Query Ranking in Probabilistic XML Data

Lijun Chang, Jeffrey Xu Yu, Lu Qin
The Chinese University of Hong Kong, Hong Kong, China
{ljchang,yu,lqin}@se.cuhk.edu.hk

ABSTRACT
Twig queries have been extensively studied as a major fragment of XPath queries to query XML data. In this paper, we study PXML-RANK query, \((Q, k)\), which is to rank top-\(k\) probabilities of the answers of a twig query \(Q\) in probabilistic XML (PXML) data. A new research issue is how to compute top-\(k\) probabilities of answers of a twig query \(Q\) in PXML in the presence of containment (ancestor/descendant) relationships. In the presence of the ancestor/descendant relationships, the existing dynamic programming approaches to rank top-\(k\) probabilities over a set of tuples cannot be directly applied, because any node/edge in PXML may have impacts on the top-\(k\) probabilities of answers. We propose new algorithms to compute PXML-RANK queries efficiently and give conditions under which a PXML-RANK query can be processed efficiently without enumeration of all the possible worlds. We conduct extensive performance studies using both real and large benchmark datasets, and confirm the efficiency of our algorithms.

1. INTRODUCTION
Probabilistic XML (PXML) have been extensively studied recently [16, 11, 12, 22, 1, 20, 14, 5]. The issues studied widely cover the PXML models, semantics, data integration, constraints, expressiveness, query evaluation, query tractability, and complexity analysis. In this paper, we study a new research issue, and we study PXML rank query, \((Q, k)\), which is to rank top-\(k\) probabilities of answers of a twig query \(Q\) in PXML data. The new challenging is how to compute top-\(k\) probabilities of answers of a twig query \(Q\) in PXML in the presence of containment (ancestor/descendant) relationships, where an answer of a twig query can be judged using any score function as studied in [7, 19, 2]. In the presence of the ancestor/descendant relationships, the existing dynamic programming approaches to rank top-\(k\) probabilities over a set of tuples cannot be directly applied, because in the context of PXML any node/edge may possibly have impacts on the top-\(k\) probabilities of answers. To the best of our knowledge, it is the first work which studies ranking of twig query results in the context of PXML.

We study three types of PXML-RANK queries, \((Q, k)\), where \(Q\) can be a node query \((/A)\), a path query \((/A/ B)\), or a tree query \((/A//C/ B)\). The main contributions of this work are summarized below. First, we focus on node queries, and propose a new dynamic programming algorithm which can compute top-\(k\) probabilities for the answers of node queries based on the previously computed results in PXML data. Our algorithm considers the containment issue (ancestor/descendant) as well as the top-\(k\) probability and the score ranking (score functions) issues. We further propose optimization techniques to share the computational cost. Second, we show that our techniques can be used to support any path queries, and certain tree queries efficiently without enumerating all the possible worlds. We give conditions on the tree queries, and discuss our approaches. Third, we conduct extensive performance studies using both real and large benchmark datasets, and confirm the efficiency of our algorithms.

The remainder of the paper is organized as follows. Section 2 reviews the definition of probabilistic XML, and gives our problem statement. In Section 3, we discuss the technique details of answering a PXML-RANK node query, and in Section 4, we discuss how to extend the algorithms of node query to process all path queries and certain tree queries. Experimental studies are given in Section 5, followed by discussions on related work in Section 6. Finally, we conclude the paper in Section 7.

2. PXML AND PXML-RANK
An XML document can be modeled as a rooted, unordered, and node-labeled tree, \(T_X(V_X, E_X)\), where \(V_X\) represents a set of XML elements (nodes), and \(E_X\) represents a set of parent/child relationships (edges) between elements in XML. In an XML tree, a node is associated with a value \(x\) which belongs to a type (tag-name) \(X\), denoted as \(x_i \in X\). An XML tree is weighted if nodes and edges in the XML tree, \(T_X(V_X, E_X)\), are associated with non-negative weights, denoted as \(w(v)\) for \(v \in V_X\) and \(w(e)\) for \(e \in E_X\), respectively. In the following, an XML tree is a weighted XML tree unless otherwise specified.

A probabilistic XML (or PXML for short) defines a probability distribution over XML trees. Following the model given in [16], which is the PXML model in [14], in this paper, we define a PXML, \(T_P(V_P, E_P)\), over a weighted XML tree \(T_X(V_X, E_X)\). Here, \(V_P\) is a set of nodes \(V_P = V_X \cup V_D\), where \(V_X\) is a set of ordinary nodes that appear in an XML tree, and \(V_D\) is a set of distribution nodes (e.g. independent, mutually exclusive). Consider a node \(u\), which has a set of child nodes, \(V_u\), in an XML tree \(T_X\). In PXML, \(T_P\), the ordinary node, \(u\), may have several distribution nodes, as its child nodes, which specify the probability distributions over the disjoint subsets of the children of \(u\), \(V_u\). And \(E_P\) is a set of edges
with an edge weight.

1. An typed node (that its two child nodes, are other four independent distribution nodes. A node-weight, say $E$ a distribution edge. A positive probability is only associated with $E$ nodes, that appear in the incoming edge to $A$.

XML

A/ /D

An example of an $A$ tree, $X$, is shown in Fig. 1(a). There is a $D$-typed node $d$, two $A$-typed nodes ($a_1$ and $a_2$), four $B$-typed nodes ($b_1$ for $1 \leq i \leq 4$), and two $C$-typed nodes ($c_1$ and $c_2$). A $XML$ tree, $T_p$, based on the $XML$ tree $X$, is shown in Fig. 1(b). In $T_p$, $d$ has an independent distribution node as its child, which specifies that its two child nodes, $a_1$ and $a_2$ are independent. The probabilities of having $a_1$ and $a_2$ are 0.3 and 0.2, as indicated in the incoming edges to $a_1$ and $a_2$, respectively. In a similar fashion, there are other four independent distribution nodes. A node-weight, say $w_i(d)$, in $X$ can be specified as the node-weight associated with $w_i(d)$ in $T_p$, and an edge-weight, say $w_i(d, a_1)$ can be specified in the incoming edge to $a_1$ in $T_p$.

A $XML$ tree, $T_p$, is a compact representation of probability distribution over a collection of $XML$ trees, $T_{X_1}, T_{X_2}, \cdots$, which is generated in two steps.

First, we traverse the $XML$ tree, $T_p$, in a top-down fashion. When we visit an independent distribution node, $ind$, which has $l$ children, we divide $T_p$ into $2^l$ subtrees where each of them has a subset of the $l$ children. When we visit a mutually exclusive distribution node, $max$, which has $l$ children, we divide $T_p$ into $l$ subtrees where each of them has one child. We repeat the same procedure for each of the divided subtrees recursively, and obtain the set of $XML$ subtrees, where every connected $XML$ subtree shares the same root node of the $XML$ tree. Let $T_p'$ be one $XML$ subtree. The probability of $T_p'$, denoted as $Pr(T_p')$, is computed in Eq. (1).

$$Pr(T_p') = \prod_{u \in V_p'} Pr(u)$$

Here, if $u$ is a ordinary node, $Pr(u) = 1$. If $u$ is a distribution node, $Pr(u)$ is computed as follows. Let $u$ be a mutually exclusive node, and suppose $u$ has $l$ children. There are only two cases, selecting one of $l$ children or none because it is mutually exclusive. For the former, $Pr(u)$ is the probability associated with its outgoing edge to the selected child node. For the latter, it is one minus the summation of all the $l$ existence probabilities. Let $u$ be an independent node. Suppose $u$ has $n$ children, $u_1, u_2, \cdots, u_n$, out of $l$ children in total in the $XML$ tree $T_p$, $u_1, u_2, \cdots, u_n, u_{n+1}, \cdots, u_l$. $Pr(u)$ is computed by multiplying the existence probability associated with $u_i$, for $1 \leq u_i \leq u_n$, and the absence probability (one minus the existence probability) for $u_i$ for $u_n < u_i$.

