Dynamic Structures for Top-k Queries on Uncertain Data

Jiang Chen[1⋆] and Ke Yi[2⋆⋆]

[1] Center for Computational Learning Systems, Columbia University
New York, NY 10115, USA. criver@cs.columbia.edu
[2] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. yike@cse.ust.hk

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 insertion and deletions in the ground set S, changes in the probability function p and the score function f. We present a fully dynamic data structure that handles an update in $O(k \log k \log n)$ time, and answers a top-j query in $O(\log n + j)$ time for any $j \leq k$. The structure has $O(n)$ size and can be constructed in $O(n \log^2 k)$ 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 $j = 1, \ldots, k$, 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 [6], fuzzy duplicates in data warehouse [2], data integration [9], data cleaning [8, 4], 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 [5, 1], an *uncertain data set* $\mathcal{S} = (S, p)$ consists of a *ground set* of elements $S = \{1, \ldots, n\}$ 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

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

⋆ Supported in part by a research contract from Consolidated Edison.
⋆⋆ Supported in part by Hong Kong Direct Allocation Grant (DAG07/08).

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 the following, 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 of the existing works can only handle one-time top-k computations [12, 10, 13]. When the underlying data changes, i.e., when the associated probabilities change, or elements are inserted or deleted, the algorithm has to recompute 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 in the literature. We adopt the following natural definition [12], which also requires a score function $f : S \to \mathbb{R}$.

Definition 1. *[12] Let $\mathcal{S} = (S, p, f)$ be an uncertain data set. For any $I \subseteq S$, let $\Psi_k(I)$ be the top-k elements in I according to the score function f; if $|I| < k$, define $\Psi_k(I) = \emptyset$. Let T be any set of k elements. The answer T^* to a top-k query on $\mathcal{S}$ is $T^* = \arg\max_T \Pr_{I\sim\mathcal{S}}[\Psi_k(I) = T] = \arg\max_T \sum_{\Psi_k(I)=T} \Pr[I \mid \mathcal{S}]$. 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 [12] 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. In this paper, we break ties by choosing the T^* with a smaller $l(T^*)$.

Previous work. Quite a few uncertain data models have been proposed in the database literature [11, 3, 1, 5]. 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 [1] 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. [12] first proposed the problem of top-k query processing in an uncertain data set. Their algorithms have been recently improved by Yi et al. [13], 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 [13]. 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 of these k elements being the top-k answer, i.e., the probability that all of these k elements appear multiplied by the probability that none of the other seen elements appear. In the end we report the k elements that achieve the maximum probability. However, this simple algorithm is inherently static, it is not clear how to extend it to handle updates without recomputation. As illustrated by the above sensor example, the uncertain data set may experience frequent changes, therefore it is important to develop dynamic algorithms for the problem.

There are a few other top-k query definitions proposed recently. For example, Soliman et al. [12] 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. [10] deals with the problem of finding the k most probable answer 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, we support more general queries than just for a specific k. Given any $j \leq k$, our structure answers the top-j query 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 $O(k \log k \log n)$ 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 $O(n \log^2 k)$ time to build the structure. The new structure uses a different approach than the static algorithm, and is based on a decomposition of the problem, which allows for efficient updates.

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 $j = 1, \ldots, k$. We give an $O(n \log^2 k + k^2)$-time algorithm for this problem. This algorithm is also a building block of our dynamic data structure, which we describe in Section 3.

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 $j = 1, \ldots, k$. Naïvely applying the basic algorithm in [13] for each j would result in a total running time of $O(nk \log k)$. Below we give an $O(n \log^2 k + k^2)$ 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 $\Theta(k^2)$.

Henceforth we will denote the top-j answer as T_j^*. 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 $[1, l(T_j^*)]$. In the following, we focus on computing $l(T_j^*)$ for all j, and present an algorithm that runs in $O(n \log^2 k)$ 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 heap of size k. If only the probabilities of these top-j answers are required, $O(n \log k)$ time suffices.

