4\cdot 0.8^2\cdot 0.0324 = \mathbf{0.0136}, 0.2\cdot 0.9^4\cdot 0.8\cdot 0.0756 = \mathbf{0.0079}, 0.2^2\cdot 0.9^4\cdot 0.24 = \mathbf{0.0063}) = \mathbf{0.0136}$
$l(T_1^*) = 2, l(T_2^*) = l(T_3^*) = 11$
v
$\pi_0^v = 0.9^4\cdot 0.8^2$
$\pi_1^v = 0.2\cdot 0.9^4\cdot 0.8$
$\pi_2^v = 0.2^2\cdot 0.9^4$
$\pi_3^v = 0.2^2\cdot 0.1\cdot 0.9^3$
$\rho_1^v = 0.9\cdot 0.2 = 0.18$
$\rho_2^v = 0.9^3\cdot 0.2^2 = 0.02916$
$\rho_3^v = 0.2^2\cdot 0.1\cdot 0.9^2 = 0.00324$
w
$\pi_0^w = 0.8\cdot 0.7^2\cdot 0.9\cdot 0.5\cdot 0.4$
$\pi_1^w = 0.5\cdot 0.8\cdot 0.9\cdot 0.7^2\cdot 0.6$
$\pi_2^w = 0.5\cdot 0.4\cdot 0.8\cdot 0.9\cdot 0.7^2$
$\pi_3^w = 0.3\cdot 0.5\cdot 0.4\cdot 0.8\cdot 0.9\cdot 0.7$
$\rho_1^w = 0.8\cdot 0.3 = 0.24$
$\rho_2^w = 0.8\cdot 0.9\cdot 0.7\cdot 0.3\cdot 0.5 = 0.0756$
$\rho_3^w = 0.8\cdot 0.3^2\cdot 0.9\cdot 0.5 = 0.0324$
0.1 0.2 0.1
0.1 0.2 0.1
0.2 0.3 0.1
0.3 0.5 0.4

Fig. 1. The dynamic data structure.

For a leaf u, we can build the list $e_1^u, \ldots, e_k^u$ easily in $O(k \log k)$ time by sorting the elements in each S^u by their probabilities. The total time spent for all the leaves is thus $O(n/k \cdot k \log k) = O(n \log k)$. Next we build the lists for the interval nodes bottom-up, level by level. For an internal node u with children v and w, we can obtain this list by merging the two lists associated with v and w, which takes $O(k)$ time, since the top-k probability elements in S^u must be the top-k probability elements in the two lists $e_1^v, \ldots, e_k^v$ and $e_1^w, \ldots, e_k^w$. Thus it takes time $O(n \log k)$ to build all these lists, and also initialize all the π_j^u's. When there is probability change at a leaf u, we first recompute the list at u and update π_j^u, in $O(k)$ time. Then we recompute all the lists associated with the nodes on the leaf-to-root path starting at u. Since recomputing each list takes $O(k)$ time, the total update time for one probability is $O(n \log k)$.

Initializing and updating the ρ_j^u's. Now we proceed to the more difficult part, maintaining the ρ_j^u's. If u is a leaf, then the ρ_j^u's can be computed by invoking the algorithm in Section 2, taking $O(k \log k)$ time per leaf and $O(n \log k)$ overall. For an internal node u, ρ_j^u can be computed as specified in the following lemma. (See Fig. 1 for an example of an update operation.)

Lemma 3. *Let u be an internal node with v and w being its left and right child, respectively. For any $1 \le j \le k$,*

$$\rho_j^u = \max\{\rho_j^v, \max_{1\le h\le j} \pi_{j-h}^v \rho_h^w\}. \tag{5}$$

Proof. Recall that the leaves of the tree are sorted in the descending order of score. Thus the left child of u, namely v, contains elements with higher scores.

By definition, ρ_j^u is the top-j query answer for the uncertain data set $\mathcal{S}^u$. There are two cases for the top-j query answer. Either we choose all of these j elements from S^v, which has a maximum probability of ρ_j^v, or choose at least one element from S^w. The latter case is further divided into j sub-cases: We can choose $j - h$ elements from S^v and h elements from S^w, for $h = 1, \ldots, j$. For each sub-case, the maximum probability is $\pi_{j-h}^v \rho_h^w$. $\square$

The naïve way to maintain the ρ_j^u's is to compute (5) straightforwardly, which takes $\Theta(k^2)$ time per internal node. In the following we show how the running time can be improved to $O(k)$, leading to an overall initialization time of $O(n \log k)$ for the whole structure, and an update time of $O(k \log n)$.

