Top-k Keyword Search over Probabilistic XML Data

Jianxin Li[1], Chengfei Liu[1], Rui Zhou[1], Wei Wang[2]

[1]*Swinburne University of Technology, Australia*
{jianxinli, cliu, rzhou}@swin.edu.au

[2]*University of New South Wales, Australia*
weiw@cse.unsw.edu.au

***Abstract*—Despite the proliferation of work on XML keyword query, it remains open to support keyword query over probabilistic XML data. Compared with traditional keyword search, it is far more expensive to answer a keyword query over probabilistic XML data due to the consideration of possible world semantics.**

In this paper, we firstly define the new problem of studying top-k keyword search over probabilistic XML data, which is to retrieve k SLCA results with the k highest probabilities of existence. And then we propose two efficient algorithms. The first algorithm PrStack can find k SLCA results with the k highest probabilities by scanning the relevant keyword nodes only once. To further improve the efficiency, we propose a second algorithm EagerTopK based on a set of pruning properties which can quickly prune unsatisfied SLCA candidates. Finally, we implement the two algorithms and compare their performance with analysis of extensive experimental results.

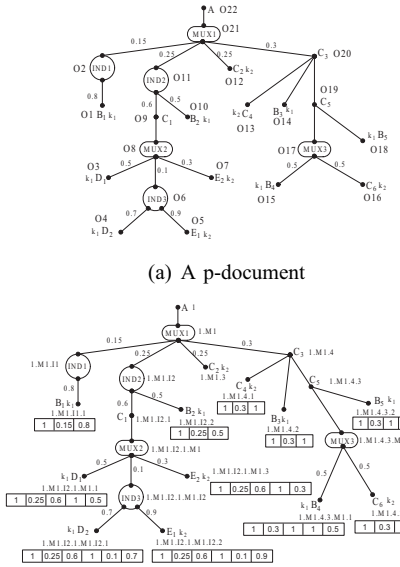
I. Introduction

Uncertainty is widespread in many web applications, such as information extraction, information integration, web data mining, etc. The flexibility of XML data model allows a natural representation of uncertain data. As a result, many probabilistic XML models are designed and analyzed [1], [2], [3], [4], [5], [6], [7]. In addition to the models themselves, uncertain data management is also becoming a critical issue in query evaluation [2], [6], [8], [9], [10], [11], algebraic manipulation [3] and updates [1], [6].

In this paper, we adopt a popular probabilistic XML model, PrXML$^{\{ind,mux\}}$ [9], which was first discussed in [2]. In this model, a probabilistic XML document (p-document) is considered as a labelled tree, consisting of two types of nodes, *ordinary* nodes representing the actual data and *distributional* nodes representing the probability distribution of the child nodes. Distributional nodes have two types, IND and MUX. An IND node has children that are *independent* of each other, while the children of a MUX node are *mutually-exclusive*, that is, at most one child can exist in a random instance document (called a *possible world*). A real number from (0,1] is attached on each edge in the XML tree, indicating the conditional probability that the child node will appear under the parent node given the existence of the parent node. An example of p-document is given in Fig. 1(a). Unweighted edges have 1 as the default conditional probability.

To query a p-document, existing works focused on using structured XML queries [8], [10], e.g., twig queries. It remains open whether keyword queries can be supported, whereas to provide keyword search support for users is appealing, because (1) it can relieve users from learning complex structured queries; (2) it does not require users to know the schema of a p-document. (A p-document may be integrated from multiple data sources, so it could be difficult for users to know its schema in advance).

A widely accepted semantics to model keyword search result on a deterministic XML tree T is Smallest Lowest Common Ancestor (SLCA) semantics [12], [13]. A node v is regarded as an SLCA if (a) the subtree rooted at the node v, denoted as $T_{sub}(v)$, contains all the keywords, and (b) there does not exist a descendant node v' of v such that $T_{sub}(v')$ contains all the keywords. Consider a keyword query $\{k_1, k_2\}$ on the p-document in Figure 1(a), the SLCAs are {IND3, MUX3}. However, for keyword search on a p-document, to naively return the SLCA nodes as the answers will bring in the following problems.



(a) A p-document

(b) An encoded p-document

Fig. 1. An example of probabilistic XML document

False Positive Problem Case 1: False SLCA node is introduced by MUX semantics, e.g., node $MUX3$ is an SLCA, for

B_4 contains k_1, C_6 contains k_2, but MUX3 is a false positive, because B_4 and C_6 cannot appear together under MUX3. Case 2: Distributional nodes are identified as SLCAs, e.g., SLCA nodes {IND3, MUX3} are all distributional nodes, but it is natural to constrain results to be ordinary nodes.

False Negative Problem Case 1: False negatives are caused by false positives, e.g., since {IND3, MUX3} are wrongly identified as SLCAs, more proper answers $\{C_1, C_5\}$ are ignored according to SLCA semantics. Case 2: All the ancestors of a traditional SLCA v_{slca} in a p-document may become SLCA answers, because v_{slca} may not appear in some possible worlds, which brings in the possibility that v_{slca}'s ancestors may be SLCA answers, e.g., C_3 is an SLCA in some possible worlds containing B_4 but not C_6, because C_5 is not an SLCA for those possible worlds.

From the above discussion, we can discover that the failure for enforcing traditional SLCA semantics on a p-document is due to the variety of possible worlds of the p-document. In other words, a node may be an SLCA in one possible world, but not in another. Consequently, an appropriate modification is to model keyword search result on a p-document by a set of 2-tuples. Each 2-tuple (v, f) is composed of v as a node in the p-document and f as the probability (confidence) for v to be an SLCA in the possible worlds. To be specific, if v appears as an SLCA in multiple possible worlds, the probability of those possible worlds will be aggregated to give the confidence for v to be an SLCA on the p-document. However, the answer set may be empty or too large if we do not set a proper probability threshold to f. Such a threshold is likely to be different for different datasets, and is hard to determine in advance. It is natural to give users k answers with the highest probabilities, where k can be specified by the users. To sum up, the problem we will study in this paper is, given a p-document, a set of keywords and a positive integer k, we want to find k nodes from the p-document with the highest probabilities to be SLCAs. Note that if there are only k' $(k' < k)$ SLCA nodes with non-zero probabilities, we only return these k' answers.

There are two major challenges: how to compute the probability for a node to be an SLCA and how to find out the SLCA nodes with top-k probabilities. A straightforward solution is to firstly generate the possible worlds and their corresponding probabilities from the given p-document, and then evaluate the keyword query on each possible world and combine the SLCA results on all the possible worlds, and at the same time probabilities are aggregated if necessary. Finally, to pick out k SLCA answers with the highest probabilities. Unfortunately, this method is time-consuming or even infeasible to process a p-document which has encoded a substantial number of possible world variations. More efficient and feasible solutions are sought-after.

To tackle the first challenge, we propose how to compute the SLCA probability for a node on a p-document without generating the possible worlds for the p-document. The computation follows a bottom-up manner. For each node v in the p-document, we compute a probability distribution table to record the probability for v to contain all or none or part of the query keywords. Such tables are easy to obtain for leaf nodes in the p-document. For an internal node v, the probability distribution table of v is computed by combining the probability distribution tables of v's children according to v's type. Different node types (IND, MUX, ordinary) are treated differently. Section III explains the details.

To target the second challenge, we propose two algorithms, PrStack and EagerTopK, to find the top-k probabilistic SLCA nodes without generating possible worlds. PrStack algorithm scans all the keyword match nodes once. It uses a stack to guarantee that the SLCA probability of a node is determined after all the SLCA probabilities of the node's child nodes have been determined. A result heap is used to collect top-k SLCA answers. While EagerTopK does not scan the keyword match nodes in document order. It first computes traditional SLCAs disregarding the node types of the SLCAs and their probabilities. Then it traces from these preliminary SLCA results upward towards the root of the document, picks out the ordinary nodes as SLCA answers and computes their probabilities. During the process, several probability upper bounds can be used to quickly discard candidate SLCA results without physically computing their probabilities, because their probability upper bounds are already less than the current kth highest probability. Section IV introduces the algorithms in detail.