Algorithm outline. To simplify our notation, we use l_j (for $1 \leq j \leq k$) to denote $l(T_j^*)$, the last element in T_j^*. We will progressively process the first i elements of $\mathcal{S}$ and update the corresponding l_j's, as i goes from 1 to n. When we finish processing all n elements, we obtain the l_j's for $\mathcal{S}$. However, since there are $\Theta(nk)$ such values (k values for each position i), we cannot even afford to list all of them explicitly; instead, we store them in a "compressed list" that allows for fast updates. The data structure makes essential use of the following two properties of the changes these l_j's may experience. The first property is monotonicity. Note that the following lemma holds for all uncertainty data sets, including those consisting of the first i elements of $\mathcal{S}$.

Lemma 1. *For any $1 \leq j < j' \leq k$, we have that $l_j \leq l_{j'}$.*

Proof. We only need to prove the case when $j' = j+1$, and the general statement will be an easy consequence. Let T be a set of size j, and $e \notin T$, denote by $r(T, e)$ the ratio of the probability of $T \cup \{e\}$ being the top-$(j+1)$ set to that of T being the top-j. We have[3]

[3] 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.

$$r(T,e) = \frac{\Pr_{I\sim\mathcal{S}}[T\cup\{e\} = \Psi_{j+1}(I)]}{\Pr_{I\sim\mathcal{S}}[T = \Psi_j(I)]} = \frac{\prod_{h\in T\cup\{e\}} p(h) \prod_{h<l(T\cup\{e\}),h\notin T\cup\{e\}}(1-p(h))}{\prod_{h\in T} p(h) \prod_{h<l(T),h\notin T}(1-p(h))}$$
$$= \begin{cases} \frac{p(e)}{1-p(e)}, & \text{if } e < l(T), \\ p(e)\prod_{l(T)<h<e}(1-p(h)), & \text{if } e > l(T). \end{cases}$$

Note that when $e > l(T)$, $r(T,e) = \frac{p(e)}{1-p(e)}\prod_{l(T)<h\le e}(1-p(h)) \le \frac{p(e)}{1-p(e)}$.

Now assuming on the contrary $l_j > l_{j+1}$, let e be an element in T^*_{j+1} but not in T^*_j. We will show that $T^*_j \cup \{e\}$ is more likely to be the top-$(j+1)$ set than T^*_{j+1}, which leads to contradiction. Since $e < l_j$, $r(T^*_j, e) = \frac{p(e)}{1-p(e)} \ge r(T,e)$ for any T. Because $l_j > l_{j+1}$, by the definition of T^*_j and our tie breaking rule, we must have $\Pr_{I\sim\mathcal{S}}[T^*_j = \Psi_j(I)] > \Pr_{I\sim\mathcal{S}}[T^*_{j+1}\setminus\{e\} = \Psi_j(I)]$. Therefore, the probability that $T^*_j \cup \{e\}$ is the top-$(j+1)$ answer is

$$\begin{aligned} \Pr_{I\sim\mathcal{S}}[T^*_j = \Psi_j(I)]\cdot r(T^*_j, e) &> \Pr_{I\sim\mathcal{S}}[T^*_{j+1}\setminus\{e\} = \Psi_j(I)]\cdot r(T^*_j, e) \\ &\ge \Pr_{I\sim\mathcal{S}}[T^*_{j+1}\setminus\{e\} = \Psi_j(I)]\cdot r(T^*_{j+1}\setminus\{e\}, e) \\ &= \Pr_{I\sim\mathcal{S}}[T^*_{j+1} = \Psi_{j+1}(I)], \end{aligned}$$

a contradiction.

The second property is that, when we process the i-th element, l_j either changes to i or stays the same because all newly added j-sets contains i. By Lemma 1, if l_j changes to i, so do all $l_{j'}$'s for $j \le j' \le k$. Thus, to process element i, the problem basically becomes finding the smallest j such that l_j becomes i.