We concentrate on computing the second terms inside the max of (5), with which computing the ρ_j^u's takes only k max-operations. That is, we focus on computing $\bar{\rho}_j^u = \max_{1\le h\le j} \pi_{j-h}^v \rho_h^w$, for $j = 1, \ldots, k$. Define

$$a_{ji} = \begin{cases} \pi_{j-i}^v \rho_i^w & \text{for } 1 \le i \le j \le k \\ 0 & \text{otherwise.} \end{cases}$$

Consider the matrix $(a_{ji})_{k\times k}$. It is easy to see that $\bar{\rho}_j^u$ corresponds to the maximum entry in the jth row. Next we show that this matrix is also totally monotone using a tie breaking rule, and thus we can use the SMAWK algorithm [1] to find all the row maxima in time $O(k)$. Specifically, we define the total ordering $\prec$ for the elements in each row as follows. For $a_{ji}, a_{ji'}$, define $a_{ji} \prec a_{ji'}$ if $a_{ji} < a_{ji'}$, or $a_{ji} = a_{ji'}$ and $i > i'$. Note that this time we break ties in favor of the column with a smaller index.

Lemma 4. *The matrix a is totally monotone under $\prec$.*

Proof. For any $1 \le i < i' \le k$ and $1 \le j < j' \le k$, we will show that the 2×2 submatrix

$$\begin{pmatrix} a_{ji} & a_{ji'} \\ a_{j'i} & a_{j'i'} \end{pmatrix}$$

is monotone. Like the proof of Lemma 1, we will show that for $j' = j + 1$ and the rest will follow.

If $a_{ji'} \le a_{ji}$, by our tie-breaking rule, the maximum of jth row is at column i. No matter where the maximum of $(j + 1)$th row is, the submatrix is monotone.

In the other case, we have that

$$a_{ji'} > a_{ji}. \tag{6}$$

Thus $a_{ji'} > 0$. According to the definition, we have that $j \ge i' > i$. Consider the ratio between $a_{j+1,i}$ and $a_{j,i}$.

$$\frac{a_{j+1,i}}{a_{ji}} = \frac{\pi_{j+1-i}^v \rho_i^w}{\pi_{j-i}^v \rho_i^w} = \frac{p(e_{j+1-i}^v)}{1 - p(e_{j+1-i}^v)}.$$

Similarly,

$$\frac{a_{j+1,i'}}{a_{ji'}} = \frac{p(e_{j+1-i'}^v)}{1 - p(e_{j+1-i'}^v)}.$$

By definition, we have $p(e_{j+1-i'}^v) \ge p(e_{j+1-i}^v)$ and thereby

$$\frac{a_{j+1,i'}}{a_{ji'}} \ge \frac{a_{j+1,i}}{a_{ji}}.$$

Recall that we assume that $p(i) > 0$ for any $1 \le i \le n$. Therefore, both ratios are positive. Combining with (6), this implies $a_{j+1,i'} > a_{j+1,i}$. Thus the submatrix is monotone. □

Therefore, we can compute the ρ_j^u's in $O(k)$ time for each internal node u of $\mathcal{T}$. To summarize, when the probability of an element changes, we first update all the π_j^u values for all the nodes on a leaf-to-root path, taking $O(k)$ time per node. Next, we recompute the ρ_j^u values at the leaf containing the updated element. This takes $O(k \log k)$ time using our all-top-k algorithm of Section 2. Finally, we update the other ρ_j^u values for all nodes on the leaf-to-root path in a bottom-up fashion, taking $O(k)$ time per node. The overall update cost is thus $O(k \log k + k \log n) = O(k \log n)$.

Querying the structure. Once we have the structure available, we can easily extract the top-k query answer by remembering which choice we have made for each ρ_j^u in Lemma 3. We briefly outline the extraction algorithm here. We visit $\mathcal{T}$ in a top-down fashion recursively, starting at the root querying for its top-k answer. Suppose we are at node $u \in \mathcal{T}$ with children v and w, querying for its top-j answer. If $\rho_j^u = \rho_j^v$, then we recursively query v for its top-j answer. Otherwise, suppose $\rho_j^u = \pi_{j-h}^v \rho_h^w$ for some h. We report $e_1^v, \ldots, e_{j-h}^v$ and then recursively query w for its top-h answer. It is not difficult to see that this extraction process takes $O(\log n + k)$ time in total.

Note that our data structure is capable of answering queries for any top-j, $j \le k$. It is not clear to us whether restricting to only the top-k answer will make the problem any easier. We suspect that the all-top-k feature of our data structure is inherent in the problem of maintaining only the top-k answer. For example, in the case when the probability of the element with the highest score, namely $p(1)$, is 0, we need to compute the top-k answer of the rest $n - 1$ elements. However, when $p(1)$ is changed to 1, the top-k answer changes to $\{1\}$ union the top-$(k-1)$ answer of the rest of $n-1$ elements. This example can be further generalized. When $p(1), p(2), \ldots, p(k - 1)$ are changed from 0 to 1 one after another, the top-k answer of the whole data set is changed from the top-k answer, to the top-$(k - 1)$ answer, then to the top-$(k - 2)$ answer,..., and finally to the top-1 answer of the rest $n - k + 1$ elements.