We summarize the contributions of this paper as follows:

- To the best of our knowledge, this is the first work that studies keyword search over probabilistic XML data.
- We have discussed the principles to calculate the SLCA probability for an element when its sub-elements follow independent distribution or mutual-exclusive distribution. The calculation does not require the generation of possible worlds.
- We have proposed two algorithms, PrStack and EagerTopK, to compute top-k SLCA results with the highest probabilities without generating possible worlds. PrStack is easy to implement, while EagerTopK is relatively more complex, but terminates earlier in most cases, because EagerTopK uses tighter bounds to prune unpromising results.
- Experimental evaluation has demonstrated the efficiency of the proposed algorithms.

The rest of this paper is organized as follows. In Section II, we introduce the probabilistic XML model and the formal semantics of keyword search result on a probabilistic XML document. Section III first presents an encoding scheme with the consideration of distributional nodes, then discusses how to compute the SLCA probability in the presence of different distribution cases. In Section IV, we propose two algorithms to find top-k SLCA answers in terms of their probabilities. We report the experiment results in Section V. Section VI discusses related works and Section VII concludes the paper.

II. Problem Definitions

Probabilistic XML A probabilistic XML document (p-document) defines a probability distribution over a space of

deterministic XML documents. Each deterministic document belonging to this space is called a possible word. A p-document represented as a labelled tree has *ordinary* and *distributional* nodes. Ordinary nodes are regular XML nodes and they may appear in deterministic documents, while distributional nodes are only used for defining the probabilistic process of generating deterministic documents and they do not occur in those documents. As we adopt PrXML$^{\{ind,mux\}}$ as the probabilistic XML model, two types of distributional nodes, *IND* and *MUX*, may appear in a p-document.

Example 1: Consider the p-document T shown in Figure 1(a). Ordinary nodes are shown with their tag names, e.g., B_1, B_2, C_1, and C_2. For the two types of distributional nodes, *MUX* is depicted as rectangular boxes with rounded corners and *IND* is displayed as circles. Consider the *IND2* node. It has two children C_1 and B_2 with the existence probabilities 0.6 and 0.5, respectively. Thus, the absence probability for neither C_1 nor B_2 appearing is (1-0.6)*(1-0.5)=0.2. Consider the *MUX2* node that has three children, namely, D_1, E_2 and the *IND3* node. Their existence probabilities are 0.5, 0.3 and 0.1, respectively. Hence, the probability for none of them appearing is 1 - 0.5 - 0.3 - 0.1 = 0.1.

Given a p-document tree T, we can generate all possible worlds or deterministic documents as follows. Basically we traverse T in a top-down fashion. When we visit a distributional node, two situations need to be dealt with differently. (1) If it is an *IND* node with m child nodes, we generate 2^m copies of T, delete the *IND* node, replace the m child nodes with one distinct subset of them for a copy, and connect each child node in the subset to the ordinary parent node of the *IND* node. For each copy, the probability for this copy occurring is the product of all existence probabilities of the child nodes in the subset and the absence probabilities (i.e., one minus the existence probability) of the child nodes not in the subset. (2) If it is a *MUX* node with m child nodes, we generate $m + 1$ copies of T, delete the *MUX* node, replace the m child nodes with one distinct child node or no child for a copy, and for the former, connect the child node to the ordinary parent node of the *MUX* node. For each copy, the probability for this copy occurring is the existence probability of the distinct child node in the subset or the absence probability for none of the child nodes appearing. For each generated copy of T, we continue the top-down traversal until all distributional nodes are deleted. At the end, each distinct copy becomes a possible world of T. In the case that two generated copies are the same, they are merged with the probability as the sum of the individual probabilities.

Example 2: For the p-document subtree rooted at C_1 in Figure 1(a), all its possible worlds can be found in Figure 2 by running the above procedure. When *MUX2* is visited, four documents are generated and the one with *IND3* in turn generates 4 more documents, resulting in 7 documents generated in total. Among them, there are two copies containing a single node C_1: one when *MUX2* is visited with the probability 1-0.5-0.1-0.3 = 0.1, and the other when *IND3* is visited with the probability 0.1*(1-0.7)*(1-0.9) = 0.003. So the sum of them is 0.103. The probabilities of the other possible worlds are easy to calculate following the above procedure.

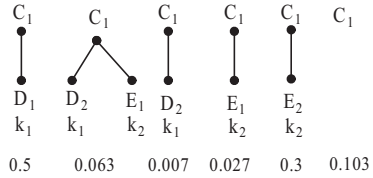


Fig. 2. Example of deterministic trees rooted at C_1

Top-k Keyword Query A top-k keyword query consists of a set of keywords $\{k_1, k_2, \ldots, k_n\}$ and a positive integer k. We define the answers for a top-k keyword query on a p-document T as k ordinary nodes on T with the highest probabilities to be SLCAs in the possible worlds generated by T. The probability of a node v being an SLCA in the possible worlds is denoted as $Pr^G_{slca}(v)$. The formal definition of $Pr^G_{slca}(v)$ is as follows:

$$Pr^G_{slca}(v) = \sum_{i=1}^{m} \{Pr(w_i) | slca(v, w_i) = true\} \quad (1)$$

where $w_1, \ldots, w_m$ denotes the possible worlds implied by T. $slca(v, w_i) = true$ indicates that v is an SLCA in the possible world w_i. $Pr(w_i)$ is the existence probability of the possible world w_i.

To develop the above discussion, $Pr^G_{slca}(v)$ can also be computed with Equation 2. Here, $Pr(path_{r \to v})$ indicates the existence probability of v in the possible worlds. It can be computed by multiplying the conditional probabilities in T, along the path from the root r to v. $Pr^L_{slca}(v)$ is the local probability for v being an SLCA in $T_{sub}(v)$, where $T_{sub}(v)$ denotes a subtree of T rooted at v.

$$Pr^G_{slca}(v) = Pr(path_{r \to v}) \times Pr^L_{slca}(v) \quad (2)$$

To compute $Pr^L_{slca}(v)$, we have the following equation similar to Equation 1.

$$Pr^L_{slca}(v) = \sum_{i=1}^{m'} \{Pr(t_i) | slca(v, t_i) = true\} \quad (3)$$

where m' deterministic trees $\{t_1, t_2, ..., t_{m'}\}$ are local possible worlds generated from $T_{sub}(v)$, $Pr(t_i)$ $(1 \leq i \leq m')$ is the probability of generating t_i from $T_{sub}(v)$; $slca(v, t_i) = true$ means v is an SLCA node in t_i, namely the root node v is the only LCA node in t_i.

Example 3: We give an example to compute $Pr^G_{slca}(C_1)$. It is not difficult to see that $Pr(path_{A \to C_1}) = 1 * 0.25 * 0.6 = 0.15$. To compute $Pr^L_{slca}(v)$, after examining the seven local possible worlds for $T_{sub}(C_1)$ in Figure 2, C_1 is an SLCA for only one local possible world with probability 0.63, so we have $Pr^L_{slca}(v) = 0.63$. As a result, $Pr^G_{slca}(C_1) = Pr(path_{A \to C_1}) \times Pr^L_{slca}(C_1) = 0.15 * 0.63 = 0.0945$.

