A Unified Approach to Ranking in Probabilistic Databases

Jian Li Barna Saha Amol Deshpande
{lijian, barna, amol}@cs.umd.edu
University of Maryland at College Park

ABSTRACT

The dramatic growth in the number of application domains that naturally generate probabilistic, uncertain data has resulted in a need for efficiently supporting complex querying and decision-making over such data. In this paper, we present a unified approach to ranking and top-k query processing in probabilistic databases by viewing it as a multi-criteria optimization problem, and by deriving a set of *features* that capture the key properties of a probabilistic dataset that dictate the ranked result. We contend that a single, specific ranking function may not suffice for probabilistic databases, and we instead propose two *parameterized ranking functions*, called $PRF^!$ and PRF^e, that generalize or can approximate many of the previously proposed ranking functions. We present novel *generating functions*-based algorithms for efficiently ranking large datasets according to these ranking functions, even if the datasets exhibit complex correlations modeled using *probabilistic and/xor trees* or *Markov networks*. We further propose that the parameters of the ranking function be learned from user preferences, and we develop an approach to learn those parameters. Finally, we present a comprehensive experimental study that illustrates the effectiveness of our parameterized ranking functions, especially PRF^e, at approximating other ranking functions and the scalability of our proposed algorithms for exact or approximate ranking.

1. INTRODUCTION

Recent years have seen a dramatic increase in the number of application domains that naturally generate uncertain data and that demand support for executing complex decision-support queries over them. In part, this has been due to the increasing prevalence of applications such as information retrieval [15], data integration and cleaning [2, 11], text analytics [23, 19], and social network analysis [1], where uncertainty arises both because of noisy input data, and because of the statistical inference typically performed on such data. At the same time, large-scale instrumentation of nearly every aspect of our world using sensor monitoring infrastructures has resulted in an abundance of uncertain, noisy data [10, 5].

By their very nature, many of these applications require support for ranking and top-k queries over large datasets. For instance, consider a *House Search* application, where a user is searching for a house using a real estate sales dataset: *House(id, price, size, zipcode, ...)*. Such a dataset, which may be constructed by crawling and combining data from multiple sources, is inherently uncertain and noisy. In fact, the houses that the user prefers the most, are also the most likely to be sold by now. We may denote such uncertainty by associating with each advertisement a *probability* that it is still valid. However, incorporating such uncertainties into the returned answers is a challenge, considering the complex interplay between the score of a house by itself, and the probability that the advertisement is still valid.

The importance of this natural problem has led to much work on ranking in probabilistic databases in recent years. That prior work (which we review in more detail later) has proposed many different functions for combining the scores and the probabilities. We begin with a systematic exploration of these issues by recognizing that ranking in probabilistic databases is inherently a multi-criteria optimization problem, and by deriving a set of *features*, the key properties of a probabilistic dataset that influence the ranked result. We empirically illustrate the diverse and conflicting behavior of several natural ranking functions, and argue that a single specific ranking function may not be appropriate to rank different uncertain databases that we may encounter in practice. Furthermore, different users may weigh the features differently, resulting in different rankings over the same dataset. We then define a general and powerful ranking function, called *PRF*, that allows us to explore the space of possible ranking functions. We discuss its relationship to previously proposed ranking functions, and also identify two specific parameterized ranking functions, called $PRF^!$ and PRF^e, as being interesting. The $PRF^!$ ranking function is essentially a linear weighted ranking function that resembles the scoring functions typically used in information retrieval, web search, data integration, keyword query answering etc. [20, 26, 4, 9, 36]. We observe that $PRF^!$ may not be suitable for ranking large datasets due to its high running time, and instead propose PRF^e, which uses a single parameter and can effectively approximate previously proposed ranking functions for probabilistic databases.

We then develop novel algorithms based on *generating functions* to efficiently rank the tuples in a probabilistic dataset using any PRF ranking function. Our algorithm can handle a probabilistic dataset with arbitrary correlations; however, it is particularly efficient when the probabilistic database contains only *mutual exclusivity* and/or *mutual co-existence* correlations (called *probabilistic and/xor trees* [31]). Our results apply to some of the previously proposed ranking functions as well (one of our results was also independently obtained by Yi et al. [38]). Our main contributions can be summarized as follows:

- We develop a framework for learning ranking functions over probabilistic databases by identifying a set of key *features* and by proposing several parameterized ranking functions.
- We present novel algorithms based on *generating functions* that enable highly efficient processing of top-k queries over very large datasets. Our key algorithm is an $O(n \log(n))$ algorithm for ranking using a PRF^e function over low-correlation datasets

VLDB '09, August 24-28, 2009, Lyon, France

(specifically, constant height probabilistic and/xor trees). The algorithm runs in $O(n)$ time if the dataset is pre-sorted by score.

- We present a polynomial time algorithm for computing the top-k answers for a correlated dataset, where the correlations are represented using a bounded-treewidth Markov network. The algorithm we present is actually for computing the probability that a given tuple is ranked at a given position across all the possible worlds, and is of independent interest.
- We develop a novel, DFT-based algorithm for approximating an arbitrary weighted ranking function using a linear combination of PRF^e functions.
- We present a comprehensive experimental study over several real and synthetic datasets, comparing the behavior of the ranking functions and the effectiveness of our proposed algorithms.

Outline: We begin with a brief discussion of the related work (Section 2). In Section 3, we review our probabilistic database model and the prior work on ranking in probabilistic databases, and propose two parameterized ranking functions. In Section 4, we present our generating functions-based algorithms for ranking. We then present an approach to approximate different ranking functions using our parameterized ranking functions, and to learn a ranking function from user preferences (Section 5). We then briefly sketch an extension of our ranking algorithms to correlated datasets where the correlations are modeled using Markov networks (Section 6). We conclude with an experimental study in Section 7.

2. RELATED WORK

There has been much work on managing probabilistic, uncertain, incomplete, and/or fuzzy data in database systems (see e.g. [29, 15, 5, 7, 37, 27]). With a rapid increase in the number of application domains where uncertain data arises naturally, such as data integration, information extraction, sensor networks, pervasive computing etc., this area has seen renewed interest in recent years [16]. This work has spanned a range of issues from theoretical development of data models and data languages to practical implementation issues such as indexing techniques, and several research efforts are underway to build systems to manage uncertain data (e.g. MYSTIQ [7], Trio [37], ORION [5], MayBMS [27], PrDB [34]). The approaches can be differentiated based on whether they capture only *tuple-level uncertainty*, where "existence" probabilities are attached to the tuples of the database, or only *attribute-level uncertainty*, where (possibly continuous) probability distributions are attached to the attributes, or both. The proposed approaches differ further based on whether they consider correlations or not. Most work in probabilistic databases has either assumed independence [15, 7] or has restricted the correlations that can be modeled [29, 2]. More recently, several approaches have been presented that allow representation of arbitrary correlations [18, 34, 28].

The area of ranking and top-k query processing has also seen much work in databases (see Ilyas et al. [22] for a survey). More recently, several researchers have considered top-k query processing in probabilistic databases. Soliman et al. [35] defined the problem of ranking over probabilistic databases, and proposed two ranking functions to combine tuple scores and probabilities. Yi et al. [38] present improved algorithms for the same ranking functions. Zhang and Chomicki [39] present a desiderata for ranking functions. Ming Hua et al. [21] recently presented a different approach called *probabilistic threshold queries*. Finally, Cormode et al. [6] also present a semantics of ranking functions and a new ranking function called *expected rank*. We will review these ranking functions in detail in next section. There has also been work on top-k query processing in probabilistic databases where the ranking is by the result tuple *probabilities* (i.e., probability and score are identical) [32]. The main challenge in that work is efficient computation of the probabilities, whereas we assume that the probability and score are either given or can be computed easily.