Updating the l_j's. We store $l_1, \ldots, l_{\min\{i,k\}}$ in a list, both j and the value of l_j. By Lemma 1 this list is automatically in the increasing order of both j and l_j. We further compress the list by representing the l_j's with equal values by ranges. For example, if $l_1 = 1, l_2 = l_3 = l_4 = 5, l_5 = l_6 = 6$, then the list looks like $(1,[1,1]),(5,[2,4]),(6,[5,6])$. Suppose that we have a comparison method to decide if l_j becomes i for any j, then we can locate the minimum such j, denoted j^*, as follows. We first visit the compressed list from right to left, checking the boundaries of each range, until we locate the range that contains j^*. Next we do a binary search inside the range to pin down its exact location. Finally, supposing that the entry in the list whose range contains j^* is $(i',[j_1,j_2])$, we first truncate all the trailing entries in the list, and then replace $(i',[j_1,j_2])$ with $(i',[j_1,j^*-1])$ (if $j_1 \le j^*$) and $(i,[j^*,i])$. Note that a special case is when j^* does not exist, i.e., no l_j becomes i. In this case if $i \le k$, we append $(i,[i,i])$ to the list; otherwise we do nothing.

We bound the number of comparisons per element as follows. In the first step when we scan the list from right to left, if we pass an entry, then it will be removed immediately. Thus, the amortized number of comparisons is $O(1)$ for

the first step. The second step involves a binary search inside a range of length at most k, which needs $O(\log k)$ comparisons. Therefore, the algorithm performs $O(n \log k)$ comparisons for all n elements.

The comparison method. To complete the algorithm, we finally specify how to conduct each comparison in the algorithm above, which decides whether some l_j should change to i. Let $T_j^*(l)$ be the highest-probability j-set whose last element is l, i.e., $T_j^*(l)$ consists of l and the $j-1$ elements in $\{1, \ldots, l-1\}$ with the largest probabilities. We need to compute both $\Pr_{I \sim \mathcal{S}}[\Psi_j(I) = T_j^*(l_j)]$ and $\Pr_{I \sim \mathcal{S}}[\Psi_j(I) = T_j^*(i)]$ and compare them. Recall that for a set T of size j,

$$\begin{aligned}\Pr_{I \sim S}[\Psi_j(I) = T] &= \prod_{e \in T} p(e) \prod_{e < l(T), e \notin T} (1 - p(e)) \\ &= \prod_{e \in T} \frac{p(e)}{1 - p(e)} \prod_{e \le l(T)} (1 - p(e)).\end{aligned}$$

The second factor is simply a prefix-product and can be easily maintained for all $l(T)$ with a table of size $O(n)$. To compute $\prod_{e \in T} \frac{p(e)}{1-p(e)}$ for $T = T_j^*(l_j)$ and $T = T_j^*(i)$, we build a data structure that supports the following queries: given any j, l, return the product of $p(e)/(1-p(e))$'s for the $j-1$ highest-probability elements e in $\{1, \ldots, l-1\}$. Below we give such a structure, which answers a query in $O(\log k)$ time and can be constructed in $O(n \log k)$ time. It is obvious that with this structure, we can perform a comparison in $O(\log k)$ time, leading to a total running time of $O(n \log^2 k)$ to process all n elements.

Again 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 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 can be verified that this binary tree can be updated in $O(\log k)$ time per element. The binary tree built after having processed the first $l_j - 1$ elements can be used to compute $\prod_{e \in T} \frac{p(e)}{1-p(e)}$ for $T = T_j^*(l_j)$ in $O(\log k)$ time. The same can be said for i and $T_j^*(i)$. However, the comparison of $T_j^*(l_j)$ and $T_j^*(i)$ requires queries on both binary trees, which are not supported by the progressive processing.

To support queries for all binary trees that ever appear, we make the data structure *partially persistent*, i.e., the structure has multiple versions, one corresponding to each binary tree ever built, and allows 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. [7] 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 $O(\log k)$. However, the space requirement increases to $O(n \log k)$.