Although Equation 2 and 3 have simplified the computation, a straightforward implementation cannot avoid generating possible worlds. Since it is already tedious for enumerating

possible worlds for a small probabilistic subtree $T_{sub}(C_1)$, one can imagine the workload for larger subtrees. However, observing that for an ancestor-descendant node pair (v_a, v_d), local possible worlds of $T_{sub}(v_d)$ always constitute local possible worlds of $T_{sub}(v_a)$, it may be appropriate to compute $Pr^G_{slca}(v_d)$ first, and reuse part of the computation done for v_d when we compute $Pr^G_{slca}(v_a)$. This point motivates our SLCA-probability computation method and algorithms. They are conceptually in a bottom-up manner.

III. Overview of this work

In this section, we briefly introduce an encoding scheme for a probabilistic XML data tree and then go through the method of computing probability of SLCA results without generating possible world.

A. Encoding Probabilistic XML Data

Dewey encoding scheme has been verified and applied as an effective and efficient way in previous keyword search approaches [12], [14]. Given two keyword nodes, it is easy to get the common ancestor by comparing their Dewey codes. In this work, we extend the Dewey encoding scheme for adapting to probabilistic XML documents. To do this, we encode each node into a number that represents its Dewey number while a char "M" or "I" is attached before the number for MUX or IND nodes. Figure 1(b) demonstrates the results of encoding the nodes in Figure 1(a). In addition, each leaf node maintains a list of conditional probabilities from the root to the leaf node. The recorded conditional probabilities can be used to calculate the probability of SLCA results. The detailed encoding algorithm is omitted due to the limited space.

B. Directly Computing Probability of SLCA Result

In this section, we illustrate the procedure of directly computing probability of SLCA result from a given set of keyword nodes, rather than from the possible worlds.

Consider a keyword query Q = $\{k_1, ..., k_n\}$ and any parent node p with its child nodes c_1, ..., c_m. For each c_i, λ_i represents the conditional probability of c_i when p appears, while tab_i is a table with the maximal size $2^{|Q|}$ maintaining the keyword distributions $\{\mu_{(2^{|Q|}-1)}, ..., \mu_0\}$ in node c_i. Our task is to calculate the distribution table tab_p of node p based on tab_i and λ_i where $1 \leq i \leq m$. To make the calculation easy, we utilize binary number to represent keyword distributions in each table. For example, given a keyword query $\{k_1,k_2\}$, distribution '11' - 'μ_3' means the probability that p is an SLCA that contains both k_1 and k_2; '10' - 'μ_2' means the probability that p contains k_1, but not k_2; '01' - 'μ_1' means the probability that p contains k_2, but not k_1, and '00' - 'μ_0' means the probability that p contains neither k_1 nor k_2.

Due to the features of probabilistic documents, the parent node p may be an ordinary node, an IND node, or a MUX node. For different node types, we have to adopt different methods to compute the distributions of keywords in p using its child nodes' distributions.

*1) **Node p is an IND node**:* For each child node c_i, we first promote its distribution tab_i=$\{(2^{|Q|}-1) \rightarrow \mu_{(2^{|Q|}-1)}, ..., 0 \rightarrow \mu_0\}$ by multiplying with its local conditional probability λ_i. As such, the updated distribution is refreshed as $tab_i^{(\uparrow)}$=$\{(2^{|Q|}-1) \rightarrow \lambda_i * \mu_{(2^{|Q|}-1)}, ..., 0 \rightarrow (\lambda_i * \mu_0 + 1 - \lambda_i)\}$. The general equation is displayed as follows.

$$tab_i^{(\uparrow)} = \begin{cases} \{x \rightarrow \lambda_i * \mu_x | \forall x \in tab_i, x \neq 0\} \\ \{x \rightarrow (\lambda_i * \mu_x + 1 - \lambda_i) | x = 0\} \end{cases} \quad (4)$$

And then we merge $tab_i^{(\uparrow)}$ with the distribution tab_p=$\{(2^{|Q|}-1)^p \rightarrow \mu^p_{(2^{|Q|}-1)}, ..., 0^p \rightarrow \mu^p_0\}$ of p using a set of *bitwise OR* operations bounded by the maximal size $(2^{|Q|}-1)^2$. Generally, keyword query size is around 3 or 4. In addition, we only cache the none-zero values in the distributional table. As such, lots of operations can be avoided. To guarantee the completeness of the results, we execute the bitwise operations for $tab_i^{(\uparrow)}$ and tab_p. The corresponding values will be multiplied and assigned to the corresponding keyword distributions. At last, the computed results will be aggregated together as the updated $tab_{p'}$.

$$tab_{p'} = \{z \rightarrow \sum \mu_x * \mu_y | z = x \text{ OR } y, \forall x \in tab_i^{(\uparrow)}, \forall y \in tab_p\} \quad (5)$$

If tab_p does not exist at the beginning, i.e., all the values are zero, we directly assign $tab_i^{(\uparrow)}$ to tab_p without any computation. The final distribution of the IND node can be obtained by processing all its relevant child nodes in the similar way.

Example 4: Let's consider the IND node 1.M1.I2.1.M1.I2 and its two children D_2 and E_1 w.r.t. a keyword query $\{k_1, k_2\}$. The distribution of keywords in D_2 is:

$D_2.tab$			
11	10	01	00
0	1	0	0

As we know the local conditional probability of D_2 is 0.7, the promoted distribution of D_2 can be updated as:

$D_2.tab^{(\uparrow)}$			
11	10	01	00
0	0.7	0	0.3

, which becomes the start distribution tab_p of the IND node. Here, $D_2.tab^{(\uparrow)}$: 00 is set as 0.3, not 0 according to Equation 4, which also conforms to the IND semantics. Similarly, we can get the promoted distribution of E_1 as:

$E_1.tab^{(\uparrow)}$			
11	10	01	00
0	0	0.9	0.1

. Because the distribution tab_p contains non-zero values now, we should merge $E_1.tab^{(\uparrow)}$ into tab_p by a set of bitwise operations:

$$\begin{array}{l} 01 \text{ OR } 10^p = 11^{p'} \rightarrow (E_1.tab^{(\uparrow)} : 01) * (tab_p : 10) = 0.9 * 0.7 = 0.63; \\ 01 \text{ OR } 00^p = 01^{p'} \rightarrow (E_1.tab^{(\uparrow)} : 01) * (tab_p : 00) = 0.9 * 0.3 = 0.27; \\ 00 \text{ OR } 10^p = 10^{p'} \rightarrow (E_1.tab^{(\uparrow)} : 00) * (tab_p : 10) = 0.1 * 0.7 = 0.07; \\ 00 \text{ OR } 00^p = 00^{p'} \rightarrow (E_1.tab^{(\uparrow)} : 00) * (tab_p : 00) = 0.1 * 0.3 = 0.03. \end{array}$$

As such, the tab_p will be updated as:

$tab_{p'}$			
11	10	01	00
0.63	0.07	0.27	0.03

. The computing procedure still conforms to the possible world semantics because the union of the current distributions in $tab_{p'}$ is still equal to one.