3. PROBLEM FORMULATION

We begin with defining our model of a probabilistic database, called *probabilistic and/xor tree* [31], that allows capturing several common types of correlations. We then review the prior work on top-k query processing in probabilistic databases, and argue that a single specific ranking function may not capture the intricacies of ranking with uncertainty. We then present our parameterized ranking functions, $PRF^!$ and PRF^e.

3.1 Probabilistic Database Model

We use the prevalent *possible worlds semantics* for probabilistic databases [7]. We denote a probabilistic relation with tuple uncertainty by D_T, where T denotes the set of tuples (see Section 4.4 for extensions of our algorithms to attribute uncertainty). The set of all possible worlds is denoted by $PW = \{pw_1, pw_2, ..., pw_n\}$. Each tuple $t_i \in T$ is associated with an existence probability $\Pr(t_i)$ and a score $\mathsf{score}(t_i)$, computed based on a scoring function $\mathsf{score} : T \rightarrow \mathsf{R}$. Usually $\mathsf{score}(t)$ is computed based on the tuple attribute values and measures the relative user preference for different tuples. In a deterministic database, tuples with higher scores should be ranked higher. We use $r_{pw} : T \rightarrow \{1, \ldots, n\} \cup \{\infty\}$ to denote the rank of the tuple t in a possible world pw according to score. If t does not appear in the possible world pw, we let $r_{pw}(t) = \infty$. We say t_1 *ranks higher* than t_2 in the possible world pw if $r_{pw}(t_1) < r_{pw}(t_2)$. For each tuple t, we define a random variable $r(t)$ which denotes the rank of t in D_T. In other words, $\Pr(r(t) = k)$ is the total probability of the possible worlds where t is ranked at position k.

Probabilistic And/Xor Tree Model: Our algorithms can handle arbitrarily correlated relations where correlations are modeled using Markov networks (Section 6). However, in most of this paper, we focus on the *probabilistic and/xor tree model*, introduced in our prior work [31], that can capture only a more restricted set of correlations, but admits highly efficient probability computation algorithms. More specifically, an and/xor tree captures two types of correlations: (1) *mutual exclusivity* (denoted $\bigcirc$ (*xor*)) and (2) *mutual co-existence* ($\wedge$ (*and*)). Two events satisfy the mutual co-existence correlation if, in any possible world, either both events occur or neither occurs. Similarly two events are mutually exclusive if there is no possible world where both happen.

DEFINITION 1. *A probabilistic and/xor tree $\mathcal{T}$ is a tree where each leaf is a singleton tuple and each inner node has a mark, $\bigcirc$ or $\wedge$. For each $\bigcirc$ node u and each of its children v, there is a non-negative value $p(u, v)$ associated with the edge (u, v). Moreover, we require $\sum_{v:(u,v)} p(u, v) \leq 1$. Let $\mathcal{T}_v$ be the subtree rooted at v and $v_1, \ldots, v_l$ be v's children. The subtree $\mathcal{T}_v$ inductively defines a random subset S_v of its leaves by the following independent process:*

- *If v is a leaf, $S_v = \{v\}$.*
- *If v is a $\bigcirc$ node, then $S_v = \begin{cases} S_{v_i} & \text{with prob. } p(v, v_i) \\ \emptyset & \text{otherwise} \end{cases}$*
- *v is a $\wedge$ node, then $S_v = \cup_i S_{v_i}$*

x-tuples (which can be used to specify mutual exclusivity correlations between tuples) correspond to the special case where we

Time	Car Loc	Plate No	Speed	...	Prob	Tuple Id
11:40	L1	X-123	120	...	0.4	t_1
11:55	L2	Y-245	130	...	0.7	t_2
11:35	L3	Y-245	80	...	0.3	t_3
12:10	L4	Z-541	95	...	0.4	t_4
12:25	L5	Z-541	110	...	0.6	t_5
12:15	L6	L-110	105	...	0.1	t_6

World	Prob
$pw_1 = \{t_1, t_2, t_4, t_6\}$	.112
$pw_2 = \{t_1, t_2, t_5, t_6\}$	.168
$pw_3 = \{t_1, t_3, t_4, t_6\}$	.048
$pw_4 = \{t_1, t_3, t_5, t_6\}$	.072
$pw_5 = \{t_2, t_4, t_6\}$	.168
$pw_6 = \{t_2, t_5, t_6\}$	.252
$pw_7 = \{t_3, t_4, t_6\}$	.072
$pw_8 = \{t_3, t_5, t_6\}$	.108

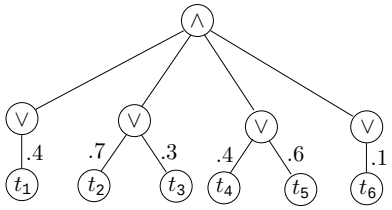


Figure 1: Example of a probabilistic database which contains automatically captured information about speeding cars. Tuple t_2 and t_3 (similarly, t_4 and t_5) are mutually exclusive. The corresponding and/xor tree compactly encodes these correlations.

have a tree of *height* 2, with a ∧ node as the root and only ∨ nodes in the second level. Figure 1 shows an example of an and/xor tree that models the data from a traffic monitoring application [35], where the tuples represent automatically captured traffic data. For example, the leftmost ∨ node indicates t_1 is present with probability .4 and the second ∨ node dictates that exactly one of t_2 and t_3 should appear. The topmost ∧ node tells us the random sets derived from these ∨ nodes coexist.

Probabilistic and/xor trees significantly generalize *x-tuples* [33, 38], block-independent disjoint tuples model, and p-or-sets [8], and can in fact represent any finite set of arbitrary possible worlds [31]. The correlations captured by and/xor trees can be represented by probabilistic c-tables [18] and provenance semirings [17]. However, that does not directly imply an efficient algorithm for ranking. And/xor trees also exhibit superficial similarities to ws-trees [28], which can also capture mutual exclusivity and coexistence between tuples. *We note that no prior work on ranking in probabilistic databases has considered more complex correlations than x-tuples.*

3.2 Ranking over Probabilistic Data: Definitions and Prior Work

The interplay between probabilities and scores complicates the semantics of ranking in probabilistic databases. This was observed by Soliman et al. [35], who first considered this problem and presented two definitions of top-k queries in probabilistic databases. Several other definitions of ranking have been proposed since then. We briefly review the ranking functions we consider in this work.

- *Uncertain Top-*k *(*U-Top*)* [35]: Here the query returns the k-tuple set that appears as the top-k answer in most possible worlds (weighted by the probabilities of the worlds).
- *Uncertain Rank-*k *(*U-Rank*)* [35]: At each rank i, we return the tuple with the maximum probability of being at the i'th rank in all possible worlds. In other words, *U-Rank* returns: $\{t_i, i = 1, 2, .., k\}$, where $t_i = argmax_t(\Pr(r(t) = i))$.
- *Probabilistic Threshold Top-k (*$PT(h)$*)* [21]: The original definition of a probabilistic threshold query asks for all tuples with probability of being in top-h answer larger than a prespecified threshold, i.e., all tuples t such that $\Pr(r(t) \leq h) > threshold$. For consistency with other ranking definitions, we slightly modify the definition and instead ask for the k tuples with the largest $\Pr(r(t) \leq h)$ values.
- *Expected Ranks (*ExpRank*)* [6]: The tuples are ranked in the increasing order by: $\sum_{pw \in PW} \Pr(pw) r_{pw}(t)$, where $r_{pw}(t)$ is defined to be $|pw|$ if $t \notin pw$.
- *Expected Score (*E-Score*)*: Another natural ranking function, also considered by [6], is simply to rank the tuples by their expected score, $\Pr(t)score(t)$.