This completes the description of the algorithm.

Theorem 1. *There is an algorithm that computes $l_1, \ldots, l_k$ in $O(n \log^2 k)$ time.*

As described at the beginning of this section, this leads to the following corollary.

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

3 The Dynamic Data Structure

We present our dynamic data structure in this section. We begin with probability updates, and assume that the ground set S is static. In Section 3.1, we describe our data structure, which can be updated in time with a naïve $O(k^2 \log n)$ algorithm. We then present a better node merging algorithm in Section 3.2, improving the update time to $O(k \log k \log n)$. Finally, in Section 3.3, 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 π_j^u is defined as

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

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. It is obvious that the structure has a size of $O(n)$.

Initializing and updating the π_j^u's. Rewriting (1), we get

$$\pi_j^u = \prod_{h=1}^{j} \frac{p(e_h^u)}{1-p(e_h^u)} \prod_{h=1}^{|S^u|} (1-p(e_h^u)) = \pi_0^u \cdot \prod_{h=1}^{j} \frac{p(e_h^u)}{1-p(e_h^u)}. \tag{2}$$

Hence π_j^u is just the j-th prefix product of the list $\frac{p(e_1^u)}{1-p(e_1^u)}, \frac{p(e_2^u)}{1-p(e_2^u)}, \ldots$ times π_0^u. This suggests us to maintain the list up to the first k elements. These can be prepared for all the leaves u in $O(n \log k)$ time by sorting the elements in each S^u by their probabilities. For an internal node u with children v and w, we just merge the two lists associated with v and w, which takes $O(k)$ time. To compute π_0^u takes $O(k)$ time per leave, but only $O(1)$ time for an internal node because $\pi_0^u = \pi_0^v \pi_0^w$. Given the list and π_0^u, all π_j^u's can be computed in time $O(k)$ for u. Thus it takes time $O(n \log k)$ to initialize all the π_j^u's. When there is probability change at a leaf, we can update all the affected π_j^u's in $O(k \log n)$ time along the leaf-to-root path.

Initializing and updating the ρ_j^u's. Now we proceed to the more difficult part, maintaining the ρ_i^u's. For a leaf e, the ρ_j^e's can be computed by invoking the algorithm in Section 2, taking $O(k \log^2 k)$ time per leaf and $O(n \log^2 k)$ overall. For an internal node u, ρ_i^u can be computed as specified in the following lemma.

Lemma 2. *Let u be an internal node with v and w being its left and right child, respectively. For any $1 \le j \le k$,*

$$\rho_j^u = \max\{\rho_j^v, \max_{1 \le h \le j} \pi_{j-h}^v \rho_h^w\}. \tag{3}$$

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 $\pi_{j-h}^v \rho_h^w$.

The naïve way to maintain the ρ_j^u's is to compute (3) straightforwardly, which takes $\Theta(k^2)$ time per internal node. In Section 3.2 we present an improved node merging algorithm that computes all ρ_j^u's for u in time $O(k \log k)$. This will lead to an overall initialization time of $(n \log^2 k)$ for the whole structure, and an update time of $O(k \log k \log n)$.

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 2. 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, $j \leq k$. 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 An improved node merging algorithm

Naïvely evaluating (3) takes $\Theta(k^2)$ time. In this section, we present an improved $O(k \log k)$-time algorithm. In the following, we concentrate on computing the second terms inside the max of (3), with which computing ρ_j^u's takes only k max-operations. That is, we focus on computing $\bar{\rho}_j^u = \max_{1 \leq h \leq j} \pi_{j-h}^v \rho_h^w$, for $j = 1, \ldots, k$.