*2) **Node p is a MUX node**:* For each child node c_i $(1 \leq i \leq m)$, we first promote its distribution tab_i=$\{(2^{|Q|}-1) \rightarrow \mu_{(2^{|Q|}-1)}, ..., 0 \rightarrow \mu_0\}$ by multiplying with its local

conditional probability λ_i. As such, the updated distribution is refreshed as $tab_i^{(\uparrow)}$=$\{(2^{|Q|}-1) \rightarrow \lambda_i * \mu_{(2^{|Q|}-1)}$, ..., $0 \rightarrow \lambda_i * \mu_0\}$. The general equation is provided as follows.

$$tab_i^{(\uparrow)} = \{x \rightarrow \lambda_i * \mu_x | \forall x \in tab_i\} \quad (6)$$

Next, we can directly add the keyword distribution values in $tab_i^{(\uparrow)}$ into the corresponding parts in tab_p based on the understanding of MUX semantics, i.e., child nodes of a MUX node cannot co-occur at the same time. Therefore, $tab_{p'}$ can be obtained by aggregating from the corresponding parts of tab_p, and $tab_i^{(\uparrow)}$ $(1 \leq i \leq m)$ with the following equation. As an acute reader, you may find that the computation of $tab_{p'} : 0 \rightarrow \mu_0$ is a special case. It is equal to the aggregation of all child nodes' $tab^{(\uparrow)} : 0 \rightarrow \mu_0$ and an extra part $1 - \sum_{i=1}^{n} \lambda_i$. The value can be obtained with the promotion of each related child node, which does not require extra cost.

$$tab_{p'} = \{z \rightarrow \mu_z + \mu_x | z = x, \forall x \in tab_i^{(\uparrow)}, \forall z \in tab_p\}; \quad (7)$$

After all $tab_i^{(\uparrow)}$ are processed,

$$tab_{p'} : 0 \Leftarrow tab_{p'} : 0 + (1 - \sum_{i=1}^{n} \lambda_i). \quad (8)$$

Example 5: Let's consider a MUX node (1.M1.I2.1.M1) and its child nodes D_1, an IND node and E_2 w.r.t. a keyword query $\{k_1, k_2\}$. At the beginning, we assume that their distributions are listed as follows:

$D_1.tab$			
11	10	01	00
0	1	0	0

$IND.tab$			
11	10	01	00
0.63	0.07	0.27	0.03

$E_2.tab$			
11	10	01	00
0	0	1	0

. Then we promote them to their parent node by considering their local conditional probabilities: 0.5, 0.1 and 0.3, respectively. Their distributions are updated as follows:

$D_1.tab^{(\uparrow)}$			
11	10	01	00
0	0.5	0	0

$IND.tab^{(\uparrow)}$			
11	10	01	00
0.063	0.007	0.027	0.003

$E_2.tab^{(\uparrow)}$			
11	10	01	00
0	0	0.3	0

. Based on the updated distributions of child nodes, the distribution of their parent node can be calculated as follows.

$11^{p'} \rightarrow (D_1.tab^{(\uparrow)} : 11 + IND.tab^{(\uparrow)} : 11 + E_2.tab^{(\uparrow)} : 11) = 0.063$;
$10^{p'} \rightarrow (D_1.tab^{(\uparrow)} : 10 + IND.tab^{(\uparrow)} : 10 + E_2.tab^{(\uparrow)} : 10) = 0.507$;
$01^{p'} \rightarrow (D_1.tab^{(\uparrow)} : 01 + IND.tab^{(\uparrow)} : 01 + E_2.tab^{(\uparrow)} : 01) = 0.327$;
$00^{p'} \rightarrow (IND.tab^{(\uparrow)} : 00 + 1 - 0.5 - 0.1 - 0.3 = 0.103)$.

As such, the tab_p will be updated as:

$MUX2.tab^{p'}$			
11	10	01	00
0.063	0.507	0.327	0.103

. The computing procedure also holds the possible world semantics because the union of the current distributions in $tab_{p'}$ is still equal to one.

*3) **Node p is an ordinary node**:* The procedure is similar to the IND case. The computational equation is the same as Equation 5. However, the difference is that we may generate an SLCA candidate with its probability when the value of $tab_{p'} : (2^{|Q|}-1)$ is not equal to zero. As such, we remove $(2^{|Q|}-1)$ from $tab_{p'}$ because the probability of an SLCA node can not contribute to its ancestors again due to the SLCA semantics.

Example 6: Consider C_1 and its child MUX2, we can directly take the keyword distribution MUX2.$tab^{p'}$ of MUX2 as that of C_1 because MUX2 is the only child of C_1 and the edge between them is an ordinary edge. Since C_1 is an ordinary node, we begin to check if it is an SLCA candidate. Because of $C_1.tab : 11 \rightarrow 0.063$, we calculate the probability of C_1 becoming an SLCA result as Pr(path$_{C_1}$)*$C_1.tab : 11$ = (1*0.25*0.6)*0.063=0.00945. After that, $C_1.tab$ will be updated by assigning zero to $C_1.tab : 11$, i.e.,

$C_1.tab$			
11	10	01	00
0.063	0.507	0.327	0.103

$\Rightarrow$

$C_1.tab'$			
11	10	01	00
0	0.507	0.327	0.103

.

Consider C_3 and its child nodes C_4, B_3 and C_5. We first get the keyword distributions from C_4 and B_3:

$C_4.tab$			
11	10	01	00
0	0	1	0

$B_3.tab$			
11	10	01	00
0	1	0	0

. After their distributions are promoted and merged, we get the distributions of their parent node C_3:

$C_3.tab$			
11	10	01	00
1	0	0	0

. Although $C_3.tab : 11$ has already been increased to 1, but we have to consider the remaining child node C_5 which may produce SLCA results and reduce the value of $C_3.tab : 11$. In this example, the probability for C_5 becoming a new SLCA result is 0.5, which comes from B_5 and C_5, where

$C_5.tab$			
11	10	01	00
0.5	0.5	0	0

. Since C_5 is also an ordinary node and contains the full keyword distribution, its global probability can be computed as Pr(path$_{C_5}$)*($C_5.tab : 11$) = 1*0.3*1*0.5 = 0.15. After that, we promote $C_5.tab : 10 \rightarrow 0.5$ to $C_3.tab$. As such, the probability for C_3 becoming an SLCA candidate can be computed as Pr(path$_{C_3}$)*($C_3.tab : 11$*$C_5.tab : 10$) = 1*0.3*1*0.5=0.15. After outputting the SLCA candidate, we exclude the value of $C_3.tab : 11$ by setting it as zero, i.e.,

$C_3.tab'$			
11	10	01	00
0	0	0	0

.

To make the above examples clear, we allow the distribution tables contain zero values. In fact, we only need to maintain the none-zero values in our implementation. We can get the probability of each SLCA result as Example 3 by repeatedly calling the above three types of operations.

IV. Top-k Keyword Search Algorithms

Given a top-k keyword query $\{k_1, k_2, ..., k_n\}$ and a probabilistic XML data tree, our target is to quickly find the k best SLCA results with the highest probabilities. To do this, two algorithm are introduced, PrStack algorithm is realized by scanning the keyword nodes once in a document order while EagerTopK algorithm is designed based on a series of upper bound properties.

A. PrStack Algorithm

In this section, we introduce a stack-based algorithm, PrStack, to discover the SLCA nodes with the k higest probabilities in a probabilistic XML tree.

1) The idea of PrStack Algorithm: PrStack scans all the keyword inverted lists once. It progressively reads keyword match nodes one by one according to their document order from the inverted lists, and compute the probabilities for the SLCA nodes that are ancestors of the keyword match nodes.

The principle is that, when computing the probability of an SLCA node v, it has retrieved all the keyword matches that are descendants of v from all the keyword lists. In other words, the SLCA probability of a node v is determined after the probabilities of all child SLCA nodes of v have been determined. The newly computed probability, together with the corresponding SLCA node, will be inserted into a k-size heap if the new probability is larger than the current kth highest probability.

Figure 1(a) shows the order of SLCA probability computation for some of the nodes. The order is indicated by $''O*''$. It is not difficult to see that the probability-computation order equals to the order for visiting nodes in a postorder traversal of the tree. This probability-computation order can be realized by maintaining a stack. The probability of each SLCA node can be computed according to the formula in Section III.