Normalized Kendall Distance: To compare different ranking functions or criteria, we need a distance measure to evaluate the closeness of two top-k answers. We use the prevalent *Kendall tau* distance defined for comparing top-k answers for this purpose [13]. It is also called *Kemeny distance* in the literature and is considered to have many advantages over other distance metrics [12]. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ denote two full ranked lists, and let $\mathcal{K}_1$ and $\mathcal{K}_2$ denote the top-k ranked tuples in $\mathcal{R}_1$ and $\mathcal{R}_2$ respectively. Then *Kendall tau distance* between $\mathcal{K}_1$ and $\mathcal{K}_2$ is defined to be:

$$dis(\mathcal{K}_1, \mathcal{K}_2) = \sum_{(i,j) \in P(K_1 \cup K_2)} \hat{K}(i, j),$$

where $P(\mathcal{K}_1, \mathcal{K}_2)$ is the set of all unordered pairs of $\mathcal{K}_1 \cup \mathcal{K}_2$; $\hat{K}(i, j) = 1$ if it can be inferred from $\mathcal{K}_1$ and $\mathcal{K}_2$ that i and j appear in opposite order in the two full ranked lists $\mathcal{R}_1$ and $\mathcal{R}_2$, otherwise $\hat{K}(i, j) = 0$. Intuitively the Kendall distance measures the number of inversions or flips between the two rankings. For ease of comparison, we divide the Kendall distance by k^2 to obtain *normalized Kendall* distance, which always lies in $[0, 1]$.

A higher value of the Kendall distance indicates a larger disagreement between the two top-k lists. It is easy to see that if the Kendall distance between two top-k answers is δ, then the two answers must share at least $1-\sqrt{\delta}$ fraction of tuples (so if the distance is 0.09, then the top-k answers share at least 70%, and typically 90% or more tuples). The distance is 0 if two top-k answers are identical and 1 if they are completely disjoint.

	E-Score	$PT(100)$	*U-Rank*	*Exp-Rank*	*U-Top*
E-Score	–	0.1241	0.3027	0.7992	0.2760
$PT(100)$	0.1241	–	0.3324	0.9290	0.3674
U-Rank	0.3027	0.3324	–	0.9293	0.2046
Exp-Rank	0.7992	0.9290	0.9293	–	0.9456
U-Top	0.2760	0.3674	0.2046	0.9456	–

IIP-100,000 (k = 100)

	E-Score	$PT(100)$	*U-Rank*	*Exp-Rank*	*U-Top*
E-Score	–	0.8642	0.8902	0.0044	0.9258
$PT(100)$	0.8642	–	0.3950	0.8647	0.5791
U-Rank	0.8902	0.3950	–	0.8907	0.3160
Exp-Rank	0.0044	0.8647	0.8907	–	0.9263
U-Top	0.9258	0.5791	0.3160	0.9263	–

Syn-IND Dataset with 100,000 tuples (k = 100)

Table 1: Normalized Kendall distance between top-k answers according to various ranking functions for two datasets

Comparing Ranking Functions: We compared the top-100 answers returned by the five ranking functions with each other using the normalized Kendall distance, for two datasets with 100,000 independent tuples each (see Section 7 for a description of the datasets). Table 1 shows the results of this experiment. As we can see, the five ranking functions return wildly different top-k answers for the two datasets, with no obvious trends. For the first dataset, *Exp-Rank* behaves very differently from all other functions, whereas for the second dataset, *Exp-Rank* happens to be quite close

to *E-Score*. However both of them deviate largely from *U-Top*, $PT(h)$ and *U-Rank*. The behavior of *E-Score* is very sensitive to the dataset, especially the score distribution: it is close to $PT(h)$ and *U-Rank* for the first dataset, but far away from all of them in the second dataset (by looking into the results, it shares less than 15 tuples with the Top-100 answers of the others). We observed similar behavior for other datasets, and for datasets with correlations.

This simple experiment illustrates the issues with ranking in probabilistic databases – although several of these definitions seem natural, the wildly different answers they return indicate that none of them may be the "right" definition.

We also observe that in large datasets, *Exp-Rank* tends to give very high priority to a tuple with a high probability even if it has a low score. In our synthetic dataset Syn-IND-100,000 with expected size ≈ 50000, t_2 (the tuple with 2nd highest score) has probability approximately 0.98 and t_{1000} (the tuple with 1000th highest score) has probability 0.99. The expected ranks of t_2 and t_{1000} are approximately 10000 and 6000 respectively, and hence t_{1000} is ranked above t_2 even though t_{1000} is only slightly more probable.

3.3 Parameterized Ranking Functions

Ranking in uncertain databases is inherently a multi-criteria optimization problem, and it is not always clear how to rank two tuples that dominate each other along different axes. Consider a database with two tuples t_1 (score = 100, $\Pr(t_1) = 0.5$), and t_2 (score = 50, $\Pr(t_2) = 1.0$). Even in this simple case, it is not clear whether to rank t_1 above t_2 or vice versa. This is an instance of the classic risk-reward trade-off, and the choice between these two options largely depends on the application domain and/or user preferences.

We propose to follow the traditional approach to dealing with such tradeoffs, by identifying a set of *features*, by defining a parameterized ranking function over these features, and by learning the parameters (weights) themselves using user preferences [20, 26, 4, 9]. To achieve this, we propose a family of ranking functions, parameterized by one or more parameters, and design algorithms to efficiently find the top-k answer according to any ranking function from these families. Our general ranking function, PRF, directly subsumes some of the previously proposed ranking functions, and can also be used to approximate other ranking functions. Moreover, the parameters can be learned from user preferences, which allows us to adapt to different scenarios and different application domains.

Features: Although it is tempting to use the tuple probability and the tuple score as the features, a ranking function based on just those two will be highly sensitive to the actual values of the scores; further, such a ranking function will be insensitive to the correlations in the database, and hence cannot capture the rich interactions between ranking and possible worlds.

Instead we propose to use the following set of features: for each tuple t, we have n features, $\Pr(r(t) = i)$, $i = 1, \cdots, n$, where n is the number of tuples in the database. In other words, for each i, we use the probability that a tuple is ranked at that position across the possible worlds as a feature. This set of features succinctly captures the possible worlds. Further, correlations among tuples, if any, are naturally accounted for, when computing the values of the features. We note that, in most cases, we do not explicitly compute all the features, and instead design algorithms that can directly compute the value of the overall ranking function.

Ranking Functions: Next we define a general ranking function which allows exploring the trade-offs discussed above.

DEFINITION 2. *Let* $\omega : T \times \mathbb{N} \rightarrow \mathbb{C}$ *be a* weight function *that maps a tuple-rank pair to a complex number. The* parameterized ranking function (PRF), $\Upsilon_\omega : T \rightarrow \mathbb{C}$ *in its most general form is defined to be:*

$$\Upsilon_\omega(t) = \sum_{pw : t \in pw} \omega(t, r_{pw}(t)) \cdot \Pr(pw)$$
$$= \sum_{pw : t \in pw} \sum_{i > 0} \omega(t, i) \Pr(pw \wedge r_{pw}(t) = i)$$
$$= \sum_{i > 0} \omega(t, i) \cdot \Pr(r(t) = i).$$

A top-k query returns the k *tuples with the highest* $|\Upsilon_\omega|$ *values.*

In most cases, ω is a real positive function and we just need to find the k tuples with highest Υ_ω values. However we allow ω to be a complex function in order to approximate other functions efficiently (see Section 5.1). Depending on the actual function ω, we get different ranking functions with diverse behaviors. We illustrate some of these choices and relate them to prior ranking functions[1]. We omit the subscript ω if the context is clear.