Following the first step, for the $XML$ subtree (Fig. 1(b)) 4 intermediate $XML$ subtrees will be generated when visiting the first independent distribution node under the root node. Three are shown in Fig. 2 in addition to the $XML$ subtree which contains the root node $d$ only. Then, 6 $XML$ subtrees will be generated from the $XML$ subtrees Fig. 2(a)(b), respectively, and 36 $XML$ subtrees will be generated from the $XML$ subtree Fig. 2(c). In total, 49 $XML$ subtrees will be generated.

Second, for each of the $XML$ subtrees, $T_p'$, where $Pr(T_p') > 0$, we construct an $XML$ tree, denoted as $tree(T_p')$, by removing all distribution nodes/edges and connecting two ordinary nodes if there are distribution nodes/edges in between. The entire set of such $XML$ trees for a $XML$ tree is then uniquely identified. We denote it as $pwd(T_p') = \{T_{X_1}, T_{X_2}, \cdots\}$. The probability of $T_{X_i}$ is given by

$$Pr(T_{X_i}) = \sum_{tree(T_p') = T_{X_i}} Pr(T_p')$$

(2)

because the same $XML$ tree $T_{X_i}$ can be constructed from several $XML$ subtrees, $T_p'$. The set $pwd(T_p')$ forms the possible worlds of the probabilistic $XML$ ($XML$), $T_p$, and it satisfies the condition that $\sum_{T_{X_i} \in pwd(T_p')} Pr(T_{X_i}) = 1$.

A twig query is a fragment of $XPATH$ queries that can be represented as a query tree, $Q(V, E)$. Here, $V = (V_1, V_2, \cdots, V_n)$ is a set of nodes representing types (tag-names), and $E$ is a set of edges. An edge between two typed nodes, for example, $A$ and $D$, is either associated with an $XPATH$ axis operator / or / represent $A/D$ or $A/D$. Given an $XML$ tree $T_X$, the former is to retrieve all $A$ and $D$ typed nodes that satisfy the ancestor/descendant relationships, and the latter is to retrieve all $A$ and $D$ typed nodes that satisfy parent/child relationships. We call the former //edge and the latter --edge in short. As a special case, the root node in the query tree has an incoming // or //edge to represent an $XPATH$ query, // or //, suppose the root node is $A$-typed. The answer of a $n$-node twig query, $Q(V, E)$, against an $XML$ tree $T_X$, is a set of connected subtrees, where a connected subtree consists of $n$ nodes $(v_1, v_2, \cdots, v_n)$ in $T_X$, for $v_i \in V_i (1 \leq i \leq n)$, that satisfy all the structural relationships imposed by $Q$, and the minimal additional nodes/edges connecting the $n$ nodes as a connected subtree.

An example of an $XPATH$ query is $Q = //A[//C]/B$ (Fig. 1(c)). In this paper, we consider three classes of twig queries: (1) node query, (2) path query, and (3) tree query. For example, //A, //A//B,
and $\{A, C\}B$ are examples of node query, path query, and tree query, respectively.

A twig query, $Q$, against a PXML tree, $T_P$, can be processed by ignoring the existence of the distribution nodes/edges in $T_P$. The result is a set of XML trees, $M(Q, T_P) = \{\varphi_1, \varphi_2, \ldots, \varphi_N\}$. Let $\varphi_i$ be an XML tree in the result for a twig query. The score of $\varphi_i$, denoted as $\omega(\varphi_i)$, can be computed using any score function as studied in [7, 19, 2]. For simplicity we define it as the total sum of its node/edge weights in this paper, i.e. $\omega(\varphi_i) = \sum_{u \in \varphi_i} w_u(u) + \sum_{e \in \varphi_i} w_e(e)$. The top-$k$ probability of $\varphi_i$, $\rho(\varphi_i)$, is given below.

$$
\rho(\varphi_i) = \sum_{T_X \in \text{pred}(T_P)} \Pr(T_Xj) \quad (3)
$$

Here, $T_{Xj}$ is one XML tree in the possible worlds of the PXML tree $T_P$ (pwd($T_P$)), and the probability of $T_{Xj}$, $\Pr(T_{Xj})$, is computed using Eq. (2). The probability of $\varphi_i$ in the possible world, $T_{xj}$, is $\Pr(T_{Xj})$ if $\varphi_i$ is contained in $T_{Xj}$ and the score of $\varphi_i$, $\omega(\varphi_i)$, is at least the $k$-th largest value in $T_{Xj}$ ($\varphi_i \in \text{topk}(T_{Xj})$). It is important to note that several answers may appear in one possible world simultaneously. The $\rho(\varphi_i)$ is defined as the sum of such probability for every possible world where $\varphi_i$ is contained.

**Problem Statement** [Top-$k$ PXML Ranking (PXML-RANK)]: Let $T_P$ be a PXML tree with possible worlds pwd($T_P$). A PXML-RANK query, $(Q, k)$, is specified by a twig query, $Q$, and a positive number $k$, against $T_P$. It ranks the top-$k$ probabilities for the answers, $\varphi_i$, that satisfy the twig query $Q$.

The algorithm for processing a PXML-RANK query, $(Q, k)$, is outlined in Algorithm 1. First, it obtains a set of XML trees, $M = \{\varphi_1, \varphi_2, \ldots, \varphi_N\}$, that satisfy $Q$, against $T_P$ (line 1). It can be done over an XML tree which virtually treats every distribution path between two ordinary nodes in $T_P$ as an edge between the two ordinary nodes. Any efficient existing algorithms that process twig queries can be adapted [17]. Second, it sorts $M$ in the non-increasing order using the scores, such as $\varphi_i$ appears before $\varphi_j$ on the sorted $M$ if $\omega(\varphi_i) \geq \omega(\varphi_j)$ (line 2). Third, it calls P-RANK to compute the top-$k$ probabilities for all answers in $M$ (line 3). P-RANK returns $M$, which is a set of pairs $(\varphi_i, \rho(\varphi_i))$ for every answer $\varphi_i$ in $M$. Finally, it removes all answers $\varphi_i$ from $M$ if their top-$k$ probabilities are zero ($\rho(\varphi_i) = 0$) (line 4), and sorts $M$ in the non-increasing order of their top-$k$ probabilities ($\rho(\varphi_i)$) (line 5). Such $M$ is returned in line 6. It is worth noting that P-RANK is a time-consuming task in computing PXML-RANK queries. Given a set of answers, $\{\varphi_1, \varphi_2, \ldots, \varphi_N\}$, a naive approach needs to compute $\rho(\varphi_i)$ by enumerating all the possible worlds, pwd($T_P$), using Eq. (3).
and top-k probabilities. Unlike the x-Relation model, we consider one additional criterion, containment. In other words, for a node query, an answer \( \varphi_i \) can be an ancestor/descendant of another answer \( \varphi_j \) in the PXML tree \( T_P \). The additional criterion makes it difficult to apply the existing techniques [24, 10, 25] to solve the problem in our setting, even for node queries. Fig. 3 illustrates the main differences. First, in an x-Relation model, the tuples, \( \{ t_1, t_2, \ldots, t_N \} \), themselves are the context in which the top-k probabilities are computed. The current \( r_{i,j} \) for the sorted set of tuples \( \{ t_1, t_2, \ldots, t_{i-1}, t_i \} \) can be computed by the previously computed \( r_{1,j-1} \) and \( r_{1,j} \) for the sorted set of tuples \( \{ t_1, t_2, \ldots, t_{i-1} \} \). Note that all tuples are sorted based on their scores in a decreasing order. Every time for computing \( r_{i,j} \), the algorithm only needs to consider an additional tuple \( t_i \). The tuples \( \{ t_{i+1}, \ldots, t_N \} \), which have smaller scores than the current tuple \( t_i \), are not needed in the x-Relation model, because they do not affect \( r_{i,j} \) computing. However, in our problem setting, it becomes invalid that the nodes which have smaller scores than the current node are not relevant. As shown on the right side of Fig. 3, a node ("\( \circ \)) with a smaller score than the current node ("\( \uparrow \)) under consideration can be an ancestor or descendant of the current node. The existence/absence of every node may have impacts on the current node.

**Remark 1**: The top-k probabilities for answers, \( \{ \varphi_1, \varphi_2, \ldots, \varphi_N \} \), need to be determined in the context of the entire PXML tree.