TABLE I
HASH MAP $\varphi : v \rightarrow dist$

node	$dist$	node	$dist$
$1.M1.I1.1$	$\{01 \rightarrow 1\}$	$1.M1.I2.1.M1.1$	$\{01 \rightarrow 1\}$
$1.M1.I2.1.M1.I2.1$	$\{01 \rightarrow 1\}$	$1.M1.I2.1.M1.I2.2$	$\{10 \rightarrow 1\}$
$1.M1.I2.1.M1.3$	$\{10 \rightarrow 1\}$	$1.M1.I2.2$	$\{01 \rightarrow 1\}$
$1.M1.4.1$	$\{10 \rightarrow 1\}$	$1.M1.4.2$	$\{01 \rightarrow 1\}$
$1.M1.4.3.M1.1$	$\{01 \rightarrow 1\}$	$1.M1.4.3.M1.2$	$\{10 \rightarrow 1\}$
$1.M1.4.3.2$	$\{01 \rightarrow 1\}$		

2) Index Structure: Before explaining the algorithm, we first introduce two indexes that will be used in the following sections. The first index is used to maintain the relationship between a node v and its local probabilistic keyword distributions $dist$ w.r.t. a given keyword query Q. It is denoted as $\varphi : v \rightarrow dist$. φ is initiated and updated during the time that we load keyword nodes from database. The maximal size of φ is equal to the total number of keyword nodes. During query evaluation, once a node is probed, its distributions can be removed from φ and promoted to its parent based on the formula in Section III. Table I displays the status after we load all the keyword nodes w.r.t. $Q = \{k_1, k_2\}$. And we use $dist$ to record the mapping from keyword distributions to their probabilities. The maximal size of $dist$ is equal to $2^{|Q|}$. For instance, $10 \rightarrow 1$ represents the local probability of a node only containing k_2 is 1 w.r.t. the query Q. The second index keeps the mapping between a node and its all conditional probabilities. It is denoted as $\gamma : v \rightarrow PrLink$. For example, the conditional probabilities of D_1 is a link, | 1 | 0.25 | 0.6 | 1 | 0.5 |.

3) Description of PrStack Algorithm: The detailed procedure of PrStack algorithm is shown in Algorithm 1. We first load the relevant keyword node lists L and their conditional probabilities γ. At the same time, the map between nodes and keyword distributions can be built based on the loaded keyword nodes. Then we get the first node v that has the smallest Dewey code from L and initiate a $stack$ based on the Dewey components of v. To simplify the explanations, v is also used to represent its Dewey code.

After that, we process the rest of keyword nodes in L one by one in a document order until the end of L. Line 4-Line 17 shows the detailed procedure. It firstly get the node

Algorithm 1 PrStack Algorithm

input: a keyword query $\{k_1, ..., k_n\}$ and an encoded probabilistic XML data tree T
output: top k ranked SLCA results R

1: load node lists $L = \{L_i\}$, $1 \leq i \leq n$, $\gamma : v \rightarrow PrLink$, create and update $\varphi : v \rightarrow dist$;
2: get the smallest Dewey v from L;
3: initiate a $stack$ using the Dewey v;
4: **while** NextNode(L) $\neq$ null **do**
5: v' = GetNextNode(L);
6: **if** v' is not promoted to its parent $p(v)$ **then**
7: $p = lcp(stack, v')$; {find the longest common prefix p such that $stack[i] = v[i]$, $0 \leq i \leq p.length - 1$}
8: **while** $stack.size > p.length$ **do**
9: $v = stack$.pop();
10: get type(v) and local conditional probability lcprob(v);
11: **if** v is an ordinary node **then**
12: GenerateResults(v);
13: **if** $p(v) \notin \varphi$ **then**
14: DirectPromotion(v, type(v), lcprob(v), p(v));
15: **else**
16: CombineProb(v, type(v), lcprob(v), p(v));
17: $stack$.push($v'[j]$) for $p.length \leq j < v'.length$;
18: use similar procedure from Line 9 to Line 16 to process the rest of nodes in $stack$ until it becomes empty;
19: **return** R;

Function GenerateResult

1: get distribution $dist$ of v from φ; {v is a real SLCA if $dist$ contains $\overbrace{1...1}^{n}$}
2: **if** v is a real SLCA **then**
3: get the local probability Pr_v^L from $dist$;
4: get the multiplied conditional probabilities Pr(path$_v$) from $\gamma(v)$;
5: pr_v^G = Pr_v^L * Pr(path$_v$);
6: **if** The number of current results is less than k **then**
7: put $v \rightarrow \text{pr}_v^G$ into R;
8: **else if** pr_v^G is larger than k-th highest value in R **then**
9: remove the k-th candidate and put $v \rightarrow \text{pr}_v^G$ into R;
10: remove Pr_v^L from $dist$;
11: write v and the updated $dist$ back to φ;

v' with the next smallest Dewey code by calling Function GetNextNode(L) that records a cursor for accessing the next node of L. Next, we need to compute the longest common prefix p between v' and v (in $stack$). If $stack.size$ is larger than the length of p, which means that there is no ancestor-descendant relationship between v' and v, then we get the Dewey code of v by accessing the code components in stack from bottom to top and pop out v from the stack. We then identify if v can become an SLCA result, as shown in Line 10-Line12. If v is an ordinary node, we call Function GenerateResults() in Algorithm 1 to generate a new SLCA result if it exists. If the type of v is IND or MUX, we only pop the stack and promote the distributions of v to its parent $p(v)$.

To correctly make the promotion, there are two different cases in Line 13-Line 16: (1) $p(v)$ does not exist in φ, we can directly promote the $dist$ of v to $p(v)$ by calling Function DirectPromotion(v, $type(v)$, $lcprob(v)$, $p(v)$). To make direct

promotion, we only need to multiply the values in $dist$ ($\varphi(v)$) of v by its local conditional probability $lcprob(v)$ as Equation 4 and Equation 6. The specific consideration is given to the "0" distribution by adding the extra part $(1-lcprob(v))$ as Equation 4 for IND or Ordinary nodes and Equation 8 for MUX nodes. (2) $p(v)$ exists in φ, we have to merge the $dist$ coming from v with the existing $dist'$ of $p(v)$ together by using Function CombineProb(v, $type(v)$, $lcprob(v)$, $p(v)$), which implements the procedure of using Equation 5 and Equation 7. The specific consideration of this procedure is also focused on the computation of "0" distribution when v is a MUX node. We need to subtract $lcprob(v)$ from the "0" distribution value.

After we process all the popped nodes from $stack$ until $stack.size$ is equal to the length of p, we push the unmatched components of v' into $stack$. The above procedure will be repeatedly processed until no keyword nodes are left. At last, we pop out the rest of nodes in $stack$ and process them using the similar procedure until $stack$ becomes empty, as shown in Line 9 to Line 16 in Algorithm 1.

B. EagerTopK Algorithm

In this section, we propose an eager algorithm, EagerTopK, for catering for top-k keyword search. It starts from the set of initial SLCAs and then select the promising candidates to evaluate. EagerTopK can quickly find top-k answers by pruning the unsatisfied candidates.

1) Pruning Properties: Before introducing the algorithm, we introduce some upper bound properties that can be used to prune unpromising nodes without physically computing the SLCA probability of those nodes. Assume we want to compute the probability for an SLCA node v, let the set of node on the path from the root r to node v (including v itself) be $V_{r\to v}$ and let $v_{c_1}, \ldots, v_{c_m}$ be the children of v on the p-document T, let $Pr^G_{all}(v_{c_i})$ denote the probability that node v_{c_i} covers all the keywords in the subtree rooted at v in terms of the possible world semantics. We have the following properties:

Property 1: If node v is an IND or ordinary node, we have the following equation:

$$\sum_{v_i \in V_{r\to v}} Pr^G_{slca}(v_i) \le \prod_{i=1}^{m} (1 - Pr^G_{all}(v_{c_i}))$$

Explanation: The reason is that an SLCA exists on the path $r \to v$ implies there is no SLCA in any of the subtree rooted at v_{c_i} $(i \in [1, m])$. Consequently, $\prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i}))$ not only gives an upper bound for $Pr^G_{slca}(v)$, it is also an upper bound of $Pr^G_{slca}(v')$ for any node v' on the path $r \to v$. If $\prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i}))$ is smaller than the current kth highest SLCA probability, we can safely prune all the nodes on the path $r \to v$.

Property 2: If node v is a MUX node, we have the following equation:

$$\sum_{v_i \in V_{r\to v}} Pr^G_{slca}(v_i) \le 1 - \sum_{i=1}^{m} Pr^G_{all}(v_{c_i})$$

Explanation: in fact, $Pr^G_{slca}(v)$ should be 0, because a MUX cannot be an answer. But we still calculate $1 - \sum_{i=1}^{m}(Pr^G_{all}(v_{c_i}))$, because it gives an upper bound for all v's ancestors to be an SLCA like Property 1.

Here, $Pr^G_{all}()$ can also be calculated in a bottom up manner. For an IND or ordinary node,

$$Pr^G_{all}(v) = 1 - \prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i})) + Pr^G_{slca}(v) \quad (9)$$

For a MUX node,

$$Pr^G_{all}(v) = \sum_{i=1}^{m} Pr^G_{all}(v_{c_i}) \quad (10)$$

In the above analysis, given a node v, we assume that all v's children have been discovered when we compute the upper bound for v. However, the prerequisite can be relaxed. For example, let $v_{d_1}, \ldots, v_{d_m}$ be a set of descendants of v which do not have ancestor-descendant relationship, suppose we have known $Pr^G_{all}(v_{d_i})$ for each d_i. We have Property 3.

Property 3: No matter node v is an IND or ordinary or MUX node, we have the following equation:

$$\sum_{v_i \in V_{r\to v}} Pr^G_{slca}(v_i) \le \prod_{i=1}^{m}(1 - Pr^G_{all}(v_{d_i}))$$

Explanation: For an IND or ordinary node v, according to Equation 9, we have $1 - Pr^G_{all}(v) = \prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i})) - Pr^G_{slca}(v) \le \prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i}))$. Similarly, for a MUX node, according to Equation 10, we have $1 - Pr^G_{all}(v) = 1 - \sum_{i=1}^{m} Pr^G_{all}(v_{c_i}) \le \prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i}))$. For both cases, we have:

$$1 - Pr^G_{all}(v) \le \prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i})) \quad (11)$$

As a result, we have the Equation 12. The last step is due to a recursive induction using Equation 11:

$$\begin{aligned} \textstyle\sum_{v_i \in V_{r\to v}} Pr^G_{slca}(v_i) &\le 1 - \textstyle\sum_{i=1}^{m} Pr^G_{all}(v_{c_i}) \\ &\le \textstyle\prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i})) \\ &\le \textstyle\prod_{i=1}^{m}(1 - Pr^G_{all}(v_{d_i})) \end{aligned} \quad (12)$$

Property 3 can be used to improve the upper bound progressively. To give an example, at the beginning, the upper bound for a node v is 1. After obtaining $Pr^G_{all}(v_{d_1})$ where v_{d_1} is an descendant of v, we can update the upper bound for v by $1 - Pr^G_{all}(v_{d_1})$. To follow the process, when we have obtained $Pr^G_{all}(v_{d_2})$, where v_{d_2} is a sibling of v_{d_1} (v_{d_2} is also a descendant of v and shares the same parent with v_1), we can update the upper bound for v as $(1 - Pr^G_{all}(v_{d_1})) * (1 - Pr^G_{all}(v_{d_2}))$. One tricky step is, when we have obtained $Pr^G_{all}(v_p)$, here v_p is the parent of v_{d_1} and v_{d_2}, the upper bound of v, which is currently $(1 - Pr^G_{all}(v_{d_1})) * (1 - Pr^G_{all}(v_{d_2}))$, can be improved as $1 - Pr^G_{all}(v_p)$ by multiplying $\frac{1 - Pr^G_{all}(v_p)}{(1 - Pr^G_{all}(v_{d_1})) * (1 - Pr^G_{all}(v_{d_2}))}$, since $1 - Pr^G_{all}(v_p) = (1 - Pr^G_{all}(v_{d_1})) * (1 - Pr^G_{all}(v_{d_2})) - Pr^G_{slca}(v_p) \le (1 - Pr^G_{all}(v_{d_1})) * (1 - Pr^G_{all}(v_{d_2}))$. The tricky step implies that whenever we go upward and gain some new information, such as $Pr^G_{slca}(v_p)$, the upper bound of higher

nodes may be reduced further. Consequently, the upper bound is decreasing as we continuously probe the SLCA candidates in a bottom-up manner.

Property 4: If node v is an IND or ordinary node, then its SLCA probability satisfies:

$$Pr^G_{slca}(v) \leq Pr(path_{r\rightarrow v}) * \prod_{i=1}^{m}(1 - \frac{Pr^G_{all}(v_{c_i})}{Pr(path_{r\rightarrow v})})$$

Explanation: here, $Pr(path_{r\rightarrow v})$ is the probability for v to occur in the possible worlds. In the local possible worlds generated by $T_{sub}(v)$, $1 - \frac{Pr^G_{all}(v_{c_i})}{Pr(path_{r\rightarrow v})}$ gives the probability that c_i does not contain all the keywords given the existence of v. Following an independent distribution, $\prod_{i=1}^{m}(1 - \frac{Pr^G_{all}(v_{c_i})}{Pr(path_{r\rightarrow v})})$ gives an upper bound for the local SLCA probability $Pr^L_{slca}(v)$. And thus, $Pr(path_{r\rightarrow v}) * \prod_{i=1}^{m}(1 - \frac{Pr^G_{all}(v_{c_i})}{Pr(path_{r\rightarrow v})})$ is an upper bound for the global probability $Pr^G_{slca}(v)$. Compared with Property 1, Property 4 gives an upper bound for a single node. Similarly, we have Property 5.

Property 5: If node v is a MUX node, then its SLCA probability satisfies:

$$Pr^G_{slca}(v) \leq Pr(path_{r\rightarrow v}) * (1 - \sum_{i=1}^{m}\frac{Pr^G_{all}(v_{c_i})}{Pr(path_{r\rightarrow v})})$$

To sum up, based on Property 1 and Property 2, we can prune the nodes that are the ancestors of a node v if $\prod_{i=1}^{m}(1 - Pr^G_{all}(v_{c_i}))$ (v is IND or ordinary) or $1 - \sum_{i=1}^{m} Pr^G_{all}(v_{c_i})$ (v is MUX) is smaller than the current kth highest SLCA probability. The most often used property is Property 3. The bound is looser than those of Property 1 and 2, but still has good pruning power, because we can decrease the upper bound for a node using its descendant nodes without necessarily knowing all its children. v's upper bound can be decreased earlier. Moreover, based on Property 4 and Property 5, we can postpone the calculation of a node v until one of its ancestors cannot be pruned by the upper bounds.

2) The idea of EagerTopK Algorithm: We first scan the keyword node lists to compute and get SLCA candidate set S using any traditional SLCA algorithm, in which distributional nodes are treated as ordinary nodes. As a result, S may contain false positive SLCA candidates. And then, we compute the probability $Pr^G(v)$ for each SLCA candidate v in S. To calculate $Pr^G(v)$, we need to access and remove the relevant keyword nodes, which are the descendant nodes of v, from the keyword node lists. In this procedure, v may be taken as a result to be put into the result heap R if v contains all the required keywords and its SLCA probability $Pr^G(v)$ is larger than the k-th largest probability in R and v is an ordinary node. And then the keyword distributions (except the full keyword distribution) of v will be promoted to its parent $p(v)$. If v is not an ordinary node, we directly promote all the keyword distributions of v and the path upper bound to its parent $p(v)$, by which the false positive SLCA candidates can be skipped.

To compute the upper bound δ of $p(v)$, we need add the constraint of v to $p(v)$ by multiplying (1- $Pr^G(v)$) according to Property 3 where the nodes with different types can be dealt with in a similar way. If δ is equal to or less than the k-th largest value, $p(v)$ will be recorded into a set DeleteSet, i.e., both $p(v)$ and its ancestor nodes cannot become new SLCA results w.r.t. the top-k keyword query. If δ is larger than the k-th highest value, $p(v)$ and its upper bound δ will be cached into a candidate set *UBMap*.

We repeat the above procedure for each SLCA candidate in S and promote its keyword distributions and upper bound to its parent $p(v)$. If $p(v)$ or its descendants have existed in DeleteSet, then $p(v)$ can be discarded directly without further verification. After all the SLCA candidates in S are processed in the similar way, we can have a set of potential candidates in *UBMap*, from which we can select the candidate with the highest upper bound as a promising candidate to be processed continuously.

Given a promising candidate $v \in$ *UBMap*, we first check whether v could be an SLCA answer by using node upper bound properties, Property 4 and Property 5. If so, we begin to calculate its keyword distributions. Based on the keyword distributions and the calculated path upper bound value of v, we need to update the candidates that are the ancestor of v in *UBMap*. And then $p(v)$ and its upper bound is put into *UBMap*. If the node upper bound of v is equal to or lower than the kth highest probability in R, v will be suspended. After that, we promote the upper bound of v to its parent $p(v)$ and collect the extra aggregated upper bounds of descendants of $p(v)$ from UBMap which may reduce the upper bound of $p(v)$. v will be released until its parent node needs to be computed or pruned.

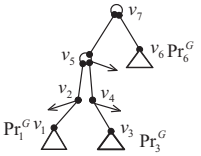


Fig. 3. Example of updating the path upper bounds

Example 7: In Figure 3, assume we have found the SLCA candidates $S = \{v_1, v_3, v_6\}$ based on traditional keyword search methods. We first calculate the SLCA probability Pr_1^G of v_1 and then calculate the upper bound of its parent v_2, i.e., δ_2 = 1 - Pr_1^G. Similarly, we can process v_3 and calculate the upper bound of its parent v_4, i.e., δ_4 = 1 - Pr_3^G. After v_6 is processed, the upper bound of v_7 is 1 - Pr_6^G. But as we know v_7 is the ancestor of v_2 and v_4, we need to aggregate the upper bounds of v_2 and v_4 to v_7. As such, δ_7 is decreased to (1 - Pr_6^G)*(1 - Pr_1^G)*(1 - Pr_3^G). At this moment, S has become empty while UBMap contains three new candidates v_2, v_4, and v_7.

Assume we have $\delta_4 > \delta_2$. v_4 will be chosen as the promising candidate because its upper bound must be larger than that of its ancestor v_7. After we process v_4, we can get its SLCA probability Pr_4^G. Next, we need to compute the upper bound of v_5 that is the parent of v_4.

Since v_7 is the ancestor of v_5, we should update the upper bound of v_7 by using the upper bound of v_4 and its SLCA probability Pr_4^G. As such, $\delta_7 = \delta_7$ *($\frac{\delta_4 - Pr_4^G}{\delta_4}$). And we know v_2 is a descendant of v_5. Hence, we need to aggregate δ_2 to v_5, i.e., we have $\delta_5 = (\delta_4 - Pr_4^G)$*$\delta_2$.

At the next step, v_2 is selected as the promising candidate. After it is processed, we also need to update the bounds of v_5 and v_7, i.e., $\delta_5 = \delta_5$ * ($\frac{\delta_2 - Pr_2^G}{\delta_2}$) = ($\delta_4 - Pr_4^G$)*($\delta_2 - Pr_2^G$) = (1 - Pr_3^G - Pr_4^G)*(1 - Pr_1^G - Pr_2^G). Similarly, we can compute the bound of v_7.

Algorithm 2 EagerTopk Algorithm

input: a keyword query $\{k_1, ..., k_n\}$ and an encoded probabilistic XML data tree T
output: top k ranked SLCA results R

1: load keyword node lists $L = \{L_i\}$, $1 \leq i \leq n$;
2: $S = get\_slca(L)$;
3: UBMap: $v \rightarrow \delta$, DeleteSet, Pr^{G}Set: $v_{slca} \rightarrow$ Pr$^G(v_{slca})$;
4: **for** each candidate $v \in S$ **do**
5: $Pr^G(v)$ = ComputeSLCAProbability(v) that is similar to Line 2-Line 18 in Algorithm 1;
6: **if** $p(v)$ cannot be deleted using DeleteSet **then**
7: set both δ_v and δ^∇ as 1 for each SLCA candidate;
8: **if** $p(v) \in$ UBMap **then**
9: UBMap($p(v)$) = UBMap($p(v)$) * (δ_v- $Pr^G(v)$);
10: **for** $v' \in$ UBMap **do**
11: **if** $p(v)//v'$ **then**
12: $\delta^\nabla = \delta^\nabla$ * UBMap(v');
13: **else if** $v'//p(v)$ **then**
14: UBMap(v') = $\frac{\text{UBMap}(v') * (\delta_v - Pr^G(v))}{\delta_v}$;
15: $\delta_{p(v)} = \delta^\nabla$ * (δ_v-$Pr^G(v)$);
16: **if** $\delta_{p(v)}$ is larger than k-th largest value in R **then**
17: UBMap.put($p(v)$, $\delta_{p(v)}$);
18: **else**
19: put node $p(v)$ into DeleteSet;
20: **repeat**
21: v, $\delta_v \leftarrow$ UBMap.removePromising();
22: **if** VerifyPromisingNode(v)=true **then**
23: $Pr^G(v)$ = ComputeSLCAProbability(v);
24: **else**
25: suspend v;
26: using Line 6 - Line 19, but δ_v is the current value;
27: **until** UBMap is empty
28: **return** R;

3) Descriptions of EagerTopK Algorithm: The pseudo-code for the eager-preferred PrKS algorithm is shown in Algorithm 2. A traditional keyword search algorithm can be employed to compute the initial SLCA candidate set S. Here we adopt the method ($get\_slca()$) in [12].

And then, we scan the keyword node lists in a document order to calculate the keyword distributions for each SLCA candidate in S by calling Function ComputeSLCAProbability(v). Similar to Line 2-Line 18 in Algorithm 1, ComputeSLCAProbability() can compute the keyword distributions according to the cached keyword distribution tables that is maintained by a hash map. After we process one node, we need to promote its keyword distributions to its parent based on the different strategies, i.e., IND, MUX, or Ordinary. Detailed procedure can be seen from Section IV-A. A little difference lies in Line 18 in Algorithm 1, at which ComputeSLCAProbability() will be terminated after node v in stack is processed, rather than running to the empty of the stack as Algorithm 1.

Line 6 - Line 19 provide the procedure of updating the path upper bounds δ in UBMap when we promote the keyword distributions of a node v to its parent $p(v)$. Before we make the promotion, we first check if the node $p(v)$ exists or is implied by a node in the delete set DeleteSet. If $p(v)$ appears (or is implied by a node) in DeleteSet, we directly process the next SLCA candidate in S. Otherwise, we are required to update UBMap based on the current result $Pr^G(v)$ to be returned by Function ComputeSLCAProbability(v). To do the update, we first check if the candidate $p(v)$ has been inserted into UBMap. If it exists, we can replace the old upper bound value UBMap($p(v)$) of $p(v)$ with UBMap($p(v)$) - $Pr^G(v)$. Otherwise, we begin to consider whether it is qualified for $p(v)$ to be inserted into UBMap. As such, we need to collect the aggregated upper bound δ^∇ from the descendants of $p(v)$ in UBMap as Line 12 while update the upper bounds of ancestors of $p(v)$ in UBMap as Line 14. After that, we can calculate the path upper bound of $p(v)$ by using δ^∇ * ($\delta_v - Pr^G(v)$) where δ_v is equal to 1. This is because we give the maximal upper bound 1 as a default value to each SLCA candidate at the beginning. In the following procedure, the value of δ_v will be updated dynamically based on the current computations. When the upper bound $\delta_{p(v)}$ is worked out, we compare it with the k-th largest probability in the result heap R. If $\delta_{p(v)}$ is larger, $p(v)$ and $\delta_{p(v)}$ will be written into UBMap as a potential candidate for future processing.

Next, in Line 20-Line 27, we repeatedly remove the most promising candidate v_m from UBMap at each time until UBMap becomes empty. The most promising candidate v_m means that it has the highest path upper bound in UBMap, i.e., $\delta_m \geq \forall \delta_i \in$ UBMap. Given a promising candidate v, we first check if it is possible to generate a new SLCA result by utilizing Function VerifyPromisingNode(v). In the function, we need to calculate the node upper bound of v according to Property 4 and Property 5, and then compare it with the k-th highest probability in R. If VerifyPromisingNode(v) returns true, it says that the node upper bound of v is larger, i.e., v may generate a satisfied SLCA result. In this case, we can access the keyword nodes that are the descendants of v in the updated keyword node lists and calculate the keyword distributions of v. At the same time, the probed keyword nodes will be removed and the keyword node lists will be updated. The procedure can be executed by calling Function ComputeSLCAProbability(v). At the next step, we promote the keyword distributions of v to its parent $p(v)$ and compute the path upper bound of $p(v)$, as in Line 6 - Line 19.

When v cannot pass the node verification successfully, it says that v cannot produce a new result anymore. As such, we can skip v and directly probe its parent $p(v)$. We use Line 6 - Line 19 to compute the upper bound of $p(v)$ and update UBMap.

TABLE II
PROPERTIES OF P-DOCUMENT

ID	name	size	#IND	#MUX	#Ordinary
Doc1	XMark	10M	16,040	16,785	169,506
Doc2		20M	46,943	46,921	340,937
Doc3		40M	69,267	70,585	676,537
Doc4		80M	179,862	178,709	1,443,987
Doc5	Modial	1.2M	4,333	4,301	30,822
Doc6	DBLP	156M	859,608	875,586	3,333,331

TABLE III
KEYWORD QUERIES FOR EACH DATASET

ID	Keyword Query	ID	Keyword Query
X_1	United States, Graduate	X_2	United States, Credit, ship
X_3	Personal, Check, alexas	X_4	Alexas, ship
X_5	internationally, ship		
M_1	muslim, multiparty	M_2	organization, United States
M_3	united states, islands	M_4	organization, pacific
M_5	chinese, polish		
D_1	Information, Retrieval, Database	D_2	XML, Keyword, Query
D_3	Query, Relational, Database	D_4	probabilistic, Query
D_5	stream, Query		

V. EXPERIMENTAL STUDIES

We conduct extensive experiments to test the performance of our algorithms that were implemented in Java and run on a 3.0GHz Intel Pentium 4 machine with 2GB RAM running Windows XP.

A. Dataset and Queries

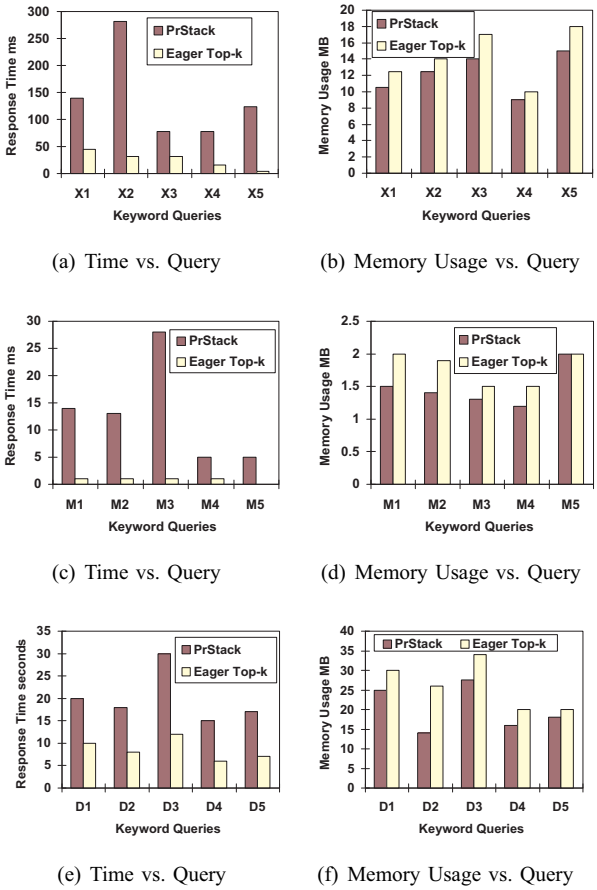
We use two real datasets, DBLP [1] and Mondial [2], and a synthetic XML benchmark dataset XMark [3] for testing the proposed algorithms. For XMark, we also generate four datasets with different sizes. The three types of datasets are selected based on their features. DBLP is a relatively shallow dataset of large size; Modial is a deep and complex, but small dataset; XMark is a balanced dataset with varied depth, complex structure and varied size. Therefore, they are chosen as test datasets.

For each XML dataset used, we generate the corresponding probabilistic XML tree, using the same method as used in [9]. We visit the nodes in the original XML tree in pre-order way. For each node v visited, we randomly generate some distributional nodes with "IND" or "MUX" types as children of v. Then, for the original children of v, we choose some of them as the children of the new generated distributional nodes and assign random probability distributions to these children with the restriction that the sum of them for a MUX node is no greater than 1. For each dataset, the percentage of the distributional nodes is controlled in about 10% - 20% of the total nodes. The generated datasets are as follows in Table II. And we randomly select terms and construct five keyword queries to be tested for each dataset, as shown in Table III.

B. Evaluation of Different Keyword Queries

Figure 4 shows the experimental results when we run the queries X_1-X_5 over Doc2, M_1-M_5 over Doc5, and D_1-D_5 over Doc6. And the required top k number is set as 10. From the results, we can find that compared with PrStack algorithm, EagerTopK algorithm can improve the time efficiency by at least 50%. Sometimes, the second algorithm can be more than 5 times faster than the first one, e.g., X_2, $X5$ in Figure 4(a) and M_1, M_2, M_3 in Figure 4(c). This is because the total real result numbers in Doc2 and Doc5 are not too large but the numbers of some matched keyword nodes are large. In this case, PrStack algorithm continues to probe the rest keyword nodes while EagerTopK algorithm can be terminated early based on the path upper bound. From Figure 4(e), we can see that PrStack algorithm consumes about 16 - 31 seconds for the given keyword queries while EagerTopK algorithm spends about 7 - 13 seconds. Most of keywords occur in leaf nodes and we know the depth of DBLP data tree is not large. Even this case is suitable to fit PrStack algorithm, EagerTopK algorithm can be executed faster than PrStack algorithm. This is because some of initial SLCA candidates hold the highest probabilities.

From Figure 4(b), Figure 4(d) and Figure 4(f), we can see that EagerTopK consumes slightly more memory usage than PrStack. This is because the keyword distributions of the intermediate results have to be cached until they are promoted to their parent nodes.



(a) Time vs. Query (b) Memory Usage vs. Query