- If $\omega(t, i) = 1$, the result is the set of k tuples with the highest probabilities [32].
- By setting $\omega(t, i) = score(t)$, we get *E-Score* :
  $$\Upsilon(t) = \sum_{pw : t \in pw} score(t) \Pr(pw) = score(t) \Pr(t)$$
- $\underline{PRF^\omega(h)}$: One important class of ranking functions is when $\omega(t, i) = w_i$ (i.e., independent of t) and $w_i = 0 \, \forall i > h$ for some positive integer h. This forms one of prevalent classes of ranking functions used in domains such as information retrieval and machine learning, with the weights typically learned from user preferences [20, 26, 4, 9].
- Two special cases of the PRF^ω function are:
  1. $\omega(i) = \begin{cases} 1, & i \leq h \\ 0, & \text{otherwise} \end{cases}$. If we return k tuples with highest $\Upsilon_\omega(t)$ value, we have exactly the answer for $PT(h)$.
  2. $\omega_j(i) = \begin{cases} 1, & \text{i=j} \\ 0, & \text{otherwise} \end{cases}$ for some $1 \leq j \leq k$. We can see the tuple with largest Υ_{ω_j} value is the rank-j answer in *U-Rank* query [35].
     This allows us to compute the *U-Rank* answer by evaluating $\Upsilon_{\omega_j}(t)$ for all $t \in T$ and $j = 1, \ldots, k$.
- $\underline{PRF^e(\alpha)}$: Finally, we define PRF^e to be a special case of the PRF^ω function, where $w_i = \omega(i) = \alpha^i$, where α is a constant and may be a real or a complex number.

PRF^ω and PRF^e form the two parameterized ranking functions that we propose in this work. Although PRF^ω is the more natural ranking function and has been used elsewhere, PRF^e is more suitable for ranking in probabilistic databases for various reasons. First, the features as we have defined above are not completely arbitrary, and the features $\Pr(r(t) = i)$ for small i are clearly more important than the ones for large i. Hence in most cases we would like the weight function, $\omega(i)$, to be monotonically non-increasing. PRF^e naturally captures this behavior (as long as $|\alpha| < 1$). More importantly, we can compute the PRF^e function in $O(n \log(n))$ time ($O(n)$ time if the dataset is pre-sorted by score) even for datasets with low degrees of correlations (i.e., modeled by and/xor trees with low heights). This makes it significantly more attractive for ranking over large datasets.

[1]The definition of the *U-Top* introduced in [35] requires the retrieved k tuples belongs to a valid possible world. However, it is not required in our definition, and hence it is not possible to simulate *U-Top* using PRF.

Furthermore, ranking by $PRF^e(\alpha)$, with suitably chosen α, can approximate rankings by many other functions reasonably well even with only real α. Finally, a linear combination of exponential functions, with complex bases, is known to be very expressive in representing other functions [3]. We make use of this fact to approximate many ranking functions by linear combinations of a small number of PRF^e functions, thus significantly speeding up the running time. We revisit this in Section 5.1.

4. RANKING ALGORITHMS

We next present an algorithm for efficiently ranking according to a PRF function. We first present the basic idea behind our algorithm assuming mutual independence, and then consider correlated tuples with correlations represented using an and/xor tree. We then present a very efficient algorithm for ranking using a PRF^e function, and then briefly discuss how to handle attribute uncertainty.

4.1 Assuming Tuple Independence

First we show how the PRF function can be computed in $O(n^2)$ time for a general weight function ω, and for a given set of tuples $T = \{t_1, \ldots, t_n\}$. In all our algorithms, we assume that $\omega(t, i)$ can be computed in $O(1)$ time.

Clearly it is sufficient to compute $\Pr(r(t) = j)$ for all tuples t and $1 \leq j \leq$

in that time), thus matching the best known upper bound by Yi et al. [38] (the original algorithm in [35] runs in $O(n^2 k)$ time).

We remark that the generating function technique can be seen as a variant of dynamic programming in some sense; however, using it explicitly in place of the obscure recursion formula gives us a much cleaner view and allows us to generalize it to handle more complicated tuple correlations. This also leads to an algorithm for extremely efficient evaluation of PRF^e functions (Section 4.3).

4.2 Probabilistic And/Xor Trees

Next we generalize our algorithm to handle a correlated database where the correlations can be captured using an *and/xor* tree. As before, let $T = \{t_1, t_2, \ldots, t_n\}$ denote the tuples sorted in an non-increasing order of their score function, and let $T_i = \{t_1, t_2, \ldots, t_i\}$. Let $\mathcal{T}$ denote the and/xor tree that models the correlations.

Suppose now we need to compute $\Pr(r(t_i) = j)$. Since a tuple with a smaller score does not have any affect on the rank of t_i, it suffices to consider only $\mathcal{T}_i$, the subtree of $\mathcal{T}$ induced by the leaf set T_i (namely, the union of all root-leaf paths with all leaves in T_i). Let $Ch(v) = \{v_1, \ldots, v_l\}$ denote the set of v's children. Let $p_v = \prod_{v_h \in Ch(v)} p(v, v_h)$. For each node $v \in \mathcal{T}_i$, we define generating function $\mathcal{F}_v^i(x, y)$ inductively as follows:

- If v is a leaf, $\mathcal{F}_v^i(x, y) = \begin{cases} x, & v \in T_i \setminus \{t_i\}; \\ y, & v = t_i. \end{cases}$.
- If v is a $\bigcirc$ node,
  $\mathcal{F}_v^i(x, y) = (1 - p_v) + \sum_{v_h \in Ch(v)} \mathcal{F}_{v_h}^i(x, y) \cdot p(v, v_h)$
- If v is a $\wedge$ node, $\mathcal{F}_v^i(x, y) = \prod_{v_h \in Ch(v)} \mathcal{F}_{v_h}^i(x, y)$.

The generating function $\mathcal{F}^i$ for $\mathcal{T}_i$ is the generating function of its root. The following theorem [31] states the close relationship between the probabilities $\Pr(r(t_i) = j)$ and the coefficients of $\mathcal{F}^i$.

THEOREM 1. *[31] Let c_j be the coefficient of the term $x^{j-1}y$ in the generating function $\mathcal{F}^i(x, y)$ defined as above. We have that: $\Pr(r(t_i) = j) = c_j$.*

EXAMPLE 2. *The generating function $\mathcal{F}^5$ for the left hand side tree in Figure 2 is $(.6 + .4x)x(.4x + .6y) = .24x^2 + .16x^3 + .36xy + .24x^2y$. So we get that $\Pr(r(t_5) = 3) = .24$. From Figure 1, we can also see $\Pr(r(t_5) = 3) = \Pr(pw_2) + \Pr(pw_4) = .24$. The right hand side of Figure 2 shows the probabilistic and/xor tree and the generating function computation for Example 1.*

See Algorithm 2 for the pseudocode of the algorithm.

If we expand $\mathcal{F}_v^i$ for each internal node v in a naive way (i.e., we do polynomial multiplication one by one), we can show that the running time is $O(n^2)$ at each internal node and thus $O(n^3)$ overall. If we do divide-and-conquer at each internal node and apply FFT (Fast Fourier Transformation) for the multiplication of polynomials, the running time can be improved to $O(n^2 \log^2 n)$. The details can be found in the extended version of the paper [30].

4.3 Computing a PRF^e Function

Next we present an $O(n \log(n))$ algorithm to evaluate a PRF^e function (the algorithm runs in linear time if the dataset is pre-sorted by score). If $\omega(i) = \alpha^i$, then we observe that:

$$\Upsilon(t_i) = \sum_{j=1}^{n} \Pr(r(t_i) = j)\alpha^j = \mathcal{F}^i(\alpha) \quad (3)$$

This surprisingly simple relationship suggests it is not necessary to expand the polynomials $\mathcal{F}^i(x)$ at all; instead we can evaluate the numerical value of $\mathcal{F}^i(\alpha)$ directly. Again, we note that the value $\mathcal{F}^i(\alpha)$ can be computed from the value of $\mathcal{F}^{i-1}(\alpha)$ in $O(1)$ time using Equation (2). Thus, we have $O(n)$ time algorithm to compute $\Upsilon(t_i)$ for all $1 \le i \le n$ if the tuples are pre-sorted.

Algorithm 2: ANDOR-PRF-RANK($\mathcal{T}$)

```
T^0 = ∅;
for i=1 to n do
    T_i = T_{i-1} ∪ the path from t_i to the root;
    F^i(x,y) = GENE(T_i, t_i);
    Expand F^i(x,y) in the form Σ_j c_j^0 x^j + (Σ_j c_j x^{j-1}) y;
    Υ(t_i) = Σ_{j=1}^{n} ω(t_i, j) c_j;
return k tuples with largest Υ values;
Subroutine: GENE(T, t);
if T is a singleton node then
    if T = {t} then return y else return x
else
    T_i is the subtree rooted at r_i for r_i ∈ Ch(r);
    p = Π_{r_i ∈ Ch(r)} p(r, r_i);
    if r is a ○ node then
        return 1 − p + Σ_{r_i ∈ Ch(r)} p(r, r_i) · GENE(T_i, t);
    if r is a ∧ node then
        return Π_{r_i ∈ Ch(r)} GENE(T_i, t);
```

EXAMPLE 3. *Consider Example 1 and the PRF^e function for t_3. We choose $\omega(i) = .6^i$. Then, we can see that $\mathcal{F}^3(x) = (.5 + .5x)(.4 + .6x)(.4x)$. So, $\Upsilon(t_3) = \mathcal{F}^3(.6) = (.5 + .5 \times .6)(.4 + .6 \times .6)(.4 \times .6) = .14592$*

We can use a similar idea to speed up the computation if the tuples are correlated and the correlations are represented using an and/xor tree. Suppose the generating function for $\mathcal{T}_i$ is $\mathcal{F}^i(x, y) = \sum_j c_j^0 x^j + (\sum_j c_j x^{j-1})y$ and $\Upsilon(t_i) = \sum_{j=1}^{n} \alpha^j c_j$. We observe an intriguing relationship between the PRF^e value and the generating function:

$$\begin{aligned} \Upsilon(t_i) &= \sum_j c_j \alpha^j = \sum_j c_j^0 \alpha^j + (\sum_j c_j \alpha^{j-1})\alpha - \sum_j c_j^0 \alpha^j \\ &= \mathcal{F}^i(\alpha, \alpha) - \mathcal{F}^i(\alpha, 0). \end{aligned}$$

Given this, $\Upsilon(t_i)$ can be computed in linear time by bottom up evaluation of $\mathcal{F}^i(\alpha, \alpha)$ and $\mathcal{F}^i(\alpha, 0)$ in $\mathcal{T}^i$. If we simply repeat it n times, once for each t_i, this gives us a $O(n^2)$ total running time.

By carefully sharing the intermediate results among computations of $\Upsilon(t_i)$, we can improve the running time to $O(n \log(n) + nd)$ where d is the height of the and/xor tree. We sketch this algorithm, which runs in iterations. Suppose the tuples are already pre-sorted by their scores. In iteration i, leaf t_i (the i'th tuple in score order) is added to the tree and the computations are done along the path from t_i to the root. Specifically, the algorithm maintains the following information in each inner node v: the numerical values of $\mathcal{F}_v^i(\alpha, \alpha)$ and $\mathcal{F}_v^i(\alpha, 0)$. The values on node v need to be updated when the value of one of its children changes. Therefore, in each iteration, the computation only happens on the path from t_i to the root. Since we update at most d nodes for each newly added node, the running time is $O(nd)$. The updating rule for $\mathcal{F}_v^i(.,.)$(both $\mathcal{F}_v^i(\alpha, \alpha)$ and $\mathcal{F}_v^i(\alpha, 0)$) in node v is as follows. We assume v's child, say u, just had its values changed.

1. v is a $\wedge$ node, $\mathcal{F}_v^i(.,.) \leftarrow \mathcal{F}_v^{i-1}(.,.)\mathcal{F}_u^i(.,.)/\mathcal{F}_u^{i-1}(.,.)$
2. v is a $\bigcirc$ node, then:
   $\mathcal{F}_v^i(.,.) \leftarrow \mathcal{F}_v^{i-1}(.,.) + p(v, u)\mathcal{F}_u^i(.,.) - p(v, u)\mathcal{F}_u^{i-1}(.,.)$

We note that, for the case of *x-tuples*, which can be represented using a two-level tree, this gives us an $O(n\log(n))$ algorithm for ranking according to PRF^e.

4.4 Attribute Uncertainty or Uncertain Scores

We briefly sketch how we can do ranking over tuples with discrete attribute uncertainty where the uncertain attributes are part of the tuple scoring function (if the uncertain attributes do not affect the tuple score, then they can be ignored for the ranking purposes). More generally, this approach can handle the case when there is a discrete probability distribution over the score of the tuple.

The algorithm works by treating the alternatives of the tuples (with a separate alternative for each different possible score for the tuple) as different tuples, and adding an *xor* constraint over the alternatives. We can then use the algorithm for the probabilistic and/xor tree model to find the values of the PRF function for each resulting tuple separately. In a final step, we calculate the score for each original tuple by adding the scores of its alternatives. If the original tuples were independent, the complexity of this algorithm is $O(n^2)$ for computing the PRF function, and $O(n\log(n))$ for computing the PRF^e function where n is the size of the input, i.e., the total number of different possible scores.

5. APPROXIMATING AND LEARNING RANKING FUNCTIONS

In this section, we discuss how to choose the PRF functions and their parameters. Depending on the application domain and the scenarios, there are two approaches to this:

- If we know the ranking function we would like to use (say $PT(h)$), then we can either simulate or approximate it using appropriate PRF functions.
- If we are instead provided user preferences data, we can learn the parameters from them. Clearly, we would prefer to use a PRF^e function, if possible, since it admits highly efficient ranking algorithms.

For this purpose, we begin with presenting an algorithm to find an approximation to an arbitrary $PRF^!$ function using a linear combination of PRF^e functions. We then discuss how to learn a $PRF^!$ function from user preferences, and finally present an algorithm for learning a single PRF^e function.

5.1 Approximating $PRF^!$ using PRF^e Functions

A linear combination of complex exponential functions is known to be very expressive, and can approximate many other functions very well [3]. Specifically, given a $PRF^!$ function, if we can write $\omega(i)$ as: $\omega(i) \approx \sum_{l=1}^{L} u_l \alpha_l^i$, then we have that:

$$\Upsilon(t) = \sum_i \omega(i)\Pr(r(t)=i) \approx \sum_{l=1}^{L} u_l \left(\sum_i \alpha_l^i \Pr(r(t)=i)\right)$$

This reduces the computation of $\Upsilon(t)$ to L individual PRF^e function computations, each of which only takes linear time. This gives us an $O(n\log(n) + nL)$ time algorithm for approximately ranking using $PRF^!$ function for independent tuples (as opposed to $O(n^2)$ for exact ranking).

Several techniques have been proposed for finding such approximations using complex exponentials [24, 3]. Those techniques are however computationally inefficient, involving computation of the inverses of large matrices and the roots of polynomials of high orders, and may be numerically unstable.

In this section, we present a clean and efficient algorithm, based on Discrete Fourier Transforms (DFT), for approximating a function $\omega(i)$, that approaches zero for large values of i. As we noted earlier, this captures the typical behavior of the $\omega(i)$ function. An example of such a function is the step function ($\omega(i) = 1\ \forall i \le h, = 0 \forall i > h$) which corresponds to the ranking function $PT(h)$. At a high level, our algorithm starts with a DFT approximation of $\omega(i)$ and then adapts it by adding several damping, scaling and shifting factors.

Discrete Fourier transformation (DFT) is a well known technique for representing a function as a linear combination of complex exponentials (also called *frequency domain representation*). More specifically, a discrete function $\omega(i)$ defined on a finite domain $[0, N-1]$ can be decomposed into exactly N exponentials as:

$$\omega(i) = \frac{1}{N}\sum_{k=0}^{N-1} \psi(k) e^{\frac{2\pi}{N} \mathbf{i} k i} \qquad i = 0, \ldots, N-1. \tag{4}$$

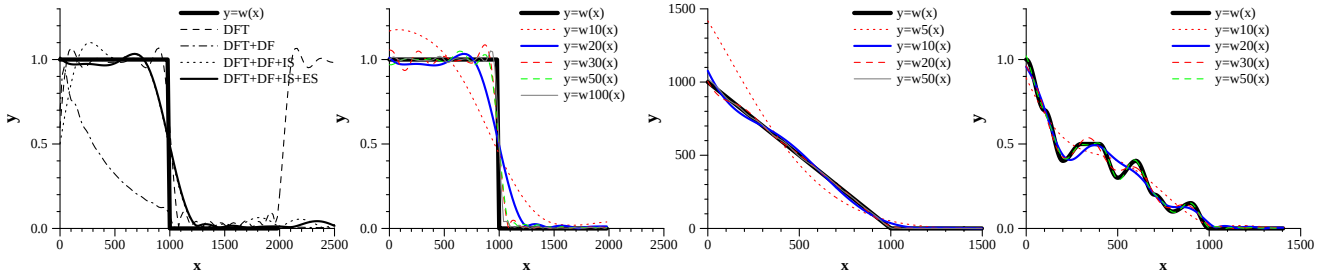
where $\mathbf{i}$ is the imaginary unit and $\psi(0), \cdots, \psi(N-1)$ denotes the DFT transform of $\omega(0), \cdots, \omega(N-1)$. If we want to approximate ω by fewer, say L, exponentials, we can instead use the L DFT coefficients with maximum absolute value. For clarity, we assume that $\psi(0), \ldots, \psi(L-1)$ are those coefficients. Then our approximation ω_L^{DFT} of ω by L exponentials is given by:

$$\omega_L^{DFT}(i) = \frac{1}{N}\sum_{k=0}^{N-1} \psi(k) e^{\frac{2\pi}{N} \mathbf{i} k i} \qquad i = 0, \ldots, N-1. \tag{5}$$

However, DFT utilizes only complex exponentials of unit norm, i.e., $e^{\mathbf{i}r}$ (where r is a real), which makes this approximation periodic (with a period of N). This is not suitable for approximating an ω function used in PRF, which is typically a monotonically non-increasing function. If we make N sufficiently large, say larger than the total number of tuples, then we usually need a large number of exponentials (L) to get a reasonable approximation. Moreover, computing DFT for very large N is computationally non-trivial. Furthermore, the number of tuples n may not be known in advance.

We next present a set of nontrivial tricks to adapt the base DFT approximation to overcome these shortcomings. To illustrate our method, we use the step function $\omega(i) = \begin{cases} 1, & i < N \\ 0, & i \ge N \end{cases}$ as our running example to show our method and the specific shortcoming it addresses. We assume $\omega(i)$ takes non-zero values within interval $[0, N-1]$ and the absolute values of both $\omega(i)$ and $\omega_L^{DFT}(i)$ are bounded by B.

1. (DFT) We perform pure DFT on the domain $[1, aN]$, where a is a small integer constant (typically < 10).
2. (Damping Factor (DF)) We introduce a damping factor $\eta \le 1$ such that $B\eta^{aN} \le \varepsilon$ where ε is a small positive real (for example, 10^{-5}). Our new approximation becomes:
$$\omega_L^{DFT+DF}(i) = \eta^i \cdot \omega_L^{DFT}(i) = \frac{1}{N}\sum_{k=0}^{N-1} \psi(k)(\eta e^{\frac{2\pi}{N} \mathbf{i} k})^i. \tag{6}$$
By incorporating this damping factor, we have that $\lim_{i\to+1} \omega_L^{DFT+DF}(i) = 0$. Especially, $\omega_L^{DFT+DF}(i) \le \varepsilon$ for $i > \alpha N$.
3. (Initial Scaling (IS)) Use of the damping factor gives a biased approximation when i is small (see Figure 3(i)). Taking the step function as an example, $\omega_L^{DFT+DF}(i)$ is approximately η^i for $0 \le i < N$ instead of 1. To rectify this, we initially perform DFT on a different sequence $\hat{\omega}(i) = \eta^{-i}\omega(i)$ (rather than $\omega(i)$) on domain $\in [0, aN]$. This gives us an unbiased approximation, which we denote by $\omega^{DFT+DF+IS}$.



(i) w(x) = step function, L = 20 (ii) w(x) = step function (iii) w(x) =1000-x (x<=1000), =0 (x>1000) (iv) w(x) = an arbitrary smooth function

Figure 3: Approximating functions using linear combinations of complex exponentials

4. (Extending and Shifting (ES)) This trick is in particular tailored for optimizing the approximation performance for ranking functions. DFT does not perform well at discontinuous points, specifically at $i = 0$ (the left boundary), which can significantly affect the ranking approximation. To handle this, we extrapolate ω to make it continuous around 0. Let the resulting function be ω which is defined on $[-bN, +\infty]$ for small $b > 0$. Again, taking the step function for example, we let $\omega(i) = \begin{cases} 1, & -bN \leq i < N; \\ 0, & i \geq N. \end{cases}$ Then, we shift $\omega(i)$ rightwards by bN to make its domain lie entirely in positive axis, do initial scaling and perform DFT on the resulting sequence. We denote the approximation of the resulting sequence by $\varpi^0(i)$ (by performing (6)). For the approximation of original $\omega(i)$ values, we only need to do corresponding leftward shifting , namely $\varpi^{\mathrm{DFT+DF+IS+ES}}(i) = \varpi^0(i + bN)$. We can see from Figure 3(i) that DFT+DF+IS+ES produces a much better approximation than others around $i = 0$.

Figures 3(i) and (ii) illustrate the efficacy of our approximation technique for the step function. As we can see, we are able to approximate that function very well with just 20 or 30 coefficients. Figure 3(iii) and (iv) show the approximations for a piecewise linear function and an arbitrarily generated continuous function respectively, both of which are much easier to approximate than the step function.

5.2 Learning a PRF^1 or PRF^e Function

Next we address the question of how to learn the weights of a PRF^1 function or the α for a single PRF^e function from user preferences. To learn a linear combination of PRF^e functions, we first learn a PRF^1 function and then approximate it as above.

Prior work on learning ranking functions (e.g., [20, 26, 4, 9]) assumes that the user preferences are provided in the form of a set of pairs of tuples, and for each pair, we are told which tuple is ranked higher. Our problem differs slightly from this prior work in that, the features that we use to rank the tuples (i.e., $\Pr(r(t) = i), i = 1, \ldots, n$) cannot be computed for each tuple individually, but must be computed for the entire dataset (since the values of the features for a tuple depend on the other tuples in the dataset). Hence, we assume that we are instead given a small sample of the tuples, and the user ranking for all those tuples. We compute the features assuming this sample constitutes the entire relation, and learn a ranking function accordingly, with the goal to find the parameters (the weights w_i for PRF^1 or the parameter α for PRF^e) that minimizes the number of disagreements with the provided sample ranking.

Given this, the problem of learning PRF^1 is identical to the problem addressed in the prior work, and we utilize the algorithm based on *support vector machines (SVM)* [26] in our experiments.

On the other hand, we are not aware of any work that has addressed learning a ranking function like PRF^e. We use a simple binary search-like heuristic to find the optimal real value of α that minimizes the Kendall distance between the user-specified ranking and the ranking according to $PRF^e(\alpha)$. In other words, we try to find $\arg\min_{\alpha \in [0;1]}(\mathrm{dis}(\sigma, \sigma(\alpha)))$ where dis() is the Kendall distance between two rankings, σ is the ranking for the given sample and $\sigma(\alpha)$ is the one obtained by using $PRF^e(\alpha)$ function. Suppose we want to find the optimal a within the interval $[L, U]$ now. We first compute $\mathrm{dis}(\sigma, \sigma(L + i \cdot \frac{U-L}{10}))$ for $i = 1, \ldots, 9$ and find i for which the distance is the smallest. Then we reduce our search range to $[\max(L, L+(i-1)\cdot\frac{U-L}{10}), \min(U, L+(i+1)\cdot\frac{U-L}{10})]$ and repeat the above recursively. Although this algorithm can only converge to a local minimum, in our experimental study, we observed that all of the prior ranking functions exhibit a uni-valley behavior (Section 7), and in such cases, this algorithm finds the global optimal.

6. PRF COMPUTATION FOR ARBITRARY CORRELATIONS

Among many models for capturing the correlations in a probabilistic database, graphical models (Markov or Bayesian networks) perhaps represent the most systematic approach [34]. The appeal of graphical models stems both from the pictorial representation of the dependencies, and a rich literature on doing inference over them. In this section, we sketch an extension of our generating function-based algorithm for computing PRF to handle correlations represented using a graphical model. The resulting algorithm is a non-trivial dynamic program over the junction tree of the graphical model, combined with the generating function method. Our main result is that we can compute the PRF function in polynomial time if the junction tree of the graphical model has bounded treewidth. It is worth noting that this result can not subsume our algorithm for and/xor trees (Section 4.2) since the treewidth of the moralized graph of a probabilistic and/xor tree may not be bounded.

Definitions: We start with briefly reviewing some notations and definitions related to graphical models and junction trees. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of tuples in D_T, sorted in an non-increasing order of their score values. For each tuple t in T, we associate an indicator random variable X_t, which is 1, if t is present and 0 otherwise. Let $\mathcal{X} = \{X_{t_1}, \ldots, X_{t_n}\}$ and $\mathcal{X}_i = \{X_{t_1}, \ldots, X_{t_i}\}$. The correlations among these variables may be represented using either a directed or an undirected graphical model; we however assume that we are provided with an equivalent junction tree over the variables (which can be constructed using standard algorithms [14]).

Let $\mathcal{T}$ be a tree with each node v associated with a subset $C_v \subseteq \mathcal{X}$. We say $\mathcal{T}$ is a *junction tree* if any intersection $C_u \cap C_v$ for any $u, v \in \mathcal{T}$ is contained in C_w for every node w on the unique path between u and v in $\mathcal{T}$ (this is called the *running intersec-*

Figure 4: (i) A graphical model; (ii) A junction tree for the model along with the (calibrated) potentials.

tion property). The treewidth of a junction tree is defined to be $\max_{v \in T} |C_v| - 1$. Denote $S_{u,v} = C_v \cap C_u$ for each $(u,v) \in T$. We call $S_{u,v}$ a separator since the removal of $S_{u,v}$ disconnects the graphical model.

We associate each clique C_v (and each separator $S_{u,v}$) with a potential $\psi_v(C_v)$ (resp. $\psi_{u,v}(S_{u,v})$), which is a function over all variables $X_{t_i} \in C_v$ ($X_{t_i} \in S_{u,v}$) and represents the correlations among those variables. Without loss of generality, we assume that the potentials are calibrated, that is, the potential corresponding to a clique (separator) is exactly the joint probability distribution over the variables in that clique (separator). Given a junction tree with arbitrary potentials, calibrated potentials can be computed using the message passing algorithm [14].

For a set of variables S, we use $Pr(S)$ to denote the joint probability distribution over those variables. Then the joint probability distribution of X, whose correlations can be captured using a calibrated junction tree T, can be written as:

$$Pr(X) = \frac{\prod_{v \in T} \psi_v(C_v)}{\prod_{(u,v) \in T} \psi_{u,v}(S_{u,v})} = \frac{\prod_{v \in T} Pr(C_v)}{\prod_{(u,v) \in T} Pr(S_{u,v})}$$

Figures 4 (i) and (ii) show an undirected graphical model over five random variables $X_1, \ldots, X_5$, and a calibrated junction tree T over them.

Algorithm Sketch: Our dynamic programming-based algorithm computes $Pr(r(t_i) = h)$ given any $t_i \in T$, for all $1 \le h \le n$, in polynomial time if the treewidth of T is bounded by a constant. The algorithm begins by rooting T at a node r such that $X_{t_i} \in C_r$. The dynamic program then runs bottom up, from the leaves to the root of the junction tree T. Let T_v denote the subtree rooted at a node v in the junction tree, and let C_v denote the corresponding clique. For each such node, we recursively compute:

$$Pr(K_v^i) = Pr((\sim_v, \sim_v^i)), \quad \forall \sim_v \in \{0,1\}^{|C_v|}, \forall 0 \le \sim_v^i \le n$$

which is the probability that the variables in C_v take the values indicated by the Boolean vector $\sim_v$ (called a configuration) and the number of variables in $T_v \setminus X_i$ is exactly equal to $\sim_v^i$.

After computing all of these values using dynamic programming, we can compute $Pr(r(t_i) = h), \forall h$, as:

$$Pr(r(t_i) = h) = \sum_{\sim_r^i = h, \; \sim_r : \sim_r[t_i] = 1} Pr(K_r^i)$$

In other words, we compute the total probability that $X_{t_i} = 1$, and that exactly h variables in X_i are equal to 1 (i.e., exactly $h-1$ tuples ranked above t_i are present in the possible world).

Given the above framework, we can construct a recursive formula for computing $Pr(K_v^i)$ from (1) the computed values for the children of the node v, and (2) the joint probability distributions corresponding to the clique and its children. However it is not computationally feasible to evaluate that recursion formula directly. Instead we develop a generating functions-based algorithm for that purpose, which allows us to efficiently compute $Pr(K_v^i)$ for all nodes v in the junction tree. Please see the full version of the paper [30] for complete details and the proofs of correctness.

Running Time: We need to run our dynamic program n times for each tuple t_i. The time complexity is $O(2^{tw}(n2^{tw} + n^2)|T|)$ for each execution of dynamic program, resulting in an overall time complexity of $O(2^{tw} n^2 (2^{tw} + n)|T|)$, where tw is the treewidth of the junction tree T.

7. EXPERIMENTAL STUDY

We conducted an extensive empirical study over several real and synthetic datasets to illustrate: (a) the diverse and conflicting behavior of different ranking functions proposed in the prior literature, (b) the effectiveness of our parameterized ranking functions, especially PRF^e, at approximating other ranking functions, and (c) the scalability of our new generating functions-based algorithms for exact and approximate ranking. We discussed the results supporting (a) in Section 3.2. In this section, we focus on (b) and (c).

Datasets: We mainly use the International Ice Patrol (IIP) Iceberg Sighting Dataset[2] for our experiments. This dataset was also used in prior work on ranking in probabilistic databases [25, 21]. The database contains a set of iceberg sighting records, each of which contains the location (latitude, longitude) of the iceberg, and the number of days the iceberg has drifted, among other attributes. Detecting the icebergs that have been drifting for long periods is crucial, and hence we use the number of days drifted as the ranking score. The sighting record is also associated with a confidence-level attribute according to the source of sighting: R/V (radar and visual), VIS (visual only), RAD (radar only), SAT-LOW (low earth orbit satellite), SAT-MED (medium earth orbit satellite), SAT-HIGH (high earth orbit satellite), and EST (estimated). We converted these seven confidence levels into probabilities 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, and 0.4 respectively. We added a very small Gaussian noise to each probability so that ties could be broken. There are nearly a million records available from 1960 to 2007; we created 10 different datasets for our experimental study containing 100,000 (IIP-100,000) to 1,000,000 (IIP-1,000,000) records, by uniformly sampling with replacement from the original dataset.

Along with the real datasets, we also use several synthetic datasets with varying degrees of correlations, where the correlations are captured using probabilistic and/xor trees. The tuple scores (for ranking) were chosen uniformly at random from [0, 10000]. The corresponding and/xor trees were also generated randomly starting with the root, by controlling the height (L), the maximum degree of the non-root nodes (d), and the proportion of $\wedge$ and $\vee$ nodes (X/A) in the tree. Specifically, we use five such datasets:

(1) Syn-IND (independent tuples), (2) Syn-XOR (L=2,X/A=1,d=5), (3) Syn-LOW (L=3,X/A=10,d=2), (4) Syn-MED (L=5,X/A=3,d=5), and (5) Syn-HIGH (L=5,X/A=1,d=10).

For Syn-IND, the tuple existence probabilities were chosen uniformly at random from [0, 1]. Note that the Syn-XOR dataset, with height set to 2 and no $\wedge$ nodes, exhibits only mutual exclusivity correlations (mimicking the x-tuples model [33, 38]), whereas the latter three datasets exhibit increasingly more complex correlations.

[2] http://nsidc.org/data/g00807.html

8. CONCLUSIONS

In this paper we presented a unified framework for ranking over probabilistic databases, and presented several novel and highly efficient algorithms for answering top-k queries. Considering the complex interplay between probabilities and scores, instead of proposing a specific ranking function, we propose using two parameterized ranking functions, called PRF^{ω} and PRF^{e}, which allow the user to control the tuples that appear in the top-k answers. We developed novel algorithms for evaluating these ranking functions over large, possibly correlated, probabilistic datasets. We also developed an approach for approximating a ranking function using a linear combination of PRF^{e} functions thus enabling highly efficient, albeit approximate computation, and also for learning a ranking function from user preferences. Our work opens up many avenues for further research that we are planning to pursue. For instance, there may be other non-trivial subclasses of PRF functions, aside from PRF^{e}, that can be computed very efficiently. Understanding the behavior of various ranking functions and their relationships across probabilistic databases with diverse uncertainties and correlation structures also remains an important open problem in this area.

Acknowledgements: This work was supported in part by NSF under Grants CCF-0728839 and IIS-0546136.

9. REFERENCES

[1] E. Adar and C. Re. Managing uncertainty in social networks. *IEEE Data Eng. Bull.*, 2007.

[2] Periklis Andritsos, Ariel Fuxman, and Renee J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.

[3] G. Beylkin and L. Monzon. On approximation of functions by exponential sums. *Applied and Computational Harmonic Analysis*, 19:17–48, 2005.

[4] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.

[5] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, 2003.

[6] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *ICDE*, 2009.

[7] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[8] Nilesh Dalvi and Dan Suciu. Management of probabilistic data: Foundations and challenges. In *PODS*, 2007.

[9] O. Dekel, C. Manning, and Y. Singer. Log-linear models for label-ranking. In *NIPS 16*, 2004.

[10] A. Deshpande, C. Guestrin, and S. Madden. Using probabilistic models for data management in acquisitional environments. In *CIDR*, 2005.

[11] Xin Luna Dong, Alon Y. Halevy, and Cong Yu. Data integration with uncertainty. In *VLDB*, 2007.

[12] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, 2001.

[13] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *SODA*, 2003.

[14] Jensen Finn and Jensen Frank. Optimal junction trees. In *UAI*, pages 360–366, 1994.

[15] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. on Info. Syst.*, 1997.

[16] Minos Garofalakis and Dan Suciu, editors. *IEEE Data Engineering Bulletin Special Issue on Probabilistic Data Management*. March 2006.

[17] Todd Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[18] Todd Green and Val Tannen. Models for incomplete and probabilistic information. In *EDBT*, 2006.

[19] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, 2006.

[20] R. Herbrich, T. Graepel, P. Bollmann-Sdorra, and K. Obermayer. Learning preference relations for information retrieval. In *ICML-98 Workshop: Text Categorization and Machine Learning*, page 83–86, 1998.

[21] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *SIGMOD*, 2008.

[22] I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 2008.

[23] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.

[24] J.F.Hauer, C.J. Demeure, and L.L.Scharf. Initial results in prony analysis of power system response signals. *IEEE Transactions on Power Systems*, 5(1):80–89, 1990.

[25] C. Jin, K. Yi, L. Chen, J. Xu Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. In *VLDB*, 2008.

[26] T. Joachims. Optimizing search engines using click-through data. In *Proc. ACM SIGKDD*, pages 133–142, 2002.

[27] C. Koch. MayBMS: A System for Managing Large Uncertain and Probabilistic Databases. Chapter in *Managing and Mining Uncertain Data*, C. Aggarwal ed., Springer, 2009.

[28] C. Koch and D. Olteanu. Conditioning Probabilistic Databases. In *VLDB*, pages 313–325, 2008.

[29] L. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *TODS*, 1997.

[30] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *CoRR*, abs/0904.1366, 2009.

[31] Jian Li and Amol Deshpande. Consensus answers for queries over probabilistic databases. In *PODS*, 2009.

[32] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.

[33] A. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.

[34] P. Sen, A. Deshpande, and L. Getoor. PrDB: Managing and Exploiting Rich Correlations in Probabilistic Databases. *VLDB Journal*, 2009.

[35] M. Soliman, I. Ilyas, and K. C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.

[36] P. Talukdar et al. Learning to Create Data-Integrating Queries. In *VLDB*, 2008.

[37] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[38] K. Yi, F. Li, D. Srivastava, and G. Kollios. Efficient processing of top-k queries in uncertain databases. In *ICDE*, 2008.

[39] Xi Zhang and Jan Chomicki. On the semantics and evaluation of top-k queries in probabilistic databases. In *DBRank*, 2008.