3.2. *Handling element insertions and deletions*

We can handle element insertions and deletions using standard techniques. We make the binary tree $\mathcal{T}$ a dynamic balanced binary tree, say a red-black tree, sorted by scores. To insert a new element, we first find the leaf where the element should be inserted. If the leaf contains less than $2k$ elements, we simply insert the new element, and then update all the affected π_i^u and ρ_i^u values as described previously. If the leaf already contains $2k$ elements, we split it into two, creating a new internal node, which becomes the parent of the two new leaves. After inserting the new element into one of the two new leaves, we update the π_i^u and ρ_i^u values as before. When the tree gets out of balance, we apply rotations. Each rotation may require the re-computation of the π_i^u and ρ_i^u values at a constant number of nodes, but this does not change the overall asymptotic complexity. Deletions can be handled similarly.

Therefore, we reach the main result of this paper.

Theorem 2. *There is a fully dynamic data structure that maintains an uncertain data set under probability changes, element insertions and deletions that takes $O(k \log n)$ time per update, and answers a top-j query in $O(\log n + j)$ time for any $j \leq k$. The structure has size $O(n)$ and can be constructed in $O(n \log k)$ time. All bounds are worst-case.*

4. Concluding remarks

In this paper we present a dynamic data structure for the top-k problem with an update cost of $O(k \log n)$. We conjecture that there is an inherent $\Omega(k)$ lower bound for the problem. As a building block of our main result, we also present an all-top-k algorithm that runs in $O(n \log k + k^2)$ time.

Many directions for this problem remain elusive. For example, we have only considered the basic uncertain data model. It would be interesting if we can extend our approach to other more powerful models, such as the x-tuple model [2]. Another orthogonal direction is to consider other top-k definitions [12,14].

Acknowledgement

The second author was supported in part by Hong Kong Direct Allocation Grant (DAG07/08).

References

[1] A. Aggarwal, M.M. Klawe, S. Moran, P. Shor, R. Wilber, Geometric applications of a matrix-searching algorithm, Algorithmica (1987).
[2] 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. International Conference on Very Large Data Bases, 2006.
[3] R. Ananthakrishna, S. Chaudhuri, V. Ganti, Eliminating fuzzy duplicates in data warehouses, in: Proc. International Conference on Very Large Data Bases, 2002.
[4] O. Benjelloun, A.D. Sarma, A. Halevy, J. Widom, ULDBs: Databases with uncertainty and lineage, in: Proc. International Conference on Very Large Data Bases, 2006.
[5] S. Chaudhuri, K. Ganjam, V. Ganti, R. Motwani, Robust and efficient fuzzy match for online data cleaning, in: Proc. ACM SIGMOD International Conference on Management of Data, 2003.
[6] J. Chen, K. Yi, Dynamic structures for top-k queries on uncertain data, in: Proc. International Symposium on Algorithms and Computation, 2007.
[7] N. Dalvi, D. Suciu, Efficient query evaluation on probabilistic databases, in: Proc. International Conference on Very Large Data Bases, 2004.
[8] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, W. Hong, Model-driven data acquisition in sensor networks, in: Proc. International Conference on Very Large Data Bases, 2004.
[9] J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan, Making data structures persistent, Journal of Computer and System Sciences 38 (1) (1989) 86–124.
[10] H. Galhardas, D. Florescu, D. Shasha, Declarative data cleaning: Language, model, and algorithms, in: Proc. International Conference on Very Large Data Bases, 2001.
[11] A. Halevy, A. Rajaraman, J. Ordille, Data integration: The teenage year, in: Proc. International Conference on Very Large Data Bases, 2006.
[12] C. Ré, N. Dalvi, D. Suciu, Efficient top-k query evaluation on probalistic databases, in: Proc. International Conference on Data Engineering, 2007.
[13] A.D. Sarma, O. Benjelloun, A. Halevy, J. Widom, Working models for uncertain data, in: Proc. International Conference on Data Engineering, 2006.
[14] M.A. Soliman, I.F. Ilyas, K.C. Chang, Top-k query processing in uncertain databases, in: Proc. International Conference on Data Engineering, 2007.
[15] K. Yi, F. Li, D. Srivastava, G. Kollios, Efficient processing of top-k queries in uncertain databases with x-relations, IEEE Transactions on Knowledge and Data Engineering (2008).