### 3.2 An Overview of Our Approach

We outline our basic ideas for processing node queries in Fig. 4. We will discuss how to extend the basic ideas to process path queries and certain tree queries, and our optimization techniques later. Let the set of answers, \( M \), be \( M = \{ \varphi_1, \varphi_2, \ldots, \varphi_N \} \) which is processed by twigQuery(\( Q, T_P \)) in Algorithm 1. All such answers in \( M \) are identified in the PXML tree \( T_P \). It is shown in Fig. 4, in the initial stage, where tree nodes (not answers) and answers in \( T_P \) are indicated as "\( \times \)" and "\( \circ \)" respectively. Then, we compute \( p_{i,j} \) for every answer \( \varphi_i \in M \), for \( 1 \leq j \leq k \). The answer \( \varphi_i \) to be computed at an iteration is called the current. For the current \( \varphi_i \), we compute \( p_{i,j} \) in two steps, computing a \( r_{i,j} \)-like variable (step 1) and computing \( p_{i,j} \) (step 2).

In step 1, given the current answer \( \varphi_i \) (indicated as "\( \uparrow \)) in Fig. 4), the \( r_{i,j} \)-like variable we compute is \( r_{\varphi_i,j}^{\varphi_j} \). There exist main differences between \( r_{i,j} \) and \( r_{\varphi_i,j}^{\varphi_j} \). Recall that \( r_{i,j} \) is the probability that a randomly generated possible world from the sorted tuple set \( \{ t_1, \ldots, t_j \} \) has exactly \( j \) tuples. In Eq. (4), \( r_{i,j} \) is computed for the current tuple \( t_i \) using the answers that have a larger score than \( t_i \)’s by utilizing the sorted tuple set, \( \{ t_1, \ldots, t_{i-1} \} \), in the decreasing order of the scores. The value of \( i \) in \( r_{i,j} \) means the position of the \( i \)-th tuple itself on the sorted tuple set. In our problem setting, there does not exist such a sorted set. In order to simulate the sorted set, in other words, the set of answers that have a larger score than the score of the current \( \varphi_i (\omega(\varphi_j)) \), the superscript of \( r_{\varphi_i,j}^{\varphi_j} \) is introduced. Also, because the sorted tuple set does not exist, the indicator of \( i \) used in \( r_{i,j} \) for x-Relation model needs to be reconsidered. In our model, instead of \( i \), we use a subscript \( \varphi_i \) to indicate a subtree in PXML tree \( T_P \) rooted at node \( \varphi_i \). The meaning of \( j \) in \( r_{\varphi_i,j}^{\varphi_j} \) remains unchanged. In summary, \( r_{\varphi_i,j}^{\varphi_j} \) is the probability that a randomly generated possible world from the subtree of the PXML tree \( T_P \) rooted at \( \varphi_i \) has exactly \( j \) answers whose score is larger than the score of \( \varphi_i \).

In addition, there is a new issue on containment. For computing \( r_{\varphi_i,j}^{\varphi_j} \), we need compute all nodes \( v \in T_P \) including the answers in \( M \) as well as the current node \( \varphi_i \) itself. For this purpose, we introduce a general variable \( r_{v,j} \) where \( v \) is a node in the PXML tree \( T_P \), and \( h \) is a score. We compute \( r_{\varphi_i,j}^{\varphi_j} \) for every node \( v \in T_P \), based on the score \( \omega(\varphi_i) \), using dynamic programming. It is important to note that \( r_{\varphi_i,j}^{\varphi_j} \) can be computed based on the subtrees of the subtree rooted at \( v \) in \( T_P \). Upon completion of the computation, \( r_{\varphi_i,j}^{\varphi_j} \) are known for every node (including the current) and for \( 0 \leq j \leq k - 1 \). An answer \( \varphi_i \in M \) is marked as "\( \uparrow \)" in Fig. 4, if it has a larger score than the current’s \( \omega(\varphi_i) > \omega(\varphi_j) \).

In fact, up to this stage, \( r_{\varphi_i,j}^{\varphi_j} \) computed local, since it is computed based on the subtree rooted at \( \varphi_i \) and is not computed in the entire PXML tree \( T_P \) globally. Note that there is a path from the root of PXML tree \( T_P \) to the current \( \varphi_i \) as indicated by "\( \rightarrow \)" in Fig. 4. The \( r_{\varphi_i,j}^{\varphi_j} \) needs to be computed globally under the condition that the path "\( \rightarrow \)" must exist. The condition of the existence of such a path "\( \rightarrow \)" may affect some other \( r_{v,j} \), which in turn affect \( r_{\varphi_i,j}^{\varphi_j} \) for the current \( \varphi_i \).

In step 2, based on the condition that the path "\( \rightarrow \)" must exist, we compute global \( r_{\varphi_i,j}^{\varphi_j} \) and \( p_{i,j} \) for the current \( \varphi_i \), for \( 1 \leq j \leq k \). This is done by condensing the path "\( \rightarrow \)" into a node indicated as "\( \varepsilon \)" in Fig. 4. In other words, the PXML tree \( T_P \) is virtually transformed into another PXML tree \( T_P \) where the path "\( \rightarrow \)" in \( T_P \) becomes a node "\( \varepsilon \)" in \( T_P \) and all nodes that are connected to the nodes along the path "\( \rightarrow \)" in \( T_P \) are connected to the node "\( \varepsilon \)" in \( T_P \). It is worth noting that the global \( r_{\varphi_i,j}^{\varphi_j} \) and therefore \( p_{i,j} \) can be computed using the same dynamic programming because the subtree rooted at \( \varphi_i \) is the entire PXML tree. Fig. 5 illustrates the main idea. The left tree is \( T_P \) where the path "\( \rightarrow \)" consists of an ordinary node (\( ord \)), a mutually exclusive node (\( max \)), an independent node (\( ind \)), and the current node \( \varphi_i \) (root of the subtree (marked 4)). The right tree is \( T_P \). The subtree (marked 2) and its incoming edge are removed, because the \( max \) node implies that the subtree (marked 2) cannot exist. The subtree (marked 3) in \( T_P \) is directly linked to the root node in \( T_P \) with the same probability. The \( ord \) node is treated as an independent node with probability one to the subtree (marked 1), which is connected to the root node in \( T_P \).
Finally, given \( p_{i,j} \), \( \rho(\varphi_i) \) can be computed using the following Eq. (5) instead of Eq. (3).

\[
\rho(\varphi_i) = \sum_{j=1}^{k} p_{i,j} \quad (5)
\]

Below, we discuss how to compute \( \rho(\varphi_i) \) in a conditional probability viewpoint. Suppose \( M' \) is the set of all answers where every \( \varphi_i \in M' \) has a larger score than \( \varphi_i \), such as \( \omega(\varphi_i) > \omega'(\varphi_i) \). The probability of \( \varphi_i \) to appear in the top-\( k \) answers, \( \rho(\varphi_i) \), can be formulated as follows.

\[
\Pr(\varphi_i \text{ appears in the top-} k \text{ answers}) = \Pr(\varphi_i \text{ appears and at most } k-1 \text{ answers in } M') = \Pr(\varphi_i \text{ appears}) \times \Pr(\text{at most } k-1 \text{ answers in } M' \mid \varphi_i \text{ appears})
\]

In [5] Cohen et al. studied probabilistic XML with constraints (constraint satisfaction, query evaluation, and sampling), and the computation of \( \Pr(\text{at most } k-1 \text{ answers in } M' \mid \varphi_i \text{ appears}) \) can be transformed to a constraint satisfaction problem. The constraint satisfaction problem can be specified by modifying the PMXML tree \( T_P \) as follows: along the path from the root to \( \varphi_i \), for each edge \((u,v)\), (i) if \( u \) is a distribution node, then change the probability \( p_{\varphi_i}(u,v) \) to one, (ii) if \( u \) is a mutually exclusive node, then remove other children and the corresponding subtrees. Let the modified PMXML be \( T'_P \). Then, \( \Pr(\text{at most } k-1 \text{ answers in } M' \mid \varphi_i \text{ appears}) \) is equal to the probability that a random generated XML tree from \( T'_P \) satisfies the constraints that it contains at most \( k-1 \) answers in \( M' \).

Cohen et al. show that the constraint satisfaction problem is polynomial time solvable, and propose an algorithm to solve it. In this work, we compute \( \rho(\varphi_i) \), for \( 1 \leq i \leq N \) for the following main reasons. Although the constraint satisfaction problem is polynomial time solvable, it is proposed for general constraints, and is still time-consuming. For a different \( \varphi_i \), there is a different \( T'_P \), and the algorithm [5] needs to compute \( \rho(\varphi_i) \) individually. Instead we mainly consider how to share the costs of computing different \( \rho(\varphi_i) \)'s using specific constraints as discussed above.

### 3.3 An Example

In this section, from a different viewpoint (conditional probability viewpoint), we explain how to compute \( \rho(\varphi_i) \) using an example. Suppose \( \varphi_i \) is the current answer and \( M' \) is the set of answers where every \( \varphi_i \in M' \) has a larger score than \( \varphi_i \), such as \( \omega(\varphi_i) > \omega'(\varphi_i) \). The probability of \( \varphi_i \) to appear in the top-\( k \) answers can be formulated as follows.

\[
\Pr(\varphi_i \text{ appears in the top-} k \text{ answers}) = \Pr(\varphi_i \text{ appears and at most } k-1 \text{ answers in } M') = \sum_{j=0}^{k-1} \Pr(\varphi_i \text{ appears and exact } j \text{ answers in } M') = \sum_{j=0}^{k-1} \Pr(\varphi_i \text{ appears}) \times \Pr(\text{exact } j \text{ answers in } M' \mid \varphi_i \text{ appears})
\]

Here, \( \Pr(\varphi_i \text{ appears}) \) can be easily computed by multiplying all probabilities, \( \rho_{\varphi_i} \), along the path from root node of the PMXML tree \( T_P \) to \( \varphi_i \). The conditional probability of \( \Pr(\text{exact } j \text{ answers in } M' \mid \varphi_i \text{ appears}) \) is computed upon the condensed new PMXML tree \( T_P \).

Fig. 6(a) shows a PMXML \( T'_P \) that specifies the relationships among E-products (E-typed). There are 7 E-products \( e_i \) for \( 1 \leq i \leq 7 \). An E-product has a score (indicated in the brackets) as its performance. There are some uncertainties. The distribution node \( \text{ind}_1 \) implies that \( e_7 \) is a part of \( e_6 \) with probability 0.6 and is a part of \( e_4 \) with probability 0.7. The two are independent. The distribution node \( \text{mux}_1 \) implies that either \( e_6 \) is used in \( e_1 \) with probability 0.4 or is used in \( e_2 \) with probability 0.5, but cannot be used in both. The two are mutually exclusive to each other. Suppose a PMXML-RANK query \((Q,k)\) is issued against \( T_P \) (Fig. 6(a)), where \( Q = \langle /E \rangle \) and \( k = 1 \). The set of E-products to be ranked is \( M = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \) which is computed using twigQuery \((Q, T_P)\) in Algorithm 1. Note that \( M \) is sorted in the decreasing order of the scores (performance indicators). Next, all E-products in \( M \) will be ranked based on top-\( k \) probabilities, \( \rho(e_i) \), for \( 1 \leq i \leq 7 \), against the possible worlds \( \text{pwd}(T_P) \).

One of the possible worlds (e.g. XML tree \( T_X \)) is shown in Fig. 6(b). \( T_X \) is with the conditions that \( e_6 \) and \( e_4 \) coexist under the independent node \( \text{ind}_1 \), \( e_5 \) is present alone under the independent node \( \text{ind}_2 \), and \( e_1 \) is present under the mutually exclusive node \( \text{mux}_1 \). The probability of \( T_X = \Pr(T_X) = (0.4 \times 0.6) \times ((0.9 \times (1 - 0.8)) \times 0.7) = 0.03024 \) where 0.4 is the probability of the subtree rooted at \( e_6 \), and \((0.9 \times (1 - 0.8)) \) is the probability of the subtree rooted at \( e_4 \) in \( T_X \), respectively. It is infeasible to compute \( \rho(e_i) \) using Eq. (3) because it needs to enumerate all possible worlds \( \text{pwd}(T_P) \) and summarize the top-\( k \) probabilities for \( e_i \) to be ranked top. Note that the size of possible worlds can be exponential. Instead we efficiently compute \( \rho(e_i) \) using Eq. (5).

There are several obvious cases. (1) \( e_7 \) has the smallest score (93) in \( M \). The only possible world for \( e_7 \) to be ranked top is the XML tree with \( e_7 \) only. \( \rho(e_7) = (1 - 0.6) \times (1 - 0.7) = 0.12 \). (2) \( e_1 \) has the largest score in \( M \). If it appears in a possible world, it will be ranked top \( \rho(e_1) = 0.6 \times 0.4 = 0.24 \). (3) \( e_2 \) is ranked top if and only if \( e_1 \) does not appear in the possible worlds where \( e_2 \) appears. Note that \( e_1 \) and \( e_2 \) are mutually exclusive. In other words, if \( e_2 \) appears, then \( e_1 \) will not appear. \( \rho(e_2) = 0.6 \times 0.5 = 0.3 \). (4) \( e_4 \) can not be ranked top, because its ancestor \( e_5 \) has a higher score than \( e_5 \), and whenever \( e_5 \) appears \( e_4 \) will always appear. \( \rho(e_5) = 0 \).

Next consider a PMXML-RANK query \((Q,k)\) against \( T_P \) (Fig. 6(a)) where \( Q \) is the same \( \langle /E \rangle \) but \( k = 3 \). The set of answers to be ranked is the same \( M = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \). We discuss computation of top-\( k \) probability, \( \rho(e_6) \), for \( e_6 \) to be ranked top-\( k \). Let \( e_6 \) be the current with \( \omega(e_6) = 94 \). The E-products that have larger scores than \( \omega(e_6) = 94 \) are \( M_h = \{e_1, e_2, e_4, e_5\} \) which we call h-answers.

Consider \( e_6 \). The PMXML tree \( T'_P \) can be divided into several parts, \( P, T_1, \) and \( T_2 \), as shown in Fig. 7. Here, \( P \) represents \( \langle /E \rangle \) which must appear because \( e_6 \) must appear. Obviously, \( \Pr(\text{e}_6 \text{ appears}) = \Pr(\text{P appears}) = 0.6 \). We have \( \rho(e_6) = \Pr(\text{P appears and at most 2 h-answers appear}) = \sum_{j=0}^{2} \Pr(\text{P appears and exact } j \text{ h-answers appear}) = \sum_{j=0}^{2} \Pr(\text{P appears}) \times \Pr(\text{exact } j \text{ h-answers appear | P appears}) \). Note that \( \Pr(\text{P appears}) = 0.6 \). We explain how to
compute 
Pr(exact $j$ h-answers appear | $P$ appears) 
(6) 
for $0 \leq j \leq 2$ below. Recall that, given the current $e_6$, in our notation, the probability that exact $j$ answers from $M_0$ appear in the subtree rooted at $v$ is denoted as $r_e^{\omega(e)} = r_e^{v}$. For simplicity, we use $r_{e,j}$ below.

**Case-1** ($j = 0$): Eq. (6) equals to $Pr(0$-h-answer in $T_1$ appears) $\times$ $Pr(0$-h-answer in $T_2$ appears). For the first part, it is $r_{mux1,0}$. The only situation that 0 h-answer appears in the subtree rooted at $mux1$ ($T_1$) is that none of $e_1$ and $e_2$ appears. Since $e_1$ and $e_2$ are mutually exclusive, we have $r_{mux1,0} = 1.0 - \rho_e(mux1, e_1) - \rho_e(mux1, e_2) = 1.0 - 0.4 = 0.6$. The second part is equal to $Pr$ (the absence of edge $(ind1, e_4)$) $\times$ $Pr$ (the existence of edge $(ind1, e_2)$) $\times$ $Pr$ (the existence of edge $(e_3, e_4)$). Here, $e_3, e_4 \neq 0$ because $e_4$ must appear when considering the subtree root at $e_4$, so the second part is $0.3 \times 0.7 = 0.21$. Combining the two parts, Eq. (6), for $j = 0$, equals to $0.1 \times 0.3 = 0.03$.

**Case-2** ($j = 1$): Eq. (6) equals to $Pr(0$-h-answer in $T_1$ appears) $\times$ $Pr(1$-h-answers in $T_2$ appear) $\times$ $Pr(1$-h-answers in $T_1$ appears) $\times$ $Pr(0$-h-answer in $T_2$ appears). Note that $Pr(0$-h-answer in $T_1$ appears) = 0.1 and $Pr(0$-h-answer in $T_2$ appears) = 0.3 are computed in Case-1.

Here, $Pr(1$-h-answer in $T_1$ appears) = $r_{mux1,1}$. The only situation that 1 h-answer is the subtree rooted at $mux1$ ($T_1$) appears is that either $e_1$ appears or $e_2$ appears. Since $e_1$ and $e_2$ are mutually exclusive, we have $Pr(1$-h-answer in $T_2$ appears) = $r_{mux1,1} = \rho_e(mux1, e_1) + \rho_e(mux1, e_2) = 0.45 = 0.9$.

Also $Pr(1$-h-answer in $T_2$ appears) = $Pr$(the existence of edge $(ind1, e_4)$) $\times$ $r_{e,1}$ = 0.7 $\times$ $r_{e,1}$. The only situation that 1 h-answer appears in the subtree root at $e_4$ is that 0 h-answer appears in the subtree rooted at $ind1, i.e., r_{ind2,0} = 0.6$. It means that neither $e_5$ nor $e_3$ appears, i.e., $r_{ind2,0}$ = $(1 - \rho_e(ind2, e_5)) \times (1 - \rho_e(ind2, e_3)) = (1 - 0.9) \times (1 - 0.8) = 0.02$. Then, we have $Pr(1$-h-answer in $T_2$ appears) = $0.7 \times 0.02 = 0.014$.

Therefore, Eq. (6), for $j = 1$, equals to $0.1 \times 0.014 + 0.9 \times 0.3 = 0.2714$.

**Case-3** ($j = 2$): Eq. (6) equals to $Pr(0$-h-answer in $T_1$ appears) $\times$ $Pr(2$-h-answers in $T_2$ appear) $\times$ $Pr(1$-h-answer in $T_1$ appears) $\times$ $Pr(1$-h-answer in $T_2$ appears) $\times$ $Pr(2$-h-answers in $T_1$ appear) $\times$ $Pr(0$-h-answer in $T_2$ appear) = $0.1 \times Pr(2$-h-answers in $T_2$ appear) $\times$ $0.9 \times 0.014 + Pr(2$-h-answers in $T_1$ appear) $\times$ $0.3$.

The probabilities for $j < 2$ are computed already in Case-1 and Case-2.

Here, $Pr(2$-h-answers in $T_1$ appear) = $r_{mux1,2}$. To have 2 h-answers appear in the subtree root at $mux1$, both $e_1$ and $e_2$ must appear. This is impossible because $e_1$ and $e_2$ are mutually exclusive. We have $Pr(2$-h-answers in $T_1$ appear) = $r_{mux1,2} = 0$.

On the other hand, $Pr(2$-h-answers in $T_2$ appear) = $Pr$(existence of edge $(ind1, e_4)$) $\times$ $r_{e,2} = 0.7 \times r_{e,2}$. The only situation that 2 h-answers appear in the subtree root at $e_4$ is that 1 h-answer appears in the subtree root at $ind1, i.e., r_{ind2,1} = r_{ind2,1}$. It means that either (a) $e_5$ appears but $e_2$ does not appear or (b) $e_5$ does not appear but $e_3$ appears. We have $r_{ind2,1} = \rho_e(ind2, e_5) \times (1 - \rho_e(ind2, e_3)) + (1 - \rho_e(ind2, e_5)) \times \rho_e(ind2, e_3) = 0.9 \times (1 - 0.8) + (1 - 0.9) \times 0.8 = 0.26$. We have $Pr(2$-h-answers in $T_2$ appear) = $0.7 \times 0.26 = 0.182$. Therefore, Eq. (6), for $j = 2$, equals to $0.1 \times 0.182 + 0.9 \times 0.04 + 0.3 = 0.0308$.

With all the three cases, $rho_e = Pr(\text{P appears}) \times \sum_{j=0}^{\infty} Pr(\text{exact j h-answers appear | P appears}) = 0.6 \times (0.03 + 0.2714 + 0.0308) = 0.19932$. For the PXML-RANK query $(\langle E, 3 \rangle$ against $T_P$ (Fig. 6(a)), the ranking is shown below.

<table>
<thead>
<tr>
<th>$\rho(e_4)$</th>
<th>$\rho(e_5)$</th>
<th>$\rho(e_7)$</th>
<th>$\rho(e_9)$</th>
<th>$\rho(e_2)$</th>
<th>$\rho(e_1)$</th>
<th>$\rho(e_6)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.07</td>
<td>0.56</td>
<td>0.37924</td>
<td>0.35784</td>
<td>0.3</td>
<td>0.24</td>
<td>0.19932</td>
</tr>
</tbody>
</table>

### 3.4 Algorithms

The algorithm to compute P-RANK for node queries is given in Algorithm 2. It takes three inputs. The PXML tree $T_P$, the top-k value $k$, and a set of answers $M = \{\phi_1, \phi_2, \ldots, \phi_N\}$ which is sorted in the decreasing order of their scores $\omega(\cdot)$. For each $\phi_i$ (the current), in a for-loop, it processes the following tasks. It computes its local $r^{\omega(\phi)}$ using dynamic programming (line 2). It identifies the path from the root of $T_P$ to the current $\phi_i$ ("\text{"}\phi_i\text{"}"), and assigns it to $P$ (line 3). Then, it virtually reconstructs $T_P$ to $T_P$ by condensing the path $P$ into a node which is the root of $T_P$, $\phi'$ (line 4-5). It computes the global $r^{\omega(\phi')}$ in line 6 using dynamic programming where children($\phi'$) indicates the children of the root node $\phi'$. In order to compute $p_{e,j}$ where $i$ implies $\phi_i$, it counts how many nodes on the path $P$ ("\text{"}\phi'\text{"}") that are with a score greater than $\omega(\phi_i)$ (line 7). The algorithm computes $p_{e,j}$ (line 8-10), and then computes $\rho(\phi_i)$ in line 11.
We explain H-PROB (used in line 2, Algorithm 2). The H-PROB algorithm is given in Algorithm 3. It takes three inputs, the PXML tree $T_P$, the current answer $\varphi$, and an integer $k$. The main task of H-PROB is to compute local $r^h_{v,j}$ where $h = \omega(\varphi)$ in a bottom-up fashion. For non-leaf nodes, it further calls H-TOPK (Algorithm 4) to compute using dynamic programming. The H-TOPK algorithm takes four inputs to compute $r^h_{v,j}$, for $0 \leq j < k$. Here, $h$ is the score of the current $\varphi$, $v$ is the non-leaf node in question, $\{v_1, \ldots, v_l\}$ are the children of $v$. It assumes that $r^h_{v,j}$, for $1 \leq i \leq l$, have already been computed. (Note that the H-PROB algorithm uses a bottom-up traversal to compute.) There are several cases handled in the H-TOPK algorithm: (1) $v$ is a mutually exclusive distribution node (line 1-3), (2) $v$ is an independent distribution node (line 4-11), and (3) $v$ is an ordinary node (line 12-20).

Below, we explain the case when $v$ has 2 children ($l = 2$): $\{v_1, v_2\}$ (Fig. 8(a)). Note that both $r^h_{v_1,j}$ and $r^h_{v_2,j}$ have been computed.

If $v$ is a mutually exclusive node, for $j > 0$, $v_1$ and $v_2$ cannot appear simultaneously. $r^h_{v,j}$ can be computed in two cases, either with the subtree rooted at $v_1$ or with the subtree rooted at $v_2$. That is, $r^h_{v,j} = \rho_e(v, v_1) \cdot r^h_{v_1,j} + \rho_e(v, v_2) \cdot r^h_{v_2,j}$. For $j = 0$, it needs to consider an additional case that none of the two subtrees are selected. $r^h_{v,0}$ is computed as follows. $r^h_{v,0} = \rho_e(v, v_1) \cdot r^h_{v_1,0} + \rho_e(v, v_2) \cdot r^h_{v_2,0} + (1 - \rho_e(v, v_1) - \rho_e(v, v_2))$.

If $v$ is an independent node, the existence of $v_1$ is independent from each other where $i = 1, 2$. $v$ chooses either $v_1$ or $v_2$, or both, or none. Let $B^h_{v,i}$ be the probability that a randomly generated XML tree, from the branch $(v, v_i)$ which consists of the edge $(v, v_i)$ and the subtree rooted at $v_i$, contains exactly $j$ answers with score greater than $h$. When $j > 0$, $B^h_{v,i} = \rho_e(v, v_i) \cdot r^h_{v_i,j}$, and, when $j = 0$, $B^h_{v,i} = \rho_e(v, v_i) \cdot r^h_{v_i,0} + (1 - \rho_e(v, v_i))$. Then, $r^h_{v,j}$ includes the cases that, in a randomly generated XML tree, one branch, say $(v, v_1)$, contains exactly $\leq j$ answers with a score greater than $h$, and the other branch, say $(v, v_2)$, contains exactly $\leq j - 1$ answers with a score greater than $h$. We design our algorithm for handling general $l$ children of a given node $v$. If $v$ is an independent node or an ordinary node, in order to compute $r^h_{v,j}$, we need to consider how many exact answers out of $j$ answers are from which subtrees by enumerating all the sequences, $i_1, \ldots, i_l$, such that $\sum_{i=1}^l i_1 = j$ for any fixed $j \in \{0, \ldots, k - 1\}$. We handle $l$ children of a given node $v$, if it is an independent/ordinary node, by transforming the node $v$ with $l$ children into a left-deep binary subtree, as shown in Fig. 8(b).
has two children \((v_1, v_2)\). The node \(v\) is a non-leaf node, when \(v\) is an independent node, then \(u_1, \ldots, u_{i-2}\) are ordinary nodes with weight \(0\). If \(v\) is an independent node, then \(u_1, \ldots, u_{i-2}\) are also independent nodes, the probability \(\rho_u(v, v_i)\) is specified on the incoming edge to \(v_i\). All other edges have probability one. It can be verified that the transformed left-deep tree will give the same result. The H-\textsc{topk} algorithm (Algorithm 4) is designed using the left-deep binary tree, where \(v_1\) is treated as \(u_0\) and \(v\) is treated as \(u_{i-1}\).

### Optimization-I

As indicated in the \textsc{p-rank} algorithm, it needs to call the H-\textsc{prob} algorithm for every answer \(\phi_i\) in \(M = \{\phi_1, \phi_2, \ldots, \phi_N\}\), which is sorted in the decreasing order of the scores. The cost of computing using dynamic programming is costly. In fact, the cost can be shared between successive calls of H-\textsc{prob}, e.g., H-\textsc{prob} \((T_1, \phi_1, k)\) and H-\textsc{prob} \((T_2, \phi_{i+1}, k)\). Consider the same \textsc{pxml-rank} query (\((/A/ /B), k\)) against the \textsc{pxml} tree \(T'_p\) (Fig. 6(a)). Table 1 shows the results of \(r_{v,j}^h\) when computing the two consecutive answers, \(e_4\) and \(e_5\). It shows that the maximum when computing \(r_{v,j}^h\) for \(e_5\) remain unchanged, given \(r_{v,j}^h\) computed for \(e_4\). The possible change part is highlighted in the dot rectangles in Fig. 9, which is along the path from the root to the previous answer. A lemma is given below.

**Lemma 1:** Let H-\textsc{prob} \((T_F, \phi_i, k)\) and H-\textsc{prob} \((T_F, \phi_{i+1}, k)\) be two consecutive executions, for two answers, \(\phi_i, \phi_{i+1}\) in the sorted answer set \(M\). When \(\omega(\phi_i) \neq \omega(\phi_{i+1})\), the values \(r_{v,j}^{\omega(\phi_i)}\) and \(r_{v,j}^{\omega(\phi_{i+1})}\) are identical for the nodes that are not on the path from the root of \(T_F\) to the node \(\phi_i\). When \(\omega(\phi_i) = \omega(\phi_{i+1})\), for all nodes, \(r_{v,j}^{\omega(\phi_i)} = r_{v,j}^{\omega(\phi_{i+1})}\).

Due to space limit, we omit the proof. The \textsc{p-rank} and H-\textsc{prob} algorithms only need to be slightly changed to adapt the optimization-I.
As an example, consider the PXML tree in Fig. 12, and a PXML-RANK query, \((Q, k)\) where \(Q = \|A\|/\|B\|/\|C\|\), and \(k = 1\). There are 10 answers. For simplicity, we use a 3-tuple to indicate a resulting subtree for the PXML-RANK query. One resulting subtree is \(r_1 = (a_1, b_1, c_1)\). There are other 9 resulting subtrees rooted at \(a_2\) with any one of the three \(b_i\), for \(2 \leq i \leq 4\), and any one of the three \(c_j\), for \(2 \leq j \leq 4\). As one example, consider \(r_2 = (a_2, b_2, c_3)\) where \(a_2\) has three children \(\{b_2, b_3, b_4\}\), \(b_2\) is in the subtree rooted at \(b_2\) and \(c_3\) is in the subtree rooted at \(b_3\). The fact states that our dynamic programming techniques cannot be used to efficiently compute top-k probabilities even for \(r_1\) over the 10 subtrees. It is because we cannot compute \(r_{a_2,b_j}\) based on the values of \(r_{a_3,b_j}\), for \(2 \leq i \leq 4\), that are obtained for the children. Any combinations are possible, and we need to enumerate all possible worlds.

We give conditions. A general PXML-RANK tree query, \((Q, k)\), can be computed in polynomial time if one of the conditions are satisfied. The conditions are imposed on the set of answers, \(M = \{\varphi_1, \varphi_2, \ldots, \varphi_N\}\), to be ranked, which is generated by twigQuery\((Q, T_p)\) in Algorithm 1. The conditions can be checked when processing twig queries without high overhead. Below, for a given tree query \(Q\), we call a path in \(Q\) a primary path and denote it as \(P_Q\). For example, \(\|A\|/\|B\|/\|C\|/\|B\|\). Note that an answer \(\varphi_i\) can be a subtree.

1. The edges of results do not overlap with each other. \(\varphi_i \cap \varphi_j = \emptyset\) for \(i \leq j\).
2. Every edge, \(e'\), of every answer \(\varphi_i \in M\), which is not on the primary path \(P_Q\), must be associated with \(\rho_e(e') = 1\). (An ordinary edge, \(e'\), is considered as an edge with \(\rho_e(e') = 1\).
3. Let \(\varphi_i\) and \(\varphi_j\) be two different answers. Suppose \(\beta_i\) and \(\beta_j\) are the paths in \(\varphi_i\) and \(\varphi_j\) that match \(P_Q\), respectively. There exist two subtrees \(\gamma_i = \beta_i - \beta_j\) and \(\gamma_j = \beta_j - \beta_j\) where \(\varphi_i \cup \varphi_j\) and \(\varphi_j \cup \varphi_j\). If there exist a node \(v_i\) on the path \(\beta_j\) that is a descendant of a node \(v_j\) on the path \(\beta_j\), then \(\gamma_i\) and \(\gamma_j\) must be identical.

As an example, consider the PXML tree, \(T_p\) in Fig. 1(b), and the twig query \(Q = \|A\|/\|C\|/\|B\|\). The answer set \(M = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}\) where \(\varphi_1 = \{(a_1, b_1), (a_1, c_1)\}, \varphi_2 = \{(a_1, b_1), (b_1, b_2), (a_1, c_1)\}, \varphi_3 = \{(a_2, b_2), (a_2, c_2)\}\), and \(\varphi_4 = \{(a_2, b_2), (b_3, b_2), (a_1, c_1)\}\) where \(P_Q = \|A\|/\|B\|\). The four answers, \(\varphi_i\), for \(1 \leq i \leq 4\), do not satisfy the first and the second condition. But they satisfy the third condition. For example, consider \(\varphi_1\) and \(\varphi_2\), \(\beta_1 = (a_1, b_1)\) and \(\beta_2 = (a_1, b_1)\), and \(\gamma_1 = (a_1, c_1)\) where \(\varphi_3\) and \(\varphi_4\) have identical \(\beta_2\), \(\gamma_1\). There exist \(b_1\) on \(\beta_1\) that is a descendant of a node \(a_1\) on \(\beta_2\).

The conditions allow us to compute a PXML-RANK tree query using our dynamic programming techniques. In brief, when either the second or the third condition is satisfied, we can compute top-k probabilities for a subtree \(\varphi_i\) or to compute top-k probabilities for the lowest node of \(\varphi_i\), on the primary path \(P_Q\) in the same way of processing path queries. Processing a tree query, when the first condition is satisfied, is complicated, because some nodes of a subtree answer \(\varphi_i\) can be ancestors of another subtree answer \(\varphi_j\). We discuss our main ideas below, due to the space limit.

Let \(M = \{\varphi_1, \varphi_2, \ldots\}\) be a set of the subtree answers to be ranked, and let \(\varphi\) be the current subtree. We call a subtree answer \(\varphi_i\) a-h-answer if it has a larger score than the current’s. For each \(\varphi_i\), we use \(root(\varphi_i)\) to denote the root of \(\varphi_i\). \(P_{\varphi_i}\) denotes the path from the root node in the PXML tree to \(root(\varphi_i)\), and \(Pr_{\varphi_i} = \Pi_{e \in \varphi_i} \rho_e(e)\), if there are no mutually exclusive nodes in \(\varphi_i\), otherwise \(Pr_{\varphi_i} = 0\). We use \(\varphi_{v,j}\) instead of \(\varphi_{v,j}^{(e)}\) for short below. First, for computing the probability \(Pr(j\text{-h-answers appear }| \varphi \text{ appears})\), we set all \(\rho_i(e) = 1\) for the edges in \(P_P\), and remove the branches which are mutually exclusive with any nodes in \(P_P \cup \varphi\). We cannot simply condense \(P_P\) to a node, because \(P_P\) may contain some part of other h-answers. The process of setting \(\rho_i(e) = 1\) serves the same purpose of condensing a path into a node. Second, we compute \(\varphi_{v,j}\) where a subtree answer \(\varphi_i\) is rooted on \(v\). Note that \(v\) is an ordinary node. We condense \(\varphi_i\) to a virtual node \(v'\) and remove the branches that are mutually exclusive with any nodes in \(\varphi_i\). (see Fig. 13). The true \(\varphi_{v,j}\) is computed as follows: \(\varphi_{v,j} = L_{\varphi_{v,j}} - \varphi_{v,j} \times \varphi_{v,j} - \varphi_{v,j} \times \varphi_{v,j-1}\), where the \(\varphi_{v,j}\) appears on the right side is computed on the left tree in Fig. 13 where every node/edge in \(\varphi_i\) is considered separately. The existence of the entire \(\varphi_i\) is ensured on the right tree in Fig. 13. It can be easily handled when multiple subtree answers are rooted at the same node \(v\).

5. EXPERIMENTAL STUDIES

We conduct extensive experiments to test the performance of our algorithms. We have implemented the PXML-RANK algorithm. The main algorithm to be tested is P-RANK. We have implemented P-RANK without Optimization-I and Optimization-II, the P-RANK algorithm using Optimization-I, and the P-RANK algorithm using both Optimization-I and Optimization-II. We denote them as pRANK, pRANK-I and pRANK-II, respectively. All algorithms were implemented in C++. We conducted all the experiments on a 2.8GHz CUP and 2GB memory PC running XP.

We use two real datasets, DBLP (http://dblp.uni-trier.de/xml/) and Mondial (http://www.informatik.uni-freiburg.de/~may/lopix/lopix-mondial.html), and the synthetic XML benchmark dataset XMark (http://monetdb.cwi.nl/xmark/) for testing. For XMark, we also generate many datasets with different sizes. For each XML dataset used, we generate the corresponding PXML tree, using the same method as used in [14]. We visit the nodes in the original XML tree from top to bottom. For each node \(v\) visited, we randomly choose some distribution nodes with
random types and probability distributions to be the children of \( v \), then for the original children of \( v \), we choose some of them to be the children of the new generated distribution nodes. We control the percentage of the distribution nodes to generate different PXML trees for each dataset.

In answering a PXML-RANK query, we first compute all answers using a modified twig pattern matching algorithm based on [17]. The algorithm we use can process the entire PXML tree in a stream-

manner, and therefore does not need to keep the entire PXML tree in memory. Then we prune nodes, \( v \), on the PXML tree if the subtree rooted at \( v \) does not effect the ranking, and get another projected PXML tree. We run our three algorithms pRank, pRank-I and pRank-II, to compute top-k probabilities for all results over the projected PXML tree if they satisfy one of our conditions. For each test, we record the time and space consumption of all algorithms.

The time consumption consists the query processing time to generate all the results, the projection time and the time for computing top-k probabilities for all results. The main space consumption is caused by maintaining \( \omega_{v,j} \) values. For pRank, \( \omega_{v,j} \) values for a node \( v \), can be released when \( \omega_{v,j} \) has been computed where \( v \) is the parent of \( v \). For pRank-I and pRank-II, in order to share the computational cost, \( \omega_{v,j} \) values need to be kept for computing \( \omega_{v,j} \). pRank-II consumes more memory because Optimization-II needs to maintain complete binary trees.

For the DBLP dataset, the original XML tree has 13,318,516 nodes with 41 different tags and the maximum depth of 6. We range the percentage of distribution node from 10% to 50% and generate 5 different PXML trees. The queries used for the DBLP dataset are listed in Table 2 from D1 to D5, with combination of both // and / operators. We list them in increasing order of result size. The parameters used for testing DBLP dataset are listed in Table 3, where DistNode means the percentage of distribution node. For the Mondial dataset, the original XML tree has 70,450 nodes with 51 different tags and the maximum depth of 5. The queries used and parameters with default values are listed in Table 2 from M1 to M5 and Table 3 respectively. For the XMark datasets, we generate 5 different datasets with different sizes for testing. There are 77 different tags with the maximum depth of 12 for each of the generated XML trees. We set the percentage of distribution node to be 30% and convert them to the corresponding PXML tree. The number of nodes for each PXML tree is shown in the last row of Table 3. The queries used and parameters with default values are listed in Table 2 from X1 to X5 and Table 3 respectively.

### Table 2: Queries used for all datasets

<table>
<thead>
<tr>
<th>ID</th>
<th>Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>dblp/book[.//author/key]</td>
<td>4,684</td>
</tr>
<tr>
<td>D2</td>
<td>dblp/article[.//title/sub]//key</td>
<td>3,928</td>
</tr>
<tr>
<td>D3</td>
<td>dblp/proceeding[.key//series//href]</td>
<td>4,039</td>
</tr>
<tr>
<td>D4</td>
<td>dblp/motioncollection//key/author</td>
<td>8,832</td>
</tr>
<tr>
<td>D5</td>
<td>dblp/article[.//cite//label]//key</td>
<td>13,785</td>
</tr>
</tbody>
</table>

### Table 3: Parameters used for testing

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>DistNode(All)</td>
<td>10%, 20%, 30%, 40%, 50%</td>
<td>30%</td>
</tr>
<tr>
<td>Top-k (All)</td>
<td>10, 20, 30, 40, 50</td>
<td>30</td>
</tr>
<tr>
<td>Query(DBLP)</td>
<td>D1, D2, D3, D4, D5</td>
<td>D3</td>
</tr>
<tr>
<td>Query(Mondial)</td>
<td>M1, M2, M3, M4, M5</td>
<td>M3</td>
</tr>
<tr>
<td>Query(XMark)</td>
<td>X1, X2, X3, X4, X5</td>
<td>X5</td>
</tr>
<tr>
<td>Node Number(XMark)</td>
<td>0.5, 1, 1.5, 2, 2.5 (( \times 10^4 ))</td>
<td>1.5</td>
</tr>
</tbody>
</table>

### 5.1 Test-DBLP

Fig. 14 shows the testing results over DBLP datasets. From Fig. 14(a) and 14(b), we know that when the percentage of distribution nodes increases, both the time and memory consumption for the three algorithms marginally increase. pRank-II is more than 300 times faster than pRank although cost about three times more memory than pRank. pRank-I is more than 10 times faster than pRank although cost about 2 times more memory than pRank. Fig. 14(c) and Fig. 14(d) show that when the number of results increases, the time and memory used for the three algorithms do not necessarily increase. It is because the increasing of the number of results does not mean that the size of the projected PXML tree to be tested also increases. The time for all the three algorithm is influenced by both the number of results and the size of the projected PXML tree. The memory consumption for all the three algorithms reflects the size of the projected tree. In Fig. 14(e) and Fig. 14(f), we can see that, when \( k \) increases, the time for all the three algorithms will increase. The memory consumption for pRank-I and pRank-II will increase linearly with \( k \), while the number of results for pRank-II is not influenced by \( k \). pRank-II is also much more faster (about 100 times faster) than pRank although cost some more memory (not larger than 8 times more). The performance of pRank-I is between the other two algorithms.

### 5.2 Test-Mondial

Fig. 15 shows the performance of the three algorithms over the Mondial dataset. Fig. 15(a) and Fig. 15(b) show that when the percentage of distribution nodes increases, the time and memory consumption for all the three algorithms will increase. pRank-II is more than 100 times faster than pRank, and is 2 times faster than pRank-I. The memory consumption of pRank-II and pRank-I are almost the same and are 3 times more than pRank. In Fig. 15(c) and Fig. 15(d), when the number of results increases in the Mondial dataset, the size of the projected tree will also increase (which is reflected by the increasing of memory consumption), so the time for all the three algorithms will increase. Fig. 15(e) and Fig. 15(f) show that when \( k \) increases, the time for all the three algorithms will increase. The memory consumption for pRank-I and pRank-II will increase linearly while the the memory consumption for pRank remains the same. Comparing to the DBLP dataset, the time for pRank-I is more similar to pRank-II, because in the DBLP dataset, we can always find nodes with very large number of children in the projected PXML tree, for example the root node tagged “DBLP”, which will decrease the efficiency of pRank-I, whereas, in the Mondial dataset, the degree of each node is not large which makes the advantages of optimization-II less obvious.

### 5.3 Test-XMark

Fig. 16 shows the performance of the algorithms on the XMark datasets. Fig. 16(a) and Fig. 16(b) show that when the number of distribution nodes increases, the time and memory consumption for all the three algorithms will also increase. Fig. 16(c) and Fig. 16(d) show that when the number of result increases, the time for all the three algorithms will increase if the size of the projected
and Fig. 16(f) show the performance of the three algorithms when the size of the data increases. Otherwise, the time used for pRank-II is about 100 times faster than pRank, and the time used for pRank-I is about 30 times faster than pRank for all experiments on the XMark datasets. The memory consumption for pRank-I is similar to the memory cost for pRank-II, and is at most 5 times the memory consumption for pRank for all experiments on the XMark datasets.

6. RELATED WORK

**Ranking Results in XML:** XML twig queries have been extensively studied [4]. The result of a twig query over an XML tree is a set of answers. In [7, 19, 2], the authors integrate IR strategies to rank the results of twig queries. The scores of answers are computed using IR models, incorporating with other factors. In [6, 8], the authors retrieve the top-k results of a keyword search over XML tree. In [3, 15], the authors treat XML trees as XML graphs, and assign weights to the nodes and edges of XML graphs, where the weight of a node indicates its importance and the weight of an edge represents the strength of its semantic connection in the XML tree. The uncertainty and probability are not addressed.

**Probabilistic XML:** The topic of probabilistic XML (PXML) has been studied recently. Many models have been proposed, together with the complexity analysis of query evaluations. Nierman et al. [16] first introduce a simple probabilistic XML model, ProTDB, which is a probabilistic tree database. Hung et al. [11, 12] model the probabilistic XML as directed acyclic graphs, with probabilities defined on sets of children. Keulen et al. [22] use a probabilistic tree approach for data integration. Abiteboul et al. [1] propose a “fuzzy trees” model, where nodes are associated with conjunctions of probabilistic event variables, they also give a full complexity analysis of query and update on the “fuzzy tree” in [20]. Cohen et al. [5] incorporate a set of constraints to express more complex dependencies among the probabilistic data. They also propose efficient algorithms to solve the constraint-satisfaction, query evaluation, and sampling problem under a set of constraints. In [14], Kimelfeld et al. summarize and extend the probabilistic XML models previously proposed, the expressiveness and tractability of queries on different models are discussed. The ranking issues are not addressed in their work.

**Top-k Queries in Probabilistic Data:** Uncertain databases have received increasing attention recently. Apart from the works on different models of uncertain relational database, some recent works concern on answering top-k queries in uncertain database. There are two scenarios in ranking the query results, [18] or [21]. In [18], Re et al. find the k most probable answers for a given general SQL query. In this scenario, each answer has a probability instead of a score, which intuitively represents the confidence of its existence, ranking is only based on probabilities. They use Monte Carlo simulations to get the top-k results efficiently. Another definition is ranking the results by the interplay between score and uncertainty. In the setting of [21, 23, 9, 25, 24, 10], each result is a tuple, associated with both a score and a probability. U-TopK and U-kRanks queries are first proposed in [21], Yi et al. [23, 24] improve the performance of the two queries using a dynamic programming approach. Hua et al. [9, 10] define the PT-k query, and proposed three approaches to answer the PT-k query, which are, dynamic programming method, sampling method, and Poisson approximation based method. Zhang et al. propose a Global-Topk definition in [25]. Jin et al. [13] adapt the U-TopK/U-kRanks/PT-k/Global-TopK (Where Global-Topk is the same as Pk-topk in [13]) queries in a uncertain stream environment with sliding-window, and design both space- and time-efficient synopses to continuously monitor the top-k results. The works do not consider the containment issues and rank in XML trees.

![Figure 14: Testing DBLP dataset](image1)

![Figure 15: Testing Mondial dataset](image2)
In this paper, we studied ranking twig queries in probabilistic XML (PXML) data, called pXML-RANK query, \((Q, k)\), where \(Q\) is a twig query, which can be a node queries, path queries, and tree queries. We proposed dynamic programming algorithms with optimization techniques to efficiently rank answers of node queries, and showed that our techniques can be used to efficiently rank answers of path queries. For tree queries, we gave conditions under which we gave our solutions to rank answers of such tree queries without enumerating all the possible worlds. Our extensive experimental studies confirmed the efficiency of our approaches.

Acknowledgment
This work was supported by a grant of RGC (No. 418206), Hong Kong SAR, China.

8. REFERENCES

Figure 16: Testing XMark datasets
In our study, we consider all the three issues, namely, ranking, probability, and structures.

7. CONCLUSION
In this paper, we studied ranking twig queries in probabilistic XML (PXML) data, called pXML-RANK query, \((Q, k)\), where \(Q\) is a twig query, which can be node queries, path queries, and tree queries. We proposed dynamic programming algorithms with optimization techniques to efficiently rank answers of node queries, and showed that our techniques can be used to efficiently rank answers of path queries. For tree queries, we gave conditions under which we gave our solutions to rank answers of such tree queries without enumerating all the possible worlds. Our extensive experimental studies confirmed the efficiency of our approaches.

In this paper, we studied ranking twig queries in probabilistic XML (PXML) data, called pXML-RANK query, \((Q, k)\), where \(Q\) is a twig query, which can be node queries, path queries, and tree queries. We proposed dynamic programming algorithms with optimization techniques to efficiently rank answers of node queries, and showed that our techniques can be used to efficiently rank answers of path queries. For tree queries, we gave conditions under which we gave our solutions to rank answers of such tree queries without enumerating all the possible worlds. Our extensive experimental studies confirmed the efficiency of our approaches.

Acknowledgment
This work was supported by a grant of RGC (No. 418206), Hong Kong SAR, China.