Our algorithm exploits the internal structure of the problem. In Figure 1, we represent each product $\pi_{j-h}^v \rho_h^w$ by a square. Thus, each $\bar{\rho}_j^u$ is the maximum over the corresponding diagonal. We number the diagonals from top-left to bottom-right, so that the product $\pi_{j-h}^v \rho_h^w$ is in diagonal j. We will again make use of the monotonicity property similar to that shown in Lemma 1. For two columns h and h' of Figure 1, we say column h *beats* column h' at diagonal j, where $2 \leq h \leq j$, if the product at the intersection of column h and diagonal j is larger than that at the intersection of column h' and diagonal j. The following lemma shows that these comparisons between two columns exhibit monotonicity.

Lemma 3. *For any $1 \leq h' < h \leq j$, if column h beats column h' at diagonal j, then column h beats column h' at any diagonal j', where $j \leq j' \leq k$.*

Proof. By assumption, we have

$$\pi_{j-h}^v \rho_h^w > \pi_{j-h'}^v \rho_{h'}^w. \tag{4}$$

Rewriting (4), we get

$$\frac{\rho_h^w}{\rho_{h'}^w} > \frac{\pi_{j-h'}^v}{\pi_{j-h}^v}. \tag{5}$$

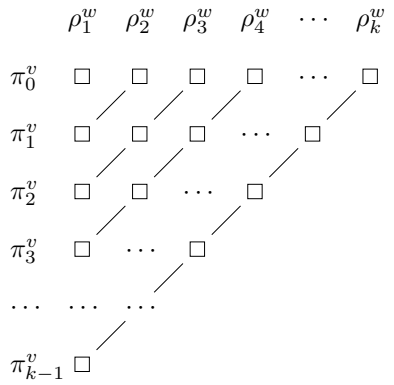

Fig. 1. A square represents the product of the corresponding π_{j-h}^v and ρ_h^w. Each $\bar{\rho}_j^u$ is the maximum on a diagonal.

Recalling (2), the RHS of (5) can be expressed as

$$\prod_{t=j-h+1}^{j-h'} \frac{p(e_t^v)}{1-p(e_t^v)},$$

which becomes smaller when j becomes larger (remember that the e_t^v's are sorted by probabilities in descending order). Therefore, (5), and hence (4), will also hold if we replace j with j'.

We will progress column by column, and for each diagonal keep the current "winner", i.e., the column that beats all the other columns seen so far. After we have processed column j (the last column that has intersection with diagonal j), the winner for diagonal j then determines $\bar{\rho}_j^u$, and we can remove diagonal j from the current maintained list.

By Lemma 3, the way how the winners change exhibits the same pattern as the l_j's do in Section 2. More precisely, when we process column j, if h^* is the minimum h such that the winner of diagonal h changes to column j, then all diagonals after h^* will have their winners changed to j. Thus, we can use the same algorithm (using a compressed list) that we designed for computing the l_j's in Section 2 to maintain the list of winners. Since here comparing two columns at a particular diagonal takes $O(1)$ time (as opposed to $O(\log k)$ in Section 2), the total running time is $O(k \log k)$.

Therefore, we can compute the ρ_j^u's in $O(k \log k)$ time for each node u. 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 $O(k \log^2 k)$ 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 $O(k \log k)$ time per node. The overall update cost is thus $O(k \log^2 k + k \log k \log n) = O(k \log k \log n)$.

3.3 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 recomputation 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 $O(k \log k \log n)$ 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 $O(n \log^2 k)$ 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 $O(k \log k \log n)$. 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 $O(n \log^2 k + k^2)$ 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 [1]. Another orthogonal direction is to consider other top-k definitions [12, 10].

References

1. P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, 2006.
2. R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.
3. O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.
4. S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.
5. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.
6. A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

7. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
8. H. Galhardas, D. Florescu, and D. Shasha. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.
9. A. Halevy, A. Rajaraman, and J. Ordille. Data integration: the teenage year. In *VLDB*, 2006.
10. C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probalistic databases. In *ICDE*, 2007.
11. A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.
12. M. A. Soliman, I. F. Ilyas, and K. C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.
13. K. Yi, F. Li, D. Srivastava, and G. Kollios. Improved top-k query processing in uncertain databases. Technical report, AT&T Labs, Inc., 2007.