Ranking and Clustering in Probabilistic Databases

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

December 11, 2008

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 address the problem of on-the-fly clustering and ranking over probabilistic databases. We begin with a systematic exploration of ranking 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 \( PRF^w \) and \( PRF^e \), that can approximate many of the previously proposed ranking functions. We present several novel algorithms for efficient computing such ranking functions using generating functions, even over databases that exhibit complex correlation patterns modeled using probabilistic and/or trees or Markov networks. We further propose that the parameters of the ranking function be learned from user preferences, and develop an approach to learn such parameters.

We also develop a hierarchical framework for efficiently combining on-the-fly clustering and ranking (called a ClusterRank query) over probabilistic databases. Our framework is based on a general definition of clustering, called restricted soft-t clustering, where a tuple is allowed to participate in at most \( t \) clusters. We show how several of our ranking functions can be seamlessly integrated into this framework, which not only allows ranking to continue in parallel with clustering, but also enables pruning of a large portion of the search space. Finally, we present a comprehensive experimental study comparing different ranking functions, and illustrating the effectiveness of our clustering framework.

1 Introduction

Recent years have seen a dramatic rise in the number of applications domains that naturally generate uncertain data and that demand aid for complex, decision-support queries over them. In part, this has been due to the increasing prevalence of applications such as information retrieval [18], data integration and cleaning [2, 14], text analytics [27, 22], social network analysis [1] etc. 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 [13, 8].

By their very nature, many of these applications require support for on-the-fly clustering and ranking over large-scale datasets. We illustrate this with an example.

Example (House Search): Consider a user Amy searching for a house in Washington DC, from a database House(id, price, size, zipcode, longitude, latitude, ...). Considering the inherently fuzzy nature of the search process, issuing a ClusterRank query [31], whereby the houses are first clustered on the location and then ranked according to the criteria provided by the user, would be the most appropriate way to explore these houses; that way Amy and her husband can parallelize the house search. Figure 1 (i) shows the result of this query over an example dataset; the size of the circle representing the house denotes its score (larger circles indicate more preferred houses).

However a house database, which may be constructed by crawling and combining data from multiple sources, is inherently uncertain and noisy. In fact, the houses that Amy 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. In Figure 1, we indicate these probabilities by the size of the horizontal bar just above the location. Such uncertainty affects both ranking and clustering. For instance, the houses ranked 1 and 2 in Cluster 1 should actually be ranked lower, since they have very low probabilities of still being on market. Similarly, it might be preferable to allow the clusters to be enlarged and to allow them to overlap, so that the top-ranked houses in a cluster are more likely to be still on market. Figure 1 (ii) shows a possible result of the ClusterRank query after taking such uncertainties into account (along with the scores). The resulting clusters and Top-k tuples in each cluster would be significantly more useful to Amy than the results obtained by ignoring the uncertainties.

In this paper, we address efficient execution of a ClusterRank query in probabilistic databases. Prior work on ranking in probabilistic databases (which we review in the next section) has proposed many different functions for combining the scores and the probabilities. We begin with a systematic exploration of these issues by recognizing that probabilistic ranking 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 then define a general and powerful ranking function, called \( \text{PRF} \), 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 \( \text{PRF}^w \) and \( \text{PRF}^m \), as being interesting. The \( \text{PRF}^w \) ranking function is essentially a linear, weighted ranking function that resembles the scoring functions typically used in information retrieval, search etc. \([23, 29, 7, 12]\). We observe that such a function may not be appropriate for probabilistic databases and instead propose \( \text{PRF}^m \), which uses a single parameter, and can approximate previously proposed ranking functions for probabilistic databases very well.

We then develop novel algorithms based on generating functions to efficiently execute Top-k queries using the \( \text{PRF} \) ranking function over a probabilistic database. 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 uncertainties (called and/or probabilistic databases). 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. \([37]\)). Finally, we consider the problem of clustering over a probabilistic relation, and present a framework for efficient, uncertainty-aware clustering over large datasets. 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 over those features.
- 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 \( O(n) \) if the dataset is pre-sorted by score) for evaluating a \( \text{PRF}^m \) function over datasets with low correlations.
We define the notion of, and present an iterative algorithm for, restricted soft-t clustering that is especially suited for correlated probabilistic databases, since it allows control over the overlap between the clusters.

We present an algorithmic framework for the ClusterRank query. We provide a memory-efficient one-pass streaming algorithm for the expected score ranking function, and a two-pass algorithm for PRF*.

Finally we present an experimental study over several real and synthetic datasets, evaluating the ranking functions and the clustering framework.

**Outline:** We begin with a discussion of the related work in Section 2. We then present our algorithms for ranking (Section 4) and clustering (Section 5) in probabilistic databases. We present our framework for ClusterRank in Section 6 and 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. [4, 30, 20, 18, 6, 8, 11, 36]). With a rapid increase in the number of application domains such as data integration, pervasive computing etc., where uncertain data arises naturally, this area has seen renewed interest in recent years [19]. 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 [11], Trio [36], ORION [8], MayBMS [3]). Much of this work has used “probability theory” as the underlying foundational basis for representing uncertainty, where the uncertainty is encoded in the form of probabilities and the operations on the uncertainty itself are done in accordance with the laws of probability theory. The probabilistic database approaches themselves can be differentiated between tuple-level uncertainty, where “existence” probabilities are attached to tuples of the database, and attribute-level uncertainty, where (possibly continuous) probability distributions are attached to attributes. The proposed approaches differ further based on whether they consider correlations or not. Most work in probabilistic databases has either assumed independence [18, 11] or has restricted the correlations that can be modeled [4, 30, 2, 33]. More recently, several approaches have been presented that allow representation of arbitrary correlations [34, 3].

The area of ranking and Top-k query processing has also seen much work in databases (see Ilyas et al. [26] 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. [37] present improved algorithms for the same ranking functions. We will review these ranking functions in detail in next section. Zhang and Chomicki [38] also present a desiderata for ranking functions. Ming Hua et al. [24, 25] recently presented a different approach called probabilistic threshold queries. Finally, Cormode et al. [9] present a semantics of ranking functions and a new ranking function called expected rank.

We note that 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.
A probabilistic and/xor tree

If

If

If

\( \mark, \lor \)

Definition 1

\( \Pr \)attribute uncertainty). The set of all possible worlds is denoted by \( PW \).

We begin with defining our model of a probabilistic database, and review some of the prior definitions of ranking.

3 Preliminaries

We use the prevalent possible worlds semantics for probabilistic databases [11]. We denote a probabilistic relation with tuple uncertainty by \( D_T \), where \( T \) denotes the set of tuples (see Section 4.5 for extensions of our algorithms to attribute uncertainty). The set of all possible worlds is denoted by \( PW = \{pw_1, pw_2, \ldots, pw_n\} \). Each tuple \( t_i \in T \) is associated with an existence probability \( \Pr(t_i) \) and a score \( s(t_i) \), computed based on a scoring function \( s : T \rightarrow \mathbb{R} \). Usually \( s \) is computed based on the tuple attribute values and measures the relative user preference for different tuples.

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 \( s \). If \( t \) does not appear in the possible world \( pw \), then \( r_{pw}(t) = \infty \). We say \( t_1 \) ranks higher than \( t_2 \) in possible world \( pw \) if \( r_{pw}(t_1) < r_{pw}(t_2) \). For ease of notation, we will use \( t \in pw \) in place of “\( t \) appears in the possible world \( pw \)”.

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: Although our algorithms can efficiently handle arbitrarily correlated relations with correlations modeled using Markov networks [34], in this paper we focus on two common types of correlations: (1) mutual co-existence and (2) mutual exclusivity. Two tuples satisfy the mutual co-existence correlation if, in any possible world, either both tuples occur or neither occurs. Similarly two tuples are mutually exclusive if there is no possible world that contains both tuples. We model such correlations using what we call a probabilistic and/xor tree, which generalizes the notion of \( x \)-tuples [33, 37]. For ease of notation, we use \( \oplus \) (or) to denote mutual exclusion and \( \otimes \) (and) for coexistence.

Definition 1

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

- If \( v \) is a leaf, \( S_v = \{v\} \).
- If \( T_v \) roots at a \( \otimes \) node, then \( S_v = \left\{ \begin{array}{ll} S_{v_i} & \text{with prob. } p(v, v_i) \\ \emptyset & \text{otherwise} \end{array} \right. \)
- If \( T_v \) roots at a \( \oplus \) node, then \( S_v = \bigcup_i S_{v_i} \)

Thus, a possible world is completely defined by a set of choices for the \( \otimes \) nodes in the tree.

Figure 2 shows an example probabilistic relation that models the data from a traffic monitoring application [35], where the tuples represent automatically captured traffic data. The inherent uncertainty in the monitoring infrastructure is captured using an and/xor tree, that encodes the tuple existence probabilities as well as the correlations between the tuples. For example, the leftmost \( \otimes \) node indicates \( t_1 \) will be present with probability \( A \) and the second \( \otimes \) node dictates that only one of \( t_2 \) and \( t_3 \) will appear. The topmost \( \otimes \) node tell us the resultant sets derived from these \( \otimes \) nodes coexist.
Ranking over uncertain data: 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:

- **Uncertain Top k (UTop-k):** 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 (URank-k):** at each rank i, we return the tuple with the maximum probability of being at i’th rank in all possible worlds. In other words, URank-k returns \{t_i^*, i = 1, 2, ..., k\}, where \(t_i^* = \text{argmax}_t(Pr(r(t) = i))\).

Several other definitions of ranking have been proposed since then. Hua et al. [24] propose the notion of probabilistic threshold top-k (PT-k) queries: here all tuples with sufficiently high probability of being in top-k are returned as the answer (i.e., all tuples \(t\) such that \(\sum_{i=1}^{k} Pr(r(t) = i) > \text{threshold}\)). Recently, Cormode et al. [9] proposed using expected ranks (Ex-Rank), where the tuples are ranked in the increasing order by: \(\sum_{pw} Pr(pw) \times r_{pw}(t)\), where \(r_{pw}(t)\) is defined to be \(|pw|\) if \(t \notin pw\).

Finally, another natural ranking function, also considered by [9], is expected score (E-Score), where we rank the tuples by their expected score, \(Pr(t) \cdot s(t)\).

### 4 PRF 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. We believe this has led to the proposal of many different and conflicting ranking functions over last few years. Instead of proposing a single ranking function, we instead propose a family of ranking functions, parameterized by one or more parameters, and design algorithms to efficiently evaluate 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. We begin with defining these new ranking functions, and then present algorithms for efficiently evaluating them.

#### 4.1 Definitions

The tradeoffs in ranking over uncertain tuples can be seen with a simple example. Consider a database with two tuples \(t_1\) (score = 100, \(p(t_1) = 0.5\)), and \(t_2\) (score = 50, \(p(t_2) = 1.0\)). 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 tradeoff, and the choice between these two options largely depends on the user preferences.

We propose to follow the traditional approach to dealing with such tradeoffs, by identifying a set of features, by using a scoring function over these features, and by learning the scoring function itself using user preferences [23, 29, 7, 12]. 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. This set of features succinctly captures the possible worlds. Further, correlations among tuples, if any, are naturally accounted for when computing 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.

#### 4.2 New Ranking Functions

Next we define a general ranking function which allows exploring the tradeoffs discussed above.
Definition 2 Let $f : T \times \mathbb{N} \rightarrow \mathbb{R}$ be some function which we call weight function. The probabilistic ranking function (PRF) $\Upsilon : T \rightarrow \mathbb{R}$ in its most general form is defined to be:

$$\Upsilon(t) = \sum_{pw \in pw} f(t, r_{pw}(t)) \cdot Pr(pw) = \sum_{i>0} f(t, i) \cdot Pr(r(t) = i)$$

A Top-k query will return $k$ tuples with the highest $\Upsilon$ values.

Depending on the actual function $f$, we get different ranking functions with diverse behaviors. We illustrate some of these choices and relate them to prior ranking functions\footnote{The definition of the U-topK 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-topK using PRF.}.

- If $f(t, i) = 1$, the result is the set of $k$ tuples with the highest probabilities [32].
- By setting $f(t, i) = s(t)$, we get the expected score ranking function:
  $$\Upsilon(t) = \sum_{pw \in pw} s(t)Pr(pw) = s(t)Pr(t) = E(s(t))$$
- **PRF**$_{w}(w_1, \ldots, w_n)$: One important class of ranking functions is when $f(t, i) = w_i$, where $w_i$ are (constant) weights (typically $w_i = 0, \forall i > k$ for some small $k$). 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 [23, 29, 7, 12]. We revisit the learning aspect below (Section 4.6).

**Two special cases of the PRF**$_{w}$ function are:

1. $f(i) = \begin{cases} 1, & i=j \\ 0, & \text{otherwise} \end{cases}$ for some $1 \leq j \leq k$. We can see the tuple with largest $\Upsilon$ value is the rank-$j$ answer in URank-$k$ query [35]. This allows us to compute the URank-$k$ answer by executing our algorithm for $j = 1, \ldots, k$.
2. $f(i) = \begin{cases} 1, & i \leq k \\ 0, & \text{otherwise} \end{cases}$. If we return all tuples with $\Upsilon(t) > \delta$, where $\delta$ is a pre-specified threshold, we have exactly the answer for Probabilistic Threshold Queries (PT-$k$) [25].

- **PRF**$_{c}(a)$: Finally, we define **PRF**$_{c}$ to be a special case of the **PRF**$_{w}$ function, where $w_i = f(i) = a^i$, where $a$ is a constant and may be a real or a complex number.

**PRF**$_{w}$ and **PRF**$_{c}$ form the two parameterized ranking functions that we propose in this work. Although **PRF**$_{w}$ is the more natural ranking function and has been used elsewhere, **PRF**$_{c}$ 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 feature $Pr(r(t) = i)$ is clearly more important than the feature $Pr(r(t) = i + 1)$ for all $i$. Hence we would like the weight function, $f(i)$, to be monotonically decreasing. **PRF**$_{c}$ naturally captures this behavior (as long as $|a| < 1$). More importantly, we can compute the **PRF**$_{c}$ 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. This makes it significantly more attractive for ranking over large datasets.

Further we note that, a linear combination of **PRF**$_{c}$ functions, with complex bases, is known to be very expressive, and can approximate many other functions very well [5]. Specifically, if we can write $f(i)$ as: $f(i) \approx \sum_{i=1}^{L} u_i a_i^i$, then we have that:

$$\Upsilon(t) = \sum_{i} f(i)Pr(r(t) = i) = \sum_{i=1}^{L} u_i \left( \sum_{i} a_i^i Pr(r(t) = i) \right)$$

This reduces the computation of $\Upsilon(t)$ to $L$ individual **PRF**$_{c}$ function computations. Since $L$ is typically quite small, this gives us a very efficient algorithm for evaluating complex ranking functions.

### 4.3 Computing a PRF Function

We next present an algorithm for efficiently computing the **PRF** function. We first present the basic idea behind our algorithm assuming mutual independence, and then consider correlated tuples.
Consider a relation with 3 independent tuples $t_1, t_2, t_3$. Let $T = \{x, y\}$ be the and/xor tree representation of the independent tuples in Example 1.

### 4.3.1 Assuming Tuple Independence

First we show how the PRF function can be computed in $O(n^2)$ time for general $f$. In all our algorithms, we assume that $f(t, i)$ can be computed in $O(1)$ time. Clearly it would be sufficient to compute $Pr(r(t) = j)$ for any tuple $t$ and $1 \leq j \leq n$ in $O(n^2)$ time. Given these values, we can directly compute the values of $\Upsilon(t)$ in $O(n^2)$ time (later, we will present several algorithms which run in $O(n)$ or $O(n \log(n))$ time which combine these two steps for some special $f$ functions).

We first sort the tuples in a non-increasing order by their score function, say $T = \{t_1, t_2, \ldots, t_n\}$. Suppose now we want to compute $Pr(r(t_i) = j)$. Let $T_i = \{t_1, t_2, \ldots, t_i\}$ and $\sigma_i$ be an indicator variable that takes value 1 if $t_i$ is present, and value 0 if $t_i$ is not present. Further, let $\sigma = (\sigma_1, \ldots, \sigma_n)$ denote a vector containing all the indicator variables. Then, we can rewrite $Pr(r(t_i) = j)$ as:

$$Pr(r(t_i) = j) = Pr(t_i) \sum_{pw:|pw| = |T_{i-1}| = j-1} Pr(pw)$$

$$= Pr(t_i) \sum_{\sigma_i} \prod_{l=1}^{i-1} Pr(t_l) \prod_{l<i, \sigma_l = 1} (1 - Pr(t_l))$$

The second formula essentially iterates over all possible worlds. The naive method to evaluate the above formula by explicitly listing all possible worlds needs exponential time. Now, we turn to the generating function method based on which a polynomial time algorithm can be obtained. Recall the coefficient of $x^k$ in $\prod_{i=1}^n (a_i + b_i x)$ is $\sum_{\beta = 0}^k \prod_{i: \beta_i = 0} a_i \prod_{i: \beta_i = 1} b_i$ where $\beta$ is a boolean vector of length $n$ and $1 = \{1, 1, \ldots, 1\}$.

Consider the following generating function:

$$F^j(x) = \left( \prod_{t \in T_{i-1}} (1 - Pr(t) + Pr(t) \cdot x) \right) (Pr(t_i) \cdot x)$$

$$= \sum_{j \geq 0} c_j x^j.$$  

It can be easily seen that the coefficient $c_j$ of $x^j$ in the expansion of $F^j$ is exactly the probability that $t_j$ is at rank $j$, i.e., $Pr(r(t_i) = j)$. Expanding $F^j$ can be easily done in $O(i^2)$ time. This allows us to compute $Pr(r(t_i) = j)$ for $t_i$ in $O(i^2)$ time; $\Upsilon(t_i)$, in turn, can be written as:

$$\Upsilon(t_i) = \sum_j f(j) \cdot Pr(r(t_i) = j) = \sum_i f(j)c_i$$

which can be computed in $O(i^2)$ time.

**Example 1** Consider a relation with 3 independent tuples $t_1, t_2, t_3$ (already sorted according to the score function) with existence probabilities 0.5, 0.6, 0.4, respectively. The generating function for $t_3$ is:

$$F^3(x) = (.5 + .5x)(.4 + .6x)(.4x) = .2x + .5x^2 + .3x^3$$

This gives us:

$$Pr(r(t_3) = 1) = .2, Pr(r(t_3) = 2) = .5, Pr(r(t_3) = 3) = .3$$
Let have that:

\[ Pr_x \] shown to be the coefficient of the term \( x \)

Algorithm 1

```
Algorithm 1 IND-PRF-RANK(D_T)
\[
\mathcal{F}^0(x) = 1
\]
for i=1 to n do
\[
\mathcal{F}^i(x) = \frac{Pr(t_i)}{Pr(t_{i-1})} \mathcal{F}^{i-1}(x)(1 - Pr(t_{i-1}) + Pr(t_{i-1})x)
\]
Expand \( \mathcal{F}(x) \) in the form of \( \sum c_j x^j \)
\[
\Upsilon(t_i) = \sum_{j=1}^{n} f(t_i, j) c_j.
\]
end for
return k tuples with largest \( \Upsilon \) values.
```

If we expand each \( \mathcal{F}^i \) for \( 1 \leq i \leq n \) from scratch, we will need \( O(n^3) \) time in total. However, the expansion of \( \mathcal{F}^i \) can be easily obtained from the expansion of \( \mathcal{F}^{i-1} \) in \( O(i) \) time as follows:

\[
\mathcal{F}^i(x) = \frac{Pr(t_i)}{Pr(t_{i-1})} \mathcal{F}^{i-1}(x)(1 - Pr(t_{i-1}) + Pr(t_{i-1})x).
\] (2)

This trick gives us a \( O(n^2) \) time complexity. Algorithm 1 illustrates the pseudocode. Note that \( O(n^2) \) time is asymptotically optimal in general since the computation involves at least \( O(n^2) \) probabilities, namely \( Pr(r(t_i) = j) \) for all \( 1 \leq i, j \leq n \).

However, for specific \( f \) functions, we may be able to achieve faster computation.

For example, if \( f(t, i) = \begin{cases} 1, & i = j, \\ 0, & \text{otherwise} \end{cases} \), we only need to expand all \( \mathcal{F}^i \)’s up to \( x^k \) term. This yields an \( O(nk) \) time algorithm for answering \( Top-k \) query, thus matching the best known upper bound by Yi et al. [37] (The original algorithm in [35] runs in \( O(n^2k) \) 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 \( PRF^e \) function (Section 4.4).

### 4.3.2 Handling Mutual Exclusion and Coexistence

Next we generalize our algorithm to handle a correlated database where the correlations can be captured using an and/or 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 \( T \) denote the and/or tree that models the correlations.

Suppose now we need to compute \( Pr(r(t_i) = j) \). Since we don’t care about any tuple with a smaller score, it suffices to consider only \( T_i \), the subtree of \( T \) induced by the leaf set \( T_i \) (namely, the union of all root-leaf paths with all leaves in \( T_i \)). Let \( T_i^v \) be the subtree rooted at \( v \) and \( v_1, \ldots, v_l \) be \( v \)’s children. For each node \( v \in T_i \), we define a generating function \( \mathcal{F}^i_v(x, y) \) inductively as follows:

- If \( v \) is a leaf, \( \mathcal{F}^i_v(x, y) = \begin{cases} x, & v \in T_i \setminus \{t_i\}; \\ y, & v = t_i. \end{cases} \)

- If \( v \) is a \( \bigcirc \) node,
\[
\mathcal{F}^i_v(x, y) = (1 - \sum_{h=1}^l p(v, v_h)) + \sum_{h=1}^l \mathcal{F}^i_{v_h}(x, y) \cdot p(v, v_h)
\]

- If \( v \) is a \( \bigotimes \) node, \( \mathcal{F}^i_v(x, y) = \prod_{h=1}^l \mathcal{F}^i_{v_h}(x, y). \)

The generating function \( \mathcal{F}^i \) for \( T_i \) is the generating function of its root. The probability \( Pr(r(t_i) = j) \) can be shown to be the coefficient of the term \( x^{j-1}y \) in \( \mathcal{F}^i(x, y) \).

**Theorem 1** Let \( c_j \) be the coefficient of the term \( x^{j-1}y \) in the generating function \( \mathcal{F}^i(x, y) \) defined by Algorithm 2. We have that: \( Pr(r(t_i) = j) = c_j. \)
Proof: Suppose $T^i$ is rooted at $r$, $r_1, \ldots, r_h$ are $r$’s children and $T^i_i$ is the subtree rooted at $r_i$. W.l.o.g, we assume $t_i \in T^i_i$. We denote by $S^i (S)$ the random set of leaves generated according to model $T^i$ ($T^i_i$). We write $F^i (x, y) = \sum_j c_j x^j + (\sum_j c_j x^j) y$ We prove by induction on the height of the and/xor tree the following claims: $Pr(|S^i| = j \land t_i \in S^i) = c_j$ and $Pr(|S^i| = j \land t_i \notin S^i) = c_j$. We consider two case. If $r$ is a $\oplus$ node, we know from Definition 1 $S^i = \cup_{t=1}^h S^i_i$. Therefore,

$$Pr(|S^i| = j \land t_i \in S^i) = Pr(\sum_{i=1}^h |S^i_i| = j \land t_i \in S^i)$$

$$= \sum_{\sum_{i=1}^h i = j} \prod_{j=1}^h Pr(|S^i_j| = j) Pr(|S^i_h| = j \land t_i \in S^i_h)$$

Let $F^i_i = \sum_j c_j^i x^j + (\sum_j c_j^i x^j) y$ be the polynomial defined by $T^i_i$. Now let us consider the coefficient $c_j$ of the term $x^j y$ in $F^i (x, y) = \prod_{j=1}^h F^i_j (x, y)$. From the Algorithm 3, we know the degree of $y$ in $F^i_i$ is 0 for $1 \leq l \leq h - 1$. So only the terms in $F^i_i$ can contribute to the $y$ terms in $F^i$. Therefore, $c_j = \sum_{\sum_{i=1}^h j = j} \prod_{i=1}^h c_j^i c_j h$. Then it is easy to see, from induction hypothesis, the theorem follows. The other case when $r$ is a $\ominus$ node can be proved by simply observing $Pr(|S^i| = j \land t_i \in S^i) = Pr(|S^i_j| = j \land t_i \in S^i) p(r, r_h)$ and the coefficient $c_j$ of the term $x^j y$ in $F^i$ is $p(r, r_h) c_j h$.

Example 2 The generating function $F^5$ for the left hand side tree in Figure 5 is $(.6 + Ax)x(Ax + .6y) = .24x^2 + .16x^3 + .36xy + .24x^2 y$. So $Pr(p(t_3) = 3) = .24$ for example. From Figure 2, we can also see $Pr(p(t_3) = 3) = Pr(pw_2) + Pr(pw_4) = .24$ which is the same as we obtained. The right hand side of Figure 5 shows the probabilistic and/xor tree and the generating function computation for Ex. 1.

See Algorithms 2 and 3 for the pseudocode of the algorithm. If we expand $F^i_x$ for each internal node $v$ in a naive way (do polynomial multiplication one by one), we can show 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 Appendix B.

4.3.3 Correlations captured using a Graphical Model

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. Our generating function-based algorithm for computing PRF can be extended 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. Specifically our algorithm can compute the PRF function in time $O(2^{tw} n^2 (2^{tw} + n)|T|)$, where $tw$ is the treewidth of the graphical model (size of the largest clique in the junction tree) and $|T|$ (which is always $\leq n$) is the number of cliques in the junction tree. In some sense, this is close to instance-optimal since the underlying inference problem is also exponential in the treewidth (this does not preclude the possibility that the ranking itself could be done more efficiently without computing the PRF function explicitly – however, such an algorithm is unlikely to exist). Please see Appendix C for the algorithm.

It is worth noting that this result can not subsume our algorithm in the last section since the treewidth of the moralized graph of a probabilistic and/xor tree may not be bounded.
Algorithm 2 PROB-ANDOR-PRF-RANK(T)

\[ T^0 = \emptyset \]

\[ \text{for } i=1 \text{ to } n \text{ do} \]
\[ T_i \text{ is obtained by overlaying } T_{i-1} \text{ and the path from } t_i \text{ to the root.} \]
\[ F^i(x, y) = GENE(T^i, t_i) \]
\[ \text{Expand } F^i(x, y) \text{ in the form of } \sum_j c_j x_j + (\sum_j c_j x_j^{j-1}) y \]
\[ \Upsilon(t_i) = \sum_{j=1}^n f(t_i, j) c_j. \]
\[ \text{end for} \]
\[ \text{return } k \text{ tuples with largest } \Upsilon \text{ values.} \]

Algorithm 3 GENE(T, t)

\[ \text{if } T \text{ is a singleton node then} \]
\[ \text{if } T = \{t\} \text{ then return } y \text{ else return } x \]
\[ \text{else} \]
\[ \{r_1 \ldots r_l \text{ are } r \text{'s children. } p = \sum_{i=1}^l p(r, r_i), T_i \text{ is the subtree rooted at } r_i \} \]
\[ \text{if the root } r \text{ of } T \text{ is a } \square \text{ node then} \]
\[ \text{return } 1 - p + \sum_{i=1}^l p(r, r_i) \cdot GENE(T_i, t) \]
\[ \text{end if} \]
\[ \text{if the root } r \text{ of } T \text{ is a } \bigcirc \text{ node then} \]
\[ \text{return } \prod_{i=1}^l GENE(T_i, t) \]
\[ \text{end if} \]
\[ \text{end if} \]

4.4 Computing PRFe function

Finally, we present an \(O(n \log(n))\) algorithm to evaluate a \(PRF^e\) function (in linear time if the dataset is pre-sorted by score). Consider the Equation 1. If \(f(i) = a^i\), it is easy to see:

\[ \Upsilon(t_i) = F^i(a). \quad (3) \]

This surprisingly simple relationship suggests we don’t have to expand the polynomial at all; instead we can evaluate the numerical value of \(F^i(a)\) directly. Again, we note that the value \(F^i(a)\) can be computed from the value of \(F^{i-1}(a)\) in \(O(1)\) time using Equation 2. Thus, we have \(O(n)\) time algorithm to compute \(\Upsilon(t_i)\) for all \(1 \leq i \leq n\) if the tuples are pre-sorted.

Example 3 Consider Example 1 and the \(PRF^e\) function for \(t_3\). We choose \(f(i) = .6^i\). Then, we can see that \(F^3(x) = (.5 + .5x)(A + .6x)(Ax)\). So, \(\Upsilon(t_3) = F^3(.6) = (.5 + .5 \times .6)(.4 + .6 \times .6)(.4 \times .6) = .14592\).

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 \(F^i(x, y) = \sum_j c_j x_j + (\sum_j c_j x_j^{j-1}) y \) and \(\Upsilon(t_i) = \sum_{j=1}^n a^j c_j\). We observe an intriguing relationship between the \(PRF^e\) value and the generating function:

\[ \Upsilon(t_i) = F^i(a, a) - F^i(a, 0). \]

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

By carefully sharing the intermediate results among computations of \(\Upsilon(t_i)\), we can improve the running time \(O(n \log(n) + nd)\) 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 value of \(F^i_v(a, a)\) and \(F^i_v(a, 0)\). 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 \( t_1 \) 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 \( \mathcal{F}^i_v(\ldots) \) in node \( v \) is as follows. We assume \( v \)'s child, say \( u \), just had its values changed.

1. \( v \) is a \( \sqcap \) node, \( \mathcal{F}^i_v(\ldots) \leftarrow \mathcal{F}^{i-1}_v(\ldots) \mathcal{F}^i_u(\ldots) / \mathcal{F}^{i-1}_u(\ldots) \)

2. \( v \) is a \( \sqcup \) node, then:
   \[
   \mathcal{F}^i_v(\ldots) \leftarrow \mathcal{F}^{i-1}_v(\ldots) + p(v, u) \mathcal{F}^i_u(\ldots) - p(v, u) \mathcal{F}^{i-1}_u(\ldots)
   \]

We note that, for the case of \( x \)-tuples, which can be represented using a two-level tree with the root being a \( \sqcap \) node and other internal nodes being \( \sqcup \) nodes, this gives us an \( O(n \log(n)) \) algorithm for computing \( \text{PRF}^e \).

### 4.5 Attribute Uncertainty and 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 attribute does not affect the tuple score, then it 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 \( \text{xor} \) constraint over the alternatives. We can then use the algorithm for the probabilistic and/or 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 (this may involve a sort operation). 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 \( \text{PRF}^e \) function.

### 4.6 Learning \( \text{PRF}^w \) or \( \text{PRF}^e \) Functions

Finally we address the question of how to learn the \( \text{PRF}^w \) or the \( \text{PRF}^e \) functions from user preferences. Prior work on learning ranking functions (e.g., [23, 29, 7, 12]) 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 tuples \( (Pr(r(t) = i), i = 1, \ldots, n) \) cannot be computed for each tuple individually, but must be computed for the entire 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 \( \text{PRF}^w \) or the parameter \( a \) for \( \text{PRF}^e \)) that minimizes the number of disagreements with the provided ranking over the samples.

Then, the problem of learning \( \text{PRF}^w \) is identical to the problem addressed in the prior work, and we utilize the algorithm based on support vector machines [29] in our experiments.

We are not aware of any work that has addressed learning a ranking function like \( \text{PRF}^e \). We use a simple binary search-like algorithm to find the optimal value of \( a \) for our experiments. Designing algorithms for learning \( \text{PRF}^e \) functions (and more generally learning linear combinations of \( \text{PRF}^e \) functions) is beyond the scope of this work.

### 5 Clustering in Probabilistic Databases

In this section we consider the problem of clustering a probabilistic dataset. We begin with defining the notion of restricted soft-t clustering which allows overlap among the resulting clusters. This is key in uncertain databases where restricting a tuple to belong to a single cluster may result in clusters with a large number of low-probability tuples, decreasing the utility of such clusters. Restricted soft-t clustering (RSC-t) generalizes both hard clustering (e.g. using \( K \text{ means} \)) and soft clustering (e.g. using \( \text{Fuzzy-c means} \)) and allows the exploration of the options in between. We then present an iterative algorithm for RSC-t.
5.1 Restricted Soft-t Clustering

Given the database $D = \{x_1, x_2, \ldots\}$, where $x_i$ is the $i$-th $d$-dimensional tuple, the optimization problem for RSC-$t$ can be written as minimizing:

$$J_D(u, c) = \sum_{i \in D} \sum_{j=1}^{c} u_{i,j}^m ||x_i - c_j||^2,$$

$$1 < m \leq \infty$$

Subject to RSC-$t$ Constraints:

$$\sum_{j=1}^{c} u_{i,j} = 1 \ \forall i, \quad u_{i,j} y_{i,j} = u_{i,j} \ \forall i, j$$

$$\sum_{j=1}^{c} y_{i,j} = t \ \forall i, \quad u_{i,j} \geq 0, \quad y_{i,j} \geq 0 \ \forall i, j$$

Here $m$ is a fuzziness parameter, $u_{i,j}$ is the degree of membership of $x_i$ in the cluster $j$, $y_{i,j}$ helps in restricting the number of clusters $x_i$ may belong to, $c_j$ is the $d$-dimensional center of cluster and $|| \cdot ||$ is any norm for measuring similarity between the measured data and the cluster center. Note that the objective function (5) of RSC-$t$ is same as the objective function of fuzzy-$c$-means; however, the constraint formulation (5) differs. The following theorem proves the correctness of the RSC-$t$ formulation.

**Theorem 2** The constraint formulation (5) satisfies the condition:

$$\forall x_i \in D, \left[\{u_{i,j}, \ s.t. \ u_{i,j} > 0, \ j = 1, 2, \ldots, c\}\right] \leq t$$

**Proof:** Suppose $x_i$ belongs to $t + a$ clusters, for some $a \geq 1$. Let without loss of generality, those clusters be $c_1, c_2, \ldots, c_{t+a}$. Therefore we have $u_{i,1}, u_{i,2}, \ldots, u_{i,(t+a)}$ all $> 0$. Hence to satisfy the constraint (2), we must have $y_{i,1} = y_{i,2} = \ldots = y_{i,(t+a)} = 1$. Hence $\sum_{j=1}^{c} y_{i,j} = t + a > t$. Hence the constraint (3) is violated. Therefore no data point can belong to more than $d$ clusters. The constraints do not restrict a point to belong to exactly $t$ clusters. A point $x_i$ may belong to $l$ clusters for any $l \leq t$. To see this, suppose, without loss of generality, $x_i$ belongs to the first $l$ clusters. Then $u_{i,j}$ values for $j = 1, 2, \ldots, l$ are nonzero. Hence we must have $y_{i,j}$ for $j = 1, 2, \ldots, l$ to be nonzero. Set $y_{i,l+1} = t - l$ and the rest of the $y_{i,j}, j > l + 1$ to be 0. We see all the constraints are satisfied.

As with $K$-means, objective function of RSC-$t$ can be minimized by maintaining the constraints (5) in an iterative fashion. The update procedure for membership values and the cluster centers at every iteration, that guarantees the convergence to a local minima is given by the following theorem.

**Theorem 3** Let $c_1, c_2, \ldots, c_t$ be the $t$ nearest centers for $i \in D_T$ in an iteration. The iterative solution for the RSC-$t$ constraint optimization problem, which converges to a local minima is given by:

$$u_{i,j} = \begin{cases} \frac{1}{\sum_{k=1}^{t} \left(||x_i - c_k||^2\right)^{\frac{m-1}{2}}} & \text{if } j = i, \ 1 \leq l \leq t \\ 0 & \text{Otherwise.} \end{cases}$$

$$c_j = \frac{\sum_{i=1}^{n} u_{i,j}^m x_i}{\sum_{i=1}^{n} u_{i,j}^m}$$

We note that the iterative algorithm can only converge to a local minima (as with $K$-means) and no approximation guarantees can be provided for it (since it generalizes $K$-means).

**Proof:** Consider the Lagrangian relaxation of the objective function

$$L = \sum_{i \in T} \sum_{j=1}^{c} u_{i,j}^m ||x_i - c_j||^2 + \sum_{i \in T} \lambda_i (1 - \sum_{j=1}^{c} u_{i,j}) + \sum_{i \in T} \sum_{j=1}^{c} \mu_{i,j} (u_{i,j} - u_{i,j} y_{i,j}) + \sum_{i \in T} \gamma_i (t - \sum_{j=1}^{c} y_{i,j}).$$

12
We take partial derivative and set them to 0:

\[
\frac{\partial L}{\partial u_{i,j}} = m \cdot u_{i,j}^{m-1}||x_i - c_j||^2 - \lambda_i + \mu_{i,j} (1 - y_{i,j}) = 0. \tag{6}
\]

\[
\frac{\partial L}{\partial y_{i,j}} = -\mu_{i,j} u_{i,j} - \gamma_i = 0. \tag{7}
\]

\[
\frac{\partial L}{\partial \lambda_i} = 1 - \sum_{j=1}^{c} u_{i,j} = 0. \tag{8}
\]

\[
\frac{\partial L}{\partial \mu_{i,j}} = u_{i,j} - y_{i,j} = 0. \tag{9}
\]

\[
\frac{\partial L}{\partial \gamma_i} = t - \sum_{j=1}^{c} y_{i,j} = 0. \tag{10}
\]

If \(u_{i,j} \neq 0\), from (9) \(y_{i,j} = 1\). So, when \(u_{i,j} \neq 0\), we have from (6),

\[
u_{i,j} = \left( \frac{\lambda_i}{m \cdot ||x_i - c_j||^2} \right)^\frac{1}{m-1}.
\] 

Plugging it into (8), we have

\[
1 - \sum_{j=1, u_{i,j} \neq 0}^{c} u_{i,j} = 1 - \sum_{j=1, u_{i,j} \neq 0}^{c} \left( \frac{\lambda_i}{m \cdot ||x_i - c_j||^2} \right)^\frac{1}{m-1} = 0.
\]

So,

\[
\left( \frac{\lambda_i}{m} \right)^\frac{1}{m-1} = \frac{1}{\sum_{j=1, u_{i,j} \neq 0}^{c} \left( \frac{1}{||x_i - c_j||} \right)^2}.
\]

Plugging it back into (11), we get if \(u_{i,j} \neq 0\), then:

\[
u_{i,j} = \frac{1}{\sum_{k=1, u_{i,k} \neq 0}^{c} \left( \frac{||x_i - c_k||}{||x_i - c_j||} \right)^2}.
\]

Let the distances of the \(i\)th element from the current centers be \(d_{i,1}, d_{i,2}, \ldots, d_{i,c}\) respectively. Consider w.l.o.g, \(d_{i,1} \leq d_{i,2} \leq \ldots \leq d_{i,c}\). Let \(u_{i,j_1}, u_{i,j_2}, \ldots, u_{i,j_t}\) be > 0, \(l \leq t\) and the rest of the membership values for the \(i\)th element be 0. Let \(j_1 < j_2 < \ldots < j_t\). It is easy to see that it holds \(j_h = h\), for \(h = 1, 2, \ldots, l\), assuming \(d_{i,1} < d_{i,2} < \ldots < d_{i,c}\). Otherwise, by making \(j_h = h\), we get a lower value for the objective function-a contradiction. In general, if \(l\) of the \(u_{i,j}\) values are nonzero for \(i \in T\) and \(c_{i_1}, c_{i_2}, \ldots, c_{i_l}\) be the \(l\) nearest centers for \(i\), then only the corresponding membership values will be > 0. In a similar fashion, using contradiction, it can be proved that \(\forall i\) at every iteration, exactly \(t\) of the \(u_{i,j}\) values will be positive. This completes the prove for the update rule of \(u_{i,j}\)’s.

The updating rule for \(c_j\) can be seen from the fact that for any \(c_j\),

\[
\sum_{i=1}^{n} u_{i,j}^m ||x_i - c_j||^2 = \sum_{i=1}^{n} u_{i,j}^m ||x_i - \bar{c}_j||^2 + \sum_{i=1}^{n} u_{i,j}^m ||\bar{c}_j - c_j||^2
\]

where \(\bar{c}_j = \frac{\sum_{i=1}^{n} u_{i,j}^m x_i}{\sum_{i=1}^{n} u_{i,j}^m}\). So, if \(u_{i,j}\)’s are fixed, the above summation is minimized at \(c_j = \bar{c}_j\). \(\square\)
5.2 RSC-t in Probabilistic Databases

We extend our notion of restricted soft-t clustering to probabilistic databases by defining the objective function to minimize the expectation of the RSC-t objective function among all possible worlds. Recall that $PW(D_T)$ denotes the possible worlds space of $D_T$. Let $pw$ be a random variable (possible world) drawn from $PW(D_T)$. Let $u^{pw}$ denote the membership values of the tuples in the possible world $pw$ and $u_{i,j}^{pw}$ denote the membership of the tuple $i$ in the cluster $j$ in $pw$. If $i$ does not belong to $pw$, we can assume $u_{i,j}^{pw} = 0$. Similarly $y_{i,j}^{pw}$ denotes the corresponding $y$ value for tuple $i$ in cluster $j$ in $pw$.

Then, RSC-t in probabilistic world with tuple uncertainty can be formalized as follows:

$$\min E[J_{pw}(u^{pw}, c)]$$

s.t. $\sum_{j=1}^{c} u_{i,j}^{pw} = 1 \forall pw \in PW(D_T), i \in pw$

$u_{i,j}^{pw} y_{i,j}^{pw} = u_{i,j}^{pw} \forall pw \in PW(D_T), i \in pw, j$

$\sum_{j=1}^{c} y_{i,j}^{pw} = t \forall pw \in PW(D_T), i \in pw, j$

$u_{i,j}^{pw} \geq 0, y_{i,j}^{pw} \geq 0 \forall pw \in PW(D_T), i \in pw, j$

The next theorem allows us to develop an efficient algorithm for the above optimization problem (12). The critical observation here is that at optimum (minimum value of $E[J_{pw}(u^{pw}, c)]$), the membership values and the corresponding $y$ values of the tuple are independent of the possible world in which a tuple belongs. The proof then follows by probability marginalization.

**Theorem 4** The optimization problem of Equation 12 has the same minimum value as the optimization problem of minimizing:

$$J_{D_T}^1(u, c) = \sum_{i \in T} \Pr(i) \sum_{j=1}^{c} u_{i,j}^m ||x_i - c_j||^2, 1 \leq m \leq \infty$$

subject to the RSC-t Constraints (Equation 5), where $\Pr(i)$ is the existence probability of the tuple $i \in D_T$.

**Proof:** First, we show that if the centers $c$ are fixed, (12) can be minimized at the point such that $u^{pw}$s are the same for all possible worlds $pw$. We can see the contribution of the tuple $i$ to (12) is

$$E(\sum_{j=1}^{c} (u_{i,j}^{pw})^m ||x_i - c_j||^2) = \sum_{pw \in PW(D_T)} \Pr(pw) \sum_{j=1}^{c} (u_{i,j}^{pw})^m ||x_i - c_j||^2$$

So the $\bar{u}$ which minimizes $\sum_{j=1}^{c} \bar{u}_{i,j}^m ||x_i - c_j||$ under the restriction $\sum_{j} \bar{u}_{i,j} = 1$ can serve the optimal choice for all $u^{pw}$s. This means that the membership values of a tuple can only depend on $c$, not on which possible world it is in. So, it allows us to use the notation $\bar{u}$ instead of $u^{pw}$.

We don’t care these $u_{i,j}^{pw}$ which are not defined, i.e, the tuple $i$ doesn’t appear in $pw$, we can imagine that they could be arbitrary value. This doesn’t affect our argument.
Then by simple probability marginalization, we have

\[
E(J_{pw}(\mathbf{u}, \mathbf{c})) = \sum_{pw \in PW(D_T)} \sum_{i \in pw, j = 1}^c u_{i,j}^m ||x_i - c_j||^2 \cdot Pr(pw) \\
= \sum_{i \in T, j = 1}^c u_{i,j}^m ||x_i - c_j||^2 \sum_{pw \in PW(D_T), i \in pw} Pr(pw) \\
= \sum_{i \in T, j = 1}^c u_{i,j}^m ||x_i - c_j||^2 \sum_{i \in pw} Pr(i)Pr(pw|i \in pw) \\
= \sum_{i \in T, j = 1}^c u_{i,j}^m ||x_i - c_j||^2 Pr(i). \tag{14}
\]

We call this Probabilistic Restricted Soft-t Clustering (PRSC-t). PRSC-t can also be minimized iteratively updating the membership values and cluster centers. The updating rules are given by the following theorem. The proof of this theorem is similar to that of Theorem 3.

**Theorem 5** Let \(c_{i_1}, c_{i_2}, \ldots, c_i\) be the \(t\) nearest centers for \(i \in T\) in an iteration. The iterative solution for the constraint optimization problem of (13), which guarantees convergence to a local minima is given by:

\[
u_{i,j} = \begin{cases} 
\frac{1}{\sum_{j=1}^c \frac{||x_i - c_j||}{\sum_{l=1}^c ||x_i - c_l||}^{(m-1)}} & \text{if } j = i_l, 1 \leq l \leq t \\
0 & \text{Otherwise.}
\end{cases}
\]

\[
c_j = \frac{\sum_{i=1}^n \Pr(i)u_{i,j}^m x_i}{\sum_{i=1}^n \Pr(i)u_{i,j}^m} 
\tag{15}
\]

Finally, we note that although in PRSC-t, each tuple has unit weight, variable weights can be easily incorporated. If tuple \(i \in T\) has weight \(w_i\), then the objective function for weighted PRSC-t (WPRS-t) becomes

\[
\min \sum_{i \in T} \Pr(i)w_i \sum_{j=1}^c u_{i,j}^m ||x_i - c_j||^2 \quad 1 \leq m < \infty
\]

All the constraints remain the same as Theorem 3. The iterative solution is obtained by substituting \(\Pr(i)w(i)\) in place of \(\Pr(i)\) in Theorem 5.

We note that our ClusterRank framework (that we discuss next) requires the ability to handle tuples with weights, even if the tuples in the database are not weighted.

## 6 Cluster Rank in Probabilistic Databases

In this section we discuss our framework and algorithms for ClusterRank using the ranking and clustering algorithms developed in the previous sections. The main features of our ClusterRank algorithm are as follows:

- We adapt the hierarchical framework proposed by Guha et al. [21] to do in memory clustering over probabilistic relations; our algorithm is not only very fast, but is also able to handle large databases and even streaming databases effectively, without any significant loss of precision.

- Our ClusterRank algorithm with E-Score ranking function requires only a single pass over the database and can perform ranking and clustering in parallel. We note that this also provides a new algorithm for ClusterRank over deterministic databases.
• Our ClusterRank algorithm with \( PRF^e \) function requires only two passes over the database assuming the tuples are sorted by score. After the final cluster centers are found after the first pass, only \( O(ck) \) space is required to compute the Top-\( k \) tuples using \( PRF^e \).

We begin with briefly describing the overall framework, and then present the two specific algorithms for ClusterRank using E-Score and \( PRF^e \) ranking functions.

6.1 Hierarchical Framework for Cluster-Rank

Hierarchical clustering divides the original data to be clustered arbitrarily into several partitions, such that each partition fits in the memory. Each partition is clustered separately and from each of them \( k \) cluster centers are identified. These centers are assigned weight equal to the number of data items that belong to that center. These clusters represent the first level of the hierarchy. The weighted centers from the first level of hierarchy are treated as data for the second level and the clustering proceeds recursively.

This hierarchy naturally defines a parent-child relation among the tuples\(^3\). If the number of levels in the hierarchy is \( h \), then a tuple \( t_x \) at level \( l \) is a child of a tuple \( t_y \) at level \( l + 1 \), if \( t_y \) is the center of the cluster in which \( t_x \) belongs. If the final clusters are \( C_1, C_2, \ldots, C_k \), then a tuple in the lowest level belongs to \( C_i \), if it has a path to \( C_i \), in the graph defined by this parent-child relationship on the tuples.

Guha et al. [21] prove that if the number of levels in the hierarchy is constant and if in every level a constant factor approximation algorithm for hard \( k \)-median is used, the error in the objective function is bounded by a constant factor from the optimum. If \( k \)-means objective function is used instead of \( k \)-median, the approximation guarantee improves by a factor of \( 2 \).

We extend this framework to probabilistic databases by instead performing \( WPRSC-t \) at the highest level and performing hard clustering (\( WPRSC-t, t = 1 \)) at the lower levels. In the appendix (D), we show that this gives us a constant factor approximation of the original \( WPRSC-t \) objective (we present more details in the next section). Note that the use of \( WPRSC-t \) only at the highest level guarantees that each tuple can belong to at most \( t \) clusters. This gives us a one-pass algorithm for \( WPRSC-t \) in probabilistic databases. Next we present algorithms for doing both clustering and ranking when using either the E-Score or \( PRF^e \) ranking function.

6.2 ClusterRank using Expected Score

Our one-pass algorithm for ClusterRank using expected score ranking function is based on the following theorem.

**Theorem 6** When \( t \geq 1 \), a tuple \( t_x \) appears as Top-\( k \) in one of the final clusters, if and only if it appears as Top-\( k \) in all of the intermediate clusters it belongs to at all levels in the hierarchy.

**Proof:** Suppose \( e \) does not appear as top-\( k \) tuple in an intermediate cluster \( I \) at level \( l \) but appears as the top-\( k \) tuple in a final cluster. Let the top-\( k \) tuples in \( I \) are \( a_1, a_2, a_k \).

Case 1: \( t = 1 \). All of \( a_1, a_2, a_k \) have higher scores than \( e \) in \( I \). At level \( l + 1 \), \( a_1, a_2, \ldots, a_k, e \) all will have cluster center \( I \) as their successor. Since a cluster once formed is never broken and the score of a tuple never changes during the execution, \( e \) and \( a_1, a_2, \ldots, a_k \) will all belong to the same final cluster. So \( e \) does not qualify as top-\( k \) tuple in the final cluster—a contradiction.

Case 2: \( t > 1 \). Let \( F_I \) be the cluster at level \( h - 1 \) (\( h \) is the ultimate level), which contains \( I \). Since up to level \( h - 1 \), only hard clustering is done, following the argument for the case 1, \( t \) does not qualify as top-\( K \) in \( F_I \). Also \( a_1, a_2, \ldots, a_k \) all belong to \( F_I \) and have higher score than \( t \). Now at level \( h \), \( F_I \) is treated as a data point and suppose it belongs to \( t \) clusters \( c_1, c_2, \ldots, c_t \) at level \( h \). Let the membership probability of \( F_I \) in these \( t \) clusters be \( m_1, m_2, \ldots, m_t \). In each of these \( c_i \)'s, \( i = 1, 2, \ldots, t \), the \( k + 1 \) elements \( e, a_1, a_2, \ldots, a_k \) have the membership probability same as the corresponding membership probability of \( F_I \). Therefore the expected score of \( e \in c_i \) is \( score(e) * m_i > score(a_j) * p_{a_j} * m_i \), \( \forall i \in [1, t] \) and \( \forall j \in [1, k] \). So \( e \) does not qualify as top-\( k \) tuples in any of the final clusters.

\(^3\)Here we use tuples to represent both the original tuple in the database, which we want to cluster and the intermediate clusters of the hierarchical framework.
Also if \( e \) appears as the top-k tuple in the clusters it belongs to at every level, it appears as the top-k in the final cluster, it belongs to.

Thus, for each intermediate cluster, we only need to maintain a subset of the original data belonging to that cluster (by parent-child relation) as possible candidates for Top-k; the rest of the data can be discarded. This enables us to do ranking and clustering in parallel, and there is no need to wait for the final clusters to materialize. In Algorithm 4, we give the sketch of the one pass algorithm for ClusterRank with E-Score. In the algorithm \( D_T \) denotes the probabilistic database, \( h \) denotes the number of hierarchies, \( l \) denotes the number of clusters and \( k \) denotes the number of top tuples that have to be returned from each cluster. Initially the weight of each tuple is set to its existence probability.

**Algorithm 4 Expected Score ClusterRank \((D_T, h, l, k)\)**

1. Accept data from \( D_T \) as long as it fits in the memory.
2. Cluster the data in memory using WPRSC-1 which is equivalent to hard clustering into \( l \) clusters.
3. Detect the cluster centers of these \( l \) clusters. Weigh each one of them by the sum of the weights of the tuples that belong to these clusters. Treat them as the input data for the level 1.
4. For each of the \( l \) clusters, find the Top-k tuples, using E-Score ranking. Discard the rest of the tuples. Assign these tuples as the Top-k tuples for the corresponding cluster centers that go to the next level.
5. Repeat Steps 1 to 4 as long as there is unconsidered data from \( D_T \) or memory is exhausted by the level 1 data.
6. Cluster the level 1 data in memory similarly to find \( l \) cluster centers which are treated as level 2 data. Weigh them properly and detect the Top-k tuples for each cluster using the Top-k tuples assigned to the data that belong to the corresponding cluster. Prune rest of the data.
7. Continue accepting unconsidered data from \( D_T \) and if there is none, proceed with the clustering and ranking step on the higher levels.
8. If the data to be clustered are at level \( h \), use WRSC-\( t \) clustering on them to obtain the final \( l \) clusters. Detect the Top-k tuples from each of the final clusters, using Top-k tuples assigned to the level \( h \) data in the corresponding clusters.

### 6.3 Two-Pass ClusterRank Algorithm for PRF\( ^e \)

The one-pass algorithm described above can not be applied to computing ClusterRank with the PRF\(^e\) function directly. A tuple that is found to not belong to the Top-k answer for a cluster cannot be thrown away since it may be needed to compute the rank of a tuple that is processed later. We next present a two-pass algorithm with low memory complexity if the tuples are independent of each other and the database is pre-sorted by score (similar assumptions have been made in prior work \([35]\)). In the first pass of the algorithm, we compute the \( c \) cluster centers using the one-pass clustering algorithm described above. We then make a second pass over the dataset to compute the Top-k tuples in each cluster using very little memory \( O(ck) \), where \( c \) is the number of clusters.

The algorithm is as follows: suppose that \( f(i) = a^t \) is used for PRF\(^e\) function and \( c_1, c_2, \ldots, c_c \) are the cluster centers obtained in the first pass. We maintain \( c \) generating function values \( F_j \) and \( c \) min-heaps (priority queues) \( H_j \) for \( 1 \leq j \leq c \), one for each cluster. Heap \( H_j \) maintains the \( k \) tuples with largest PRF\(^e\) values for cluster \( j \) seen so far.

When we read in a new tuple \( t_i \), we can compute the probability that \( t_i \) belongs to cluster \( c_j \) \((u_{i,j} = \Pr(t_i \in c_j))\) for each \( 1 \leq j \leq c \) using Equation (15). (In hard clustering, we only need to find the closest cluster center). Recall from Section 4.3.1 and the definition of soft clustering, we only need to let the new generating function value be:

\[
F_j(a) \leftarrow \frac{u_{i,j} \Pr(t_i)}{u_{i-1,j} \Pr(t_{i-1})} F_j(a)(1 - u_{i,j} \Pr(t_i)) + a u_{i,j} \Pr(t_i)).
\]

The above formula may become undefined if \( u_{i,j} \) is 0. If \( u_{i,j} \) is 0, we simply ignore this tuple in cluster \( j \) instead of using the above formula. By Eq. (1), we know \( Y_j(t_i) = F_j(a) \) where \( Y_j(t_i) \) is the final PRF value of \( t_i \) in cluster \( j \). Then, we compare \( Y_j(t_i) \) with the minimum element in \( H_j \) and see if we need to update \( H_j \). The whole process takes \( O(c \log k) \) processing time for new incoming tuple and \( O(ck) \) space in total.
6.4 General Case

In the general case, when the tuples exhibit more complex correlations than mutual exclusivity or when a ranking function other than E-Score or PRF is used, neither of the above algorithms can be directly applied. We can always use the naive algorithm of computing the clusters first using our framework, and then using the algorithms described in Section 4 for PRF computation on the tuples in each cluster. Understanding the space-time tradeoffs in the general case remains a rich area of future work.

7 Experiments

In this section we present the results of an empirical study that illustrates the diverse and conflicting behaviors of different ranking functions, and demonstrates the effectiveness of our clustering and cluster-rank framework. The algorithms are implemented in C++, and the experiments were run on a PC with 2.4 GHz Intel (dual hyperthreaded CPUs each with 1MB cache), 2 GB RAM, and 120 GB disks, running Linux.

7.1 Data-Sets & Comparison Metric

Data Set: We mainly use the International Ice Patrol (IIP) Iceberg Sighting Database for our experiments. This dataset is also used in prior work on ranking and Top-k queries [28, 25]. The database contains a set of sighting records, each of which contains, among other attributes, the location (latitude, longitude) of the iceberg and the number of days the iceberg has drifted. 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 location coordinates are used for clustering purpose. 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 six confidence levels into (RD-200,000),...,(RD-1,000,000) records, obtained by uniform random sampling.

We also use synthetic datasets to demonstrate some special properties or behaviors. The synthetic dataset for ranking is used solely to show the effect of correlation on ranking, where the correlations are captured by an and/or tree. The data-set consists of 1000 tuples. By varying the depth, degree, and the ratio of and/or nodes, three datasets with different levels of correlation (LOW, MEDIUM, HIGH) are generated. For clustering and cluster-rank experiments, a synthetic dataset of 100,000 tuples is used, with each tuple having 4 attributes. For each tuple in the dataset, a multivariate Gaussian distribution is selected randomly from 10 such distribution (with different means and covariances) and the tuple attribute values are generated according to the chosen Gaussian. The tuple existence probabilities are generated uniformly from the interval [lb, 1], with the lower bound lb for 10 different datasets varying from 0.1 to 1 (from highly uncertain to certain).

Comparison Metrics: To measure the “closeness” of two ranking results, we use the Kendall’s tau distance defined for comparing Top-k answers [16]. It is also called Kemeny distance in the literature and is considered to have many advantages over other distance metrics [15]. Let \( R_1 \) and \( R_2 \) denote two ranked list and let \( K_1 \) and \( K_2 \) denote the top-k ranked tuples of \( R_1 \) and \( R_2 \) respectively. Then Kendall’s tau distance between \( K_1 \) and \( K_2 \) is defined to be:

\[
\text{dis}(K_1, K_2) = \sum_{i \in K_1, j \in K_2} \hat{K}(i, j),
\]

where \( \hat{K}(i, j) = 1 \), if it can be inferred from \( K_1 \) and \( K_2 \) that \( i \) and \( j \) appear in opposite order in the two full ranked lists \( R_1 \) and \( R_2 \). Otherwise \( \hat{K}(i, j) = 0 \). A high value of Kendall’s tau distance indicates a larger “disagreement” between the two Top-k lists. For ease of comparison, we divide the Kendall distance by a factor of \( k^2 \) to obtain normalized Kendall’s tau distance, which always lies in [0, 1].

To measure the clustering quality, we simply use the clustering objective function (the sum squared distance in our case). For ClusterRank experiment, we propose the following metric to measure the closeness of the results obtained by

\footnote{http://nsidc.org/data/g00807.html}
two different ClusterRank algorithms. Suppose \( C_1, C_2 \) are two sets of cluster centers found by two different clustering algorithms on the same data-set, with \( |C_1| = |C_2| \). If both of them are good approximations of the optimal clustering, it is likely that there is an one-to-one correspondence \( \sigma : C_1 \rightarrow C_2 \) such that, for each cluster center \( c \in C_1, c \) and \( \sigma(c) \in C_2 \) are close to each other. We find this correspondence by computing a minimum weight perfect matching between the cluster centers \( C_1 \) and \( C_2 \). Then the \( Top-k \) answers of two corresponding clusters are compared using the normalized Kendall’s tau distance. Total distance between the two clusterings is computed to be the average of these normalized Kendall’s tau distances. If two ClusterRank algorithm return similar clustering and for each cluster, they return similar \( Top-k \) answers, we expect the average Kendall’s tau distance to be small.

### 7.2 Experiments with Ranking Functions

We begin with evaluating the behavior of \( E\text{-Score} \), \( PRF^c(a=0.95) \), \( PT-k \), \( URank-k \), \( Ex-Rank \), and \( UTop-k \) ranking functions over independent tuples. Table 1 tabulates the normalized Kendall’s tau distance between the Top-100 answers obtained using different ranking functions for two datasets. The value in each cell indicates the computed distance between the corresponding ranking functions (we omit the redundant column corresponding to \( UTop-k \)). Both these tables and Figure 5(i) (which we discuss shortly), suggest that \( PRF^c \) (for reasonable values of \( a \), e.g. \( a = 0.95 \)), \( PT-k \), and \( URank-k \) form a group, i.e., they are close to each other; whereas the behavior of other functions is quite sensitive to the dataset. From the first table, we can see that \( Ex-Rank \) behaves very differently from all other functions, whereas in the second table, \( Ex-Rank \) happens to be quite close to \( E\text{-Score} \). However both of them deviate largely from \( PRF^c \), \( PT-k \) and \( URank-k \).

Our investigation into the behavior of \( Ex-Rank \) indicated the following undesirable property of that function. In a large dataset, \( Ex-Rank \) tends to give unreasonably high priority to a tuple with a high probability even it has a low score. For example, in our synthetic dataset with 100000 tuples with expected size \( \approx 50000 \), \( t_2 \) (the tuple with 2nd highest score) has probability approximately 0.98 and \( t_{1000} \) 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 \( t_{1000} \) has a much lower score that \( t_2 \), and is only slightly more probable than \( t_2 \).

We also observe that \( UTop-k \) behaviour is not stable: it is reasonably close to \( PRF^c \), \( PT-k \) for the first dataset, but far away from them in the second dataset. The original \( URank-k \) function may return the same tuple at different

<table>
<thead>
<tr>
<th></th>
<th>( E\text{-Score} )</th>
<th>( PRF^c(a=0.95) )</th>
<th>( PT-k )</th>
<th>( URank-k )</th>
<th>( Ex-Rank )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E\text{-Score} )</td>
<td>0.0000</td>
<td>0.1649</td>
<td>0.1241</td>
<td>0.3027</td>
<td>0.7992</td>
</tr>
<tr>
<td>( PRF^c(a=0.95) )</td>
<td>0.1649</td>
<td>0.0000</td>
<td>0.2022</td>
<td>0.2761</td>
<td>0.9070</td>
</tr>
<tr>
<td>( PT-k )</td>
<td>0.1241</td>
<td>0.2022</td>
<td>0.0000</td>
<td>0.3324</td>
<td>0.9290</td>
</tr>
<tr>
<td>( URank-k )</td>
<td>0.3027</td>
<td>0.2761</td>
<td>0.3324</td>
<td>0.0000</td>
<td>0.9293</td>
</tr>
<tr>
<td>( Ex-Rank )</td>
<td>0.7992</td>
<td>0.9070</td>
<td>0.9290</td>
<td>0.9293</td>
<td>0.0000</td>
</tr>
<tr>
<td>( UTop-k )</td>
<td>0.2760</td>
<td>0.2017</td>
<td>0.3674</td>
<td>0.2046</td>
<td>0.9456</td>
</tr>
</tbody>
</table>

**RD-100,000 (k = 100)**

<table>
<thead>
<tr>
<th></th>
<th>( E\text{-Score} )</th>
<th>( PRF^c(a=0.95) )</th>
<th>( PT-k )</th>
<th>( URank-k )</th>
<th>( Ex-Rank )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E\text{-Score} )</td>
<td>0.0000</td>
<td>0.8678</td>
<td>0.8642</td>
<td>0.8902</td>
<td>0.0044</td>
</tr>
<tr>
<td>( PRF^c(a=0.95) )</td>
<td>0.8678</td>
<td>0.0000</td>
<td>0.2937</td>
<td>0.2416</td>
<td>0.8683</td>
</tr>
<tr>
<td>( PT-k )</td>
<td>0.8642</td>
<td>0.2937</td>
<td>0.0000</td>
<td>0.3950</td>
<td>0.8647</td>
</tr>
<tr>
<td>( URank-k )</td>
<td>0.8902</td>
<td>0.2416</td>
<td>0.3950</td>
<td>0.0000</td>
<td>0.8907</td>
</tr>
<tr>
<td>( Ex-Rank )</td>
<td>0.0044</td>
<td>0.8683</td>
<td>0.8647</td>
<td>0.8907</td>
<td>0.0000</td>
</tr>
<tr>
<td>( UTop-k )</td>
<td>0.9258</td>
<td>0.3608</td>
<td>0.5791</td>
<td>0.3160</td>
<td>0.9263</td>
</tr>
</tbody>
</table>

**Synthetic dataset with 100,000 tuples (k = 100)**
Figure 4: Experimental results for ranking: (i) Effect of correlation on PRF\textsuperscript{e} ranking as \(a\) varies; (ii) Effect of correlation for PRF\textsuperscript{e}, URank-\(k\) and PT-\(k\); (iii) Running times with various database sizes and various values of \(k\).

Figure 5: (i) Comparing PRF\textsuperscript{e} function with other ranking functions; (ii, iii) Effect of number of samples on learnability of different ranking functions using PRF\textsuperscript{e} and PRF\textsuperscript{w}.

ranks (also observed by the authors [35]), which is usually not desirable. This problem becomes even severe when the dataset and \(k\) are both large. For example, in RD-100,000, the same tuple is ranked at positions 67895 to 100000. In the table, we show a slightly modified version of URank-\(k\) to enforce distinct tuples in the answer.

Next we evaluate the behavior of ranking functions over probabilistic datasets modeled using probabilistic and/or trees. We use three different correlated datasets with different degrees of correlations which are generated by changing the height, the maximum degree, and the proportion of \(\lor\) and \(\land\) nodes. Roughly speaking, higher height, larger degree and more mixed degree of \(\lor\) and \(\land\) nodes introduce stronger correlation. We compute PRF\textsuperscript{e} ranking by both using PROB-ANDOR-PRF-RANK and IND-PRF-RANK (which means we simply ignore the correlation). We test PRF\textsuperscript{e} functions with varying \(f(i) = a^i\) from \(a = 0.1\) to 1.0. In Figure 4 (i), we plot the normalized Kendall’s distance between the rankings obtained by the two algorithms. As we can see, on highly-correlated datasets, ignoring the correlations can result in significantly inaccurate Top-\(k\) answers. As \(a\) approaches 1, PRF\textsuperscript{e} tends to sort the tuples by probabilities, so all 3 curves become close to 0. We also compare PRF\textsuperscript{e} with PT-\(k\) and URank-\(k\) (Figure 4 (ii)) for varying degrees of correlations. From the figure, we can see there is a substantial difference between Top-\(k\) answers obtained by the algorithms in presence of high correlations. We note that ranking by E-Score is invariant to the correlations, which is a significant drawback of that function.

Figure 4 (iii) shows the execution time for different ranking functions, PRF\textsuperscript{e}, PT-\(k\), URank-\(k\) and Ex-Rank. As expected both PRF\textsuperscript{e} and Ex-Rank run very fast (1000000 tuples can be ranked within 1 or 2 seconds). Indeed, after sorting the dataset in a non-decreasing score order, PRF\textsuperscript{e} needs only a single scan of the dataset and Ex-Rank needs scan the dataset twice. Execution time for PT-\(k\) and URank-\(k\) increase linearly with \(k\) and the algorithms become slow as \(k\) increases. On the other hand the running times of both PRF\textsuperscript{e} and Ex-Rank are not affected by \(k\).

An important property of the PRF functions is their ability to learn the user preferences characterized implicitly by other ranking functions. Figure 5 (i) shows the Kendall’s distance for top 100 tuples between PRF\textsuperscript{e} and the other
7.3 Experiments with Clustering and Cluster-Rank Algorithms

We implemented the probabilistic version of K-means and the Grid method [31] for comparison purposes – specifically, each tuple has a weight (its existence probability), and we multiply the tuple related term in any computation performed by the algorithm using this weight. For our first experiment, we compare the total sum squared error for the clusterings returned by each of the algorithms (Figure 6(i)). From the figure, we see that as \( t \) (the number of clusters a tuple can belong to in soft clustering) increases, the errors for \( RSC-t \) decrease. This matches the theoretical result that (carefully) splitting tuples’ membership among more clusters results in less error than using hard clustering. In all the cases, the clustering error for hierarchical clustering is within a factor of 1.5 of the corresponding K-means (for \( t = 1 \)) or \( PRSC-t \) (\( t \geq 2 \)) which justifies the clustering quality obtained from the hierarchical clustering. Figure 6(ii) shows the decreasing trend of clustering errors when we increase the number of partitions and \( t \) in hierarchical clustering. As
we expected, more partitions produce more accurate solution. However the space usage increases as the number of partitions increases.

Our ClusterRank experiment was done as follows. We first apply our ClusterRank algorithm to probabilistic dataset to obtain a set of Top-k answers, one for each cluster (denoted \( \text{ans}_p \)). Then we independently sample a set of possible worlds, according to the probability distribution over the possible worlds (i.e., we choose a possible world \( \text{pw} \) with \( \Pr(\text{pw}) \)). For each of these worlds, we run the deterministic version of the ClusterRank algorithm to obtain an answer (\( \text{ans}_i \)), and compute the distance between \( \text{ans}_i \) and \( \text{ans}_p \) (by first finding a minimum weight matching between the cluster centers, and then computing the average Kendall distance between the matching clusters). The plotted value is obtained by averaging the distance over 100 independent samples (possible worlds). To understand the behavior of probabilistic cluster-rank properly, we use the synthetic data-set of size 100,000 tuples, which has 10 predefined clusters. We vary the lower bound \( lb \) of the tuple probabilities from 0.1 to 1 to obtain databases with varying degree of uncertainties (Figure 6 (iii)). As we can see, the average distances become smaller as the datasets become less uncertain, and reach 0 when the dataset is certain. When \( lb \) is small, the average distance is around 0.5 or higher which seems to be a bad result at the first glance. However, even if we assume the clustering is always done perfectly, the distance between a Top-k answer for some cluster \( C \) in the probabilistic setting and the Top-k answer for a sample of \( C \) is not likely to be very small given the dataset is quite uncertain. Therefore, we believe capturing more than 50% of Top-k answers from possible worlds in average is quite acceptable and we can not expect much better results. When the same ranking function \( PRF^e \) is used, the best result is obtained for probabilistic k-means. Hierarchical clustering outperforms grid-based clustering except in a small region of probabilities. Hierarchical clustering also runs faster in comparison to K-means and for highly uncertain databases, its performance is better than probabilistic k-means in several cases. Figure 6 (iv) indicates the average distance for ClusterRank queries is not sensitive to the choice of the ranking function. This implies we can use any ranking function in the hierarchical framework if we believe the ranking function we use characterizes the user preference. We recommend the two pass algorithm along with \( PRF^e \) or even one pass algorithm for E-Score, if E-Score perform reasonable well, mostly for efficiency concerns.

8 Conclusions

The increasing amounts of uncertain data in a variety of diverse application domains has created a need for efficient algorithms for supporting complex queries such as ranking and clustering over probabilistic databases. In this paper we presented a framework for analyzing 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 single ranking function, we propose using a parameterized family of ranking functions, which allows the user to control the tuples that appear in the Top-k answers. We also introduced the notion of restricted soft-t clustering and developed a framework to support clustering over large probabilistic databases. Our work opens up many avenues for further research that we are planning to pursue. For instance, there may be other non-trivial subclass of PRF functions, aside from \( PRF^e \), that are easy to compute and integrate in the ClusterRank framework. Understanding the behavior of clustering and ranking across probabilistic databases with diverse uncertainty and correlation structures also remains an important open problem in this area.

References


A Desiderata for Ranking Functions

Below we propose three desirable properties that a ranking function should satisfy. As we discuss later, there are many ranking functions that may satisfy these properties, and we present a ranking function that can explore this space of ranking functions.

1. (Dominating Tuple First (DTF):) A tuple appears as a Top-k answer only if all tuples that dominate it do. We say the tuple $t_1$ dominates another tuple $t_2$ in probabilistic database $D_T$ if $\forall i = 1, \ldots, n: p_i(t_1) \geq p_i(t_2)$ and at least one inequality holds strictly.

2. (Strongly Dominating Tuple First (SDTF):) A tuple appears as a Top-k answer only if all tuples that strongly dominate it do. Formally, we say the tuple $t_1$ strongly dominates another tuple $t_2$ if $t_2 \in pw \rightarrow t_1 \in pw$ for any $pw \in PW$ and $s(t_1) \geq s(t_2)$.

   It is worth noting that $t_1$ dominates $t_2$ if $t_1$ strongly dominates $t_2$ but the converse is not necessarily true.

3. (Uniqueness Of Solution(UOS):) A Top-k query should return the same results for two isomorphic probabilistic databases. Formally, we say two probabilistic databases $D_1$ and $D_2$ are isomorphic with respect to ranking if there is a one-to-one mapping $\phi: D_1 \rightarrow D_2$ and $\varphi: PW(D_1) \rightarrow PW(D_2)$ such that:

   (1) $Pr(pw) = Pr(\varphi(pw))$ for any $pw \in PW(D_1)$,
   (2) $t \in pw$ if and only if $\phi(t) \in \varphi(pw)$ for any $t \in D_1, pw \in PW(D_1)$, and
   (3) $r_{pw}(t) = r_{\varphi(pw)}(\phi(t))$ for any $t \in D_1, pw \in PW(D_1)$.

   Intuitively, two isomorphic probabilistic databases have essentially the same possible worlds spaces and same ranking in each possible world, after renaming. This property implies that the same answers should be generated for a single database regardless of the real form of the score functions we use, as long as they give the same ranking in each possible world.
Table 7 summarizes the behavior of the ranking functions with respect to the desirable properties discussed above. We illustrate some of the key entries in the above table with examples and lemmas.

**Example 4** (URank-k) Consider the database shown in Figure 8 containing 9 highly-correlated tuples. URank - 6 will return 1, 2, 3, A, e, f as the answer (in that order). This answer does not satisfy the SDTF property (and hence the DTF property), since the tuples B and C are ranked higher than e and f, and they appear in every world that e and f appear. In other words, B and C dominate e and f, but are ranked lower according to the URank-k function.

<table>
<thead>
<tr>
<th>Property</th>
<th>SDTF</th>
<th>DTF</th>
<th>UOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-Score</td>
<td>Yes (Lemma 1)</td>
<td>No (Example 5)</td>
<td>No (Example 5)</td>
</tr>
<tr>
<td>URank-k</td>
<td>No (Example 4)</td>
<td>No (Example 4)</td>
<td>Yes</td>
</tr>
<tr>
<td>PRFw</td>
<td>Yes (Lemma 2)</td>
<td>Yes (Lemma 2)</td>
<td>Yes</td>
</tr>
<tr>
<td>UTop-k</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Figure 7:** Comparing ranking functions

Figure 8: An example probabilistic relation over highly-correlated tuples A, B, C, d, e, f, 1, 2, 3. In each possible world, the tuples are listed in the increasing order by score.

**Example 5** (E-Score) The Top-k query results of E-Score depend on the real form of the score function. Consider a database with two independent tuples A and B with Pr(A) = 0.1, and Pr(B) = 1. Say we have two score functions s1, s2 such that s1(A) = 5, s1(B) = 1 and s2(A) = 101, s2(B) = 1. In each possible world, we have the same ranking by either score function. However, E-Score function will rank B above A when using s1 and A higher than B when using s2, thus violating the UOS property. Further, it is easy to see that B dominates A. Since E-Score with scoring function s2 ranks A above B, it does not satisfy the DTF property.

**Lemma 1** E-Score function satisfies SDTF property.

**Proof:** Suppose t1 strongly dominates t2. It is trivial to see Pr(t1) ≥ Pr(t2) from t2 ∈ pw → t1 ∈ pw. So, E(s(t1)) = Pr(t1) · s(t1) ≥ Pr(t2) · s(t2) = E(s(t2)). □

**Lemma 2** PRFw function satisfies SDTF and DTF.

**Proof:** We only need to prove PRFw function satisfies DTF. Suppose t1 dominate t2. Define piecewise constant functions

\[ g_{th}(x) = f(i) \quad \text{if} \quad p_{i-1}(t_h) \leq x < p_i(t_h) \]

for h = 1, 2. It is not hard to see

\[ \Upsilon(t_h) = \sum_{i=0}^{n} f(i)Pr(r(t_i) = i) = \int_0^{1} g_{th}(x)dx. \]

Assume that p_{i-1}(t_1) ≤ x < p_i(t_1) for some 0 ≤ x ≤ 1. From the definition of dominance, it is easy to see p_{i-1}(t_2) ≤ x. So, we can get g_{th}(x) ≤ f(i) = g_{t_1}(x) since f is nonincreasing. Therefore,

\[ \Upsilon(t_1) - \Upsilon(t_2) = \int_0^{1} (g_{t_1}(x) - g_{t_2}(x)) dx \geq 0. \] □
B Multiplication of a Set of Polynomials

Given a set of polynomials $P_i = \sum_{j \geq 0} c_{ij} x^j$ for $1 \leq i \leq k$, we want to compute the multiplication $P = \prod_{i=1}^{k} P_i$ written in the form $P = \sum_{j \geq 0} c_{j} x^j$. Let $d(P)$ be the degree of the polynomial $P$.

First we note that the naive method (multiply $P_i$ one by one) gives us a $O(n^2)$ time algorithm by simple counting argument. Let $P_i = \prod_{j=1}^{m} P_j$. It is easy to see $d(P_i) = \sum_{j=1}^{m} d(P_j)$. So the time to multiply $P_i$ and $P_{i+1}$ is $O(d(P_i) \cdot d(P_{i+1}))$. Then, we can see the total time complexity is

$$\sum_{i=1}^{k-1} O(d(\bar{P}_i) \cdot d(P_{i+1})) = O(n) \cdot \sum_{i=1}^{k-1} d(P_{i+1}) = O(n^2).$$

Now, we show how to use divide-and-conquer and FFT(Fast Fourier Transformation) to achieve a $O(n \log^2 n)$ time algorithm. It is well known that the multiplication of two polynomials of degree $O(n)$ can be done in $O(n \log n)$ time using FFT. The divide-and-conquer algorithm is as follows: If there exists any $P_i$ such that $d(P_i) \geq \frac{1}{3} d(P)$, we evaluate $\prod_{j \neq i} P_j$ recursively and then multiply it with $P_i$ using FFT. If not, we partition all $P_i$s into two sets $S_1$ and $S_2$ such that $\frac{1}{3} d(P) \leq d(\prod_{i \in S_1} P_i) \leq \frac{2}{3} d(P)$. Then we evaluate $S_1$ and $S_2$ separately and multiply them together using FFT. It is easy to see the time complexity of the algorithm running on input size $n$ satisfies

$$T(n) \leq \max\{T(\frac{2n}{3}) + O(n \log n), T(n_1) + T(n_2) + O(n \log n)\}$$

where $n_1 + n_2 = n$ and $\frac{1}{3} n \leq n_1 \leq n_2 \leq \frac{2}{3} n$. By solving the above recursive formula, we know $T(n) = O(n \log^2 n)$.

C PRF Computation in Graphical Models

In this section, we show how to evaluate the PRF function if the probabilistic database is modeled as a Markov Network or a Bayesian network [34]. The resulting algorithm is a non-trivial dynamic program over junction tree combined with the generating function method. Let’s briefly review some notations and definitions related to graphical model.

For each tuple $t$ in $T$, we associate a boolean random variable $x_t$, and let $\mathcal{X}$ denote the set of all such tuple existence random variables. $x_t = 1$ stands for $t$ is present and $x_t = 0$ otherwise. Let $T$ be a tree with each node $v$ associated with a subset $C_v \subseteq \mathcal{X}$. We say $T$ is a junction tree if any intersection $C_u \cap C_v$ for any $u, v \in T$ is contained in every node on the unique path between $u$ and $v$ in $T$ (this is called the running intersection property). The treewidth of a junction tree is $\max_{v \in T} |C_v| - 1$. Denote $S_{u,v} = C_v \cap C_u$ for each $(u, v) \in T$. Suppose we are given a set of potentials $5 \{\pi_v | v \in T\} \cup \{\mu_{u,v} | (u, v) \in T\}$ where $\pi_v$ is a potential over $C_v$ and $\mu_{u,v}$ is a potential over $S_{u,v}$. We say the joint distribution $Pr(\mathcal{X})$ can be factorized over $T$ if it can be written in the form:

$$Pr(\mathcal{X}) = \frac{\prod_{v \in T} \pi_v(C_v)}{\prod_{(u,v) \in T} \mu_{u,v}(S_{u,v})}$$

The set of potentials $Q$ is calibrated when for each $(u, v) \in T$ the potential $\mu_{u,v}$ is the marginal of $\pi_u$ and $\pi_v$, i.e., $\mu_{u,v}[S_{u,v}] = \sum_{C_u \setminus S_{u,v}} \pi_u[C_u]$. It has been proven that if $Q$ is calibrated, then $\pi(C_v) = Pr(C_v)$ and $\mu_{u,v}(S_{u,v}) = Pr(S_{u,v})$, i.e., the potentials are exactly equal to the marginal probability distributions. We can assume without loss of generality that the given set of potentials is calibrated since calibrated potentials can be computed from an arbitrary set of potential using message passing [17].

Let $T = \{t_1, t_2, \ldots, t_n\}$ be the set of tuples in $D_T$, sorted in a nonincreasing order of their score function, and $T_i$ be the set $\{t_1, t_2, \ldots, t_i\}$. Now we show how to compute $Pr(r(t_i) = h)$ in polynomial time for some $t_i \in T$ and any positive integer $1 \leq h \leq n$. Note that for each $t_i$, we need to run the dynamic program again. Root $T$ at an arbitrary node $r$ such that $t_i \in C_r$. Let $T_v$ be the subtree rooted at the vertex $v \in T$. Let $\sigma_v \in \{0, 1\}^{|C_v|}$ be a boolean vector of

---

5A potential is just a function mapping to non-negative real numbers.
size $|C_v|$, which we call the existential configuration of $C_v$. If $C_v$ (recall $C_v$ is a set of random variable) takes value $\sigma_v$, we say $C_v$ agrees with the existential configuration $\sigma_v$.

We say two existential configurations $\sigma_u$ and $\sigma_v$ are consistent with each other if they agree with each other on $C_u \cap C_v$, denoted by $\sigma_u \sim \sigma_v$. We use $|\sigma|$ to denote the number of 1’s in vector $\sigma$. With a little abuse of notation, we let $\sigma_v \cap C$ (or $\sigma_v \cap N$) be the induced vector which only contain entries in $C_v \cap C$ (or $C_v \cap u \in T$). We define $\sigma_v \cup \sigma_u$ similarly. If $\sigma_u \sim \sigma_v$, we denote $\sigma_{u,v}$ as the existential configuration of $S_{u,v} = C_u \cap C_v$ and $\tilde{\sigma}_v$ as an extension of $\sigma_v$ to the subtree $T_v$, i.e., $\tilde{\sigma}_v$ is an existential configuration of $T_v$ and agrees with $\sigma_v$.

Now, we describe our dynamic program which runs over the junction tree $T$ bottom up. Actually, our algorithm will compute $\Pr(v, \sigma_v, \delta)$, the probability that $C_i$ agrees with $\sigma_v$ and there are $\delta$ tuples present in $T_v \cap T_{i-1}$, i.e., $\sum_{t \in T_v \cap T_{i-1}} x_t = \delta$. Our final solution will be $\Pr(r(t) = h) = \sum_{\sigma_v} \Pr(r(\sigma_v, h - 1))$ where $r$ is the root of the junction tree $T$. If $v_1, v_2, \ldots, v_l$ be the children of $v$ in $T$. Before providing the recursion formula, we need the following simple fact.

Lemma 3 1. If $\sigma_{v_i} \sim \sigma_v$ and $\sigma_{v_j} \sim \sigma_v$ in $T$, then $\sigma_{v_i} \sim \sigma_{v_j}$.

2. If $\sigma_{v_i} \sim \sigma_v$, then $\sigma_{v_i} \sim \sigma_v$.

Proof: Both parts follow directly from the running intersection property of junction trees. More specifically, the first part can be seen from the fact that $C_{v_i} \cap C_{v_j} \subseteq C_v$ and the second holds because $C_u \cup C_v \subseteq C_{v_j}$ for any $u \in T_{v_i}$. □

We use shorthand notation $[\sigma_{v_j}]$ to denote that the set $\sigma_{v_1}, \ldots, \sigma_{v_l}$. We first provide the recursion formula, prove its correctness and then discuss how to compute the recursion efficiently by adopting generating function. The recursion formula of the dynamic program is given by

$$
\Pr(v, \sigma_v, \theta_v) = \frac{\pi(\sigma_v)}{\prod_{j=1}^{l} \mu_{v,j}(\sigma_{v_j})} \sum_{\sigma_{v_j} \sim \sigma_v} \left( \delta(P^i(\sigma_v, \theta_v, [\sigma_{v_j}], \theta_{v_j})) \prod_{j=1}^{l} \Pr(v_j, \sigma_{v_j}, \theta_{v_j}) \right)
$$

(16)

where $P^i(\sigma_v, \theta_v, [\sigma_{v_j}], \theta_{v_j})$ denote the predicate $\sum_{j=1}^{l} |\sigma_{v_j} \cap T_{i-1}| - |\sigma_v \cap T_{i-1}| = \theta_v - \sum_{j=1}^{l} \theta_{v_j}$ and $\delta(p) = 1$ if $p = \text{true}$ and 0 otherwise.

We need the following lemma for proving the correctness of the recursion.

Lemma 4 Let $\theta_{v_j} = |\sigma_{v_j} \cap T_{i-1}|$ for $1 \leq j \leq l$, $\theta_v = |\sigma_v \cap T_{i-1}|$ if and only if $P^i(\sigma_v, [\sigma_{v_j}], [\theta_{v_j}]) = \text{true}$.

Proof: We will prove $|\bigcup_{j=1}^{h} \sigma_{v_j} | \cap T_{i-1}| = |\bigcup_{j=1}^{h} \sigma_{v_j} | \cap T_{i-1}| + |\sigma_v \cap T_{i-1}| - \sum_{j=1}^{h} |\sigma_{v_j} \cap T_{i-1}|$ for all $1 \leq h \leq l$ by induction on $h$. Note that the lemma is just the claim when $h = l$. If $h = 0$, the claim is trivially true. Assume the claim holds for any integer at most $h$. We have

$$
|\bigcup_{j=1}^{h+1} \sigma_{v_j} \cap T_{i-1}| = |\bigcup_{j=1}^{h+1} \sigma_{v_j} | \cap T_{i-1}| + |\sigma_{v_{h+1}} \cap T_{i-1}| - |\bigcup_{j=1}^{h} \sigma_{v_j} \cap T_{i-1}|)
$$

= $\sum_{j=1}^{h} |\sigma_{v_j} \cap T_{i-1}| + |\sigma_v \cap T_{i-1}| - \sum_{j=1}^{h} |\sigma_{v_j} \cap T_{i-1}| + |\sigma_{v_{h+1}} \cap T_{i-1}| - |\sigma_{v_{h+1}} \cap T_{i-1}|

= $\sum_{j=1}^{h+1} |\sigma_{v_j} \cap T_{i-1}| + |\sigma_v \cap T_{i-1}| - \sum_{j=1}^{h+1} |\sigma_{v_j} \cap T_{i-1}|.

The first equality follows from inclusion-exclusion principle and the second from the running intersection property. So, the claim holds. □

Theorem 7 The above recursion formula correctly compute the probability

$$
\Pr(v, \sigma_v, \theta_v) = \sum_{\tilde{\sigma}_v : |\tilde{\sigma}_v \cap T_{i-1}| = \theta_v} \Pr(\tilde{\sigma}_v).
$$

27
Proof: We prove this theorem by induction on the tree structure. Assume it holds for all \( v \)’s children, i.e.,

\[
\Pr(v_j, \sigma_{v_j}, \theta_{v_j}) = \sum_{\tilde{\sigma}_{v_j}:|\tilde{\sigma}_{v_j} \cap T_{i-1}|=\theta_{v_j}} \Pr(\tilde{\sigma}_{v_j})
\]

for all \( j, \sigma_{v_j} \) and \( \theta_{v_j} \). Then, plugging them into the recursion, we get

\[
\Pr(v, \sigma_v, \theta_v) = \frac{\pi(\sigma_v)}{\prod_{j=1}^{l} \mu_{v,v_j}(\sigma_{v,v_j})} \sum_{\sigma_{v_j} \sim \sigma_v} \sum_{[\theta_{v_j}] j} \left( \delta(P^i(\sigma_v, \theta_v, [\sigma_{v_j}], [\theta_{v_j}])) \prod_{j=1}^{l} \Pr(v_j, \sigma_{v_j}, \theta_{v_j}) \right)
\]

\[
= \frac{\pi(\sigma_v)}{\prod_{j=1}^{l} \mu_{v,v_j}(\sigma_{v,v_j})} \sum_{[\sigma_{v_j}] j} \sum_{[\theta_{v_j}] j} \left( \delta(P^i(\sigma_v, \theta_v, [\sigma_{v_j}], [\theta_{v_j}])) \sum_{[\tilde{\sigma}_{v_j}] j} \frac{\Pr(\tilde{\sigma}_{v_j})}{\prod_{j=1}^{l} \mu_{v,v_j}(\sigma_{v,v_j})} \right)
\]

\[
= \sum_{[\sigma_{v_j}] j} \sum_{[\theta_{v_j}] j} \left( \sum_{[\tilde{\sigma}_{v_j}] j} \delta(P^i(\sigma_v, \theta_v, [\sigma_{v_j}], [\theta_{v_j}])) \sum_{\sigma_v} \delta(\tilde{\sigma}_v \sim [\tilde{\sigma}_{v_j}]) \Pr(\tilde{\sigma}_v) \right)
\]

\[
= \sum_{\sigma_v} \delta([\tilde{\sigma}_v \cap T_{i-1}] = \theta_v) \Pr(\tilde{\sigma}_v) = \sum_{\tilde{\sigma}_v:|\tilde{\sigma}_v \cap T_{i-1}|=\theta_v} \Pr(\tilde{\sigma}_v)
\]

The fifth equality holds since \( \sigma_{v_j} \sim \sigma_v \) if and only if \( \tilde{\sigma}_{v_j} \sim \tilde{\sigma}_v \). The second last equality holds because of Lemma 4. Now, we prove the fourth equality \( \sum_{[\sigma_{v_j}] j} \sum_{[\theta_{v_j}] j} \left( \sum_{[\tilde{\sigma}_{v_j}] j} \delta(P^i(\sigma_v, \theta_v, [\sigma_{v_j}], [\theta_{v_j}])) \sum_{\sigma_v} \delta(\tilde{\sigma}_v \sim [\tilde{\sigma}_{v_j}]) \Pr(\tilde{\sigma}_v) \right) \). Written in probability form, it is \( \Pr(\tilde{\sigma}_v | \sigma_v) = \prod_{j=1}^{l} \Pr(\tilde{\sigma}_{v_j} | \sigma_v) \). Therefore, for any \( \tilde{\sigma}_v \) such that \( \tilde{\sigma}_v \sim [\sigma_{v_j}] \), we have

\[
\Pr(\tilde{\sigma}_v) = \Pr(\tilde{\sigma}_v | \sigma_v) \Pr(\sigma_v) = \pi(\sigma_v) \prod_{j=1}^{l} \Pr(\tilde{\sigma}_{v_j} | \sigma_v)
\]

\[
= \pi(\sigma_v) \prod_{j=1}^{l} \Pr(\tilde{\sigma}_{v_j}) = \pi(\sigma_v) \prod_{j=1}^{l} \frac{\Pr(\tilde{\sigma}_{v_j})}{\mu_{v,v_j}(\sigma_{v,v_j})}
\]

□

However, it is still not clear how (16) can be evaluated in polynomial time since it involves exponential number of terms. Suppose we have already computed \( \Pr(v_j, \sigma_{v_j}, \theta_{v_j}) \) for all \( v_j, \sigma_{v_j}, \theta_{v_j} \) and want to compute \( \Pr(v, \sigma_v, \theta_v) \) for some fixed \( v, \sigma_v, \theta_v \) now. First, we observe \( I = \theta_v - \sum_{j=1}^{l} |\sigma_{v,v_j} \cap T_{i-1}| - |\sigma_v \cap T_{i-1}| \) doesn’t depend on actual value of \( [\sigma_{v_j}] \) if \( [\sigma_{v_j}] \sim \sigma_v \). So, we can treat it as a constant \( h \) in this computation. The term \( \frac{\pi(\sigma_v)}{\prod_{j=1}^{l} \mu_{v,v_j}(\sigma_{v,v_j})} \) is also
fixed. Thus, we can rewrite the rest part of 16 as follows:

\[
N(v, \sigma_v, h) = \sum_{[\sigma_{v_j}] \sim \sigma_v} \sum_{[\theta_{v_j}]} \left( \delta(P^{h}(\sigma_v, \theta_v, [\sigma_{v_j}]), [\theta_{v_j}]) \prod_{j=1}^{l} Pr(v_j, \sigma_{v_j}, \theta_{v_j}) \right)
\]

\[
= \sum_{[\sigma_{v_j}] \sim \sigma_v} \sum_{\theta_{v_j}} \prod_{j=1}^{l} Pr(v_j, \sigma_{v_j}, \theta_{v_j})
\]

\[
= \sum_{\sum_{j=1}^{l} \theta_{v_j} = h} \prod_{j=1}^{l} \left( \sum_{[\sigma_{v_j}] \sim \sigma_v} Pr(v_j, \sigma_{v_j}, \theta_{v_j}) \right) = \sum_{\sum_{j=1}^{l} \theta_{v_j} = h} \prod_{j=1}^{l} \alpha_{v_j, \theta_{v_j}}
\]

where \( \alpha_{v_j, \theta_{v_j}} = \sum_{[\sigma_{v_j}] \sim \sigma_v} Pr(v_j, \sigma_{v_j}, \theta_{v_j}) \). Note that \( \alpha_{v_j, \theta_{v_j}} \) can be easily computed for all \( 1 \leq j \leq l \) and \( 0 \leq \theta_{v_j} \leq n \). Consider the following generating function:

\[
F(x) = \prod_{j=1}^{l} \left( \sum_{i=0}^{n} \alpha_{v_j, i}x^i \right)
\]

It is not hard to see the coefficient of \( x^h \) is exactly \( N(v, \sigma_v, h) \). Therefore, all computation we have to do is to evaluate \( a_{v_j, i} \) for all \( v_j \) and \( i \), construct and expand \( F(x) \).

Now, we estimate the running time of this algorithm. We need to run our dynamic program \( n \) times, each for a tuple \( t_i \). In the dynamic program, we need to construct and expand \( O(2^{C_v}) \) generating functions for each node \( v \) in \( T \), since one generating function is able to generate \( Pr(v, \sigma_v, \theta_v) \) for \( 1 \leq \theta_v \leq n \) and we have \( 2^{C_v} \) different \( \sigma_v \) values. To construct and expand a generating function takes \( O(l(n2^{C_v} + n^2)) \) time. Specifically, computing each \( a_{v_j, i} \) takes \( O(2^{C_v}) \), expand the polynomial takes \( O(ln^2) \) time. Therefore, we need \( O(2tw(n2^{tw} + n^2)|T|) \) time for each execution of dynamic program and \( O(2^{tw}n2(2^{tw} + n)|T|) \) time in total where \( tw \) is the treewidth of the junction tree \( T \).

### D Proof of Approximation Factor for WPRSC-\( t \) in the Hierarchical Framework

Remember from section (5), in the hierarchical framework, restricted soft clustering is performed only at the highest level. In all the lower levels hard clustering is performed. This maintains the condition, that each element can belong to at most \( t \) clusters. If at every level of the hierarchy a constant factor approximation algorithm for the corresponding clustering objective function is used and the number of hierarchies is constant, then we can prove that the overall guarantee of approximation is again bounded by a constant. It is to be noted here that the iterative algorithm for WPRSC-\( t \) that we designed, does not guarantee any approximation factor of the WPRSC-\( t \) objective function. Hence, naturally if used in the hierarchical framework, it cannot ensure a constant approximation. However, as the experimental results showed, in practice it performs extremely well. Also if a new algorithm for original WPRSC-\( t \) is developed in future, that ensures bounded! error of approximation, the algorithm can be plugged into the hierarchical framework. The proofs in this section are adapted from [21]. We prove the constant factor bound for level 2 hierarchy, where at the first level hard clustering is performed and the in the next level restricted soft clustering is used. The extension to more than 2 levels of hierarchy can be obtained in a similar fashion to that of [21]. We ignore the probabilities here. The proofs which consider the probabilities are identical to the proofs below and left to the reader.

Let \( S \) denote the set of elements to be clustered. \( S \) is divided into \( l \) arbitrary disjoint pieces \( \chi_1, \chi_2, \ldots, \chi_l \). For each \( \chi_i, i = 1, 2, \ldots, l \), \( k \) centers are obtained using a constant factor hard clustering algorithm. Each point in \( \chi_i \) is assigned to its closest center. Let \( \chi' \) be the \( lk \) centers obtained in the first hierarchy, where each center \( c \) is weighted by the number of points assigned to it. \( \chi' \) is clustered using a constant factor approximation algorithm for WPRSC-\( t \). The next two theorems establishes the constant factor approximation guarantee for the level 2 hierarchies.
Theorem 8  Consider any set of $n$ points arbitrarily partitioned into disjoint sets $\chi_1, \chi_2, \ldots, \chi_l$. The value of the optimum k-means, objective function is identical to the cost of the optimum solution for all $n$ points.

Proof: The proof is simple and similar to the Theorem 2.2 of [21]. For the sake of completeness, we give the proof here. Consider the centers used for the optimum solution. If each partition uses these centers, the cost of the solution will be exactly the cost of the optimum solution. This follows since the objective function for k-mean is the sum of distances to the nearest center for every point. However the set of centers chosen by the optimum solution need not be present in a partition. But since in k-means we allow, the centers points to be arbitrary points in the space, the theorem is proved.

Theorem 9  If the sum of the costs of the $l$ optimum k-means solutions for $\chi_1, \chi_2, \ldots, \chi_l$ is $C$ and if $C^*$ is the cost of the optimum WPRSC-t solution for the entire set $S$, then there exists a solution of cost at most $(C + C^*)$ to the new weighted instance $\chi'$.

Proof: Let the $t$ centers to which $i'$ is assigned to in the optimum WPRSC-t solution for $\chi'$ be $\tau(i', 1), \tau(i', 2), \ldots, \tau(i', t)$, with membership $v(i', 1), v(i', 2), \ldots, v(i', t)$. Further let $n_{i'}$ be the number of points assigned to the center $i'$. The cost of $\chi'$ can be expressed as $\sum_{i'} \sum_{j=1}^{t} v(i', j) \sum_{i} d(i', \tau(i', j)) n_{i'}$, where $d(x, y) = ||x - y||^2$. Each point $i' \in \chi'$ can be viewed as a collection of points $i$ assigned to $i'$ in the first level. Thus the cost of $\chi'$ can also be written as $\sum_{i} \sum_{j=1}^{t} v(i', j) \sum_{i} d(i', \tau(i', j))$.

Let the $t$ optimum cluster centers in which $i$ was assigned to be $\rho(i, 1), \rho(i, 2), \ldots, \rho(i, t)$, in $S$, with memberships $u(i, 1), u(i, 2), \ldots, u(i, t)$. Certainly, $\sum_{i} \sum_{j=1}^{t} v(i', j) \sum_{i} d(i', \tau(i', j)) \leq \sum_{i} \sum_{j=1}^{t} u(i, j) \sum_{i} d(i', \rho(i, j))$, since the former is optimum for $\chi'$. Now we have,

$$\sum_{i} \sum_{j=1}^{t} v(i', j) \sum_{i} d(i', \tau(i', j))$$

$$\leq \sum_{i} \sum_{j=1}^{t} u(i, j) \sum_{i} d(i', \rho(i, j))$$

$$\leq \sum_{i} \sum_{j=1}^{t} u(i, j) \left[ d(i', i) + d(i, \rho(i, j)) \right], \text{by triangle inequality}$$

$$= \sum_{i} \sum_{j=1}^{t} u(i, j) \left[ d(i', i) + \sum_{j=1}^{t} u(i, j) d(i, \rho(i, j)) \right]$$

$$\leq C + C^*$$

Here the last inequality follows from the fact $\sum_{j=1}^{t} u(i, j) \leq \sum_{j=1}^{t} u(i, j) = 1$. \qed

Now if an $\alpha$ approximation algorithm is used in the first level and a $\beta$ approximation algorithm is used on the next level, then the overall approximation factor from the previous two theorems is \(\frac{\beta(C + C^*)}{C} \leq \beta(1 + \alpha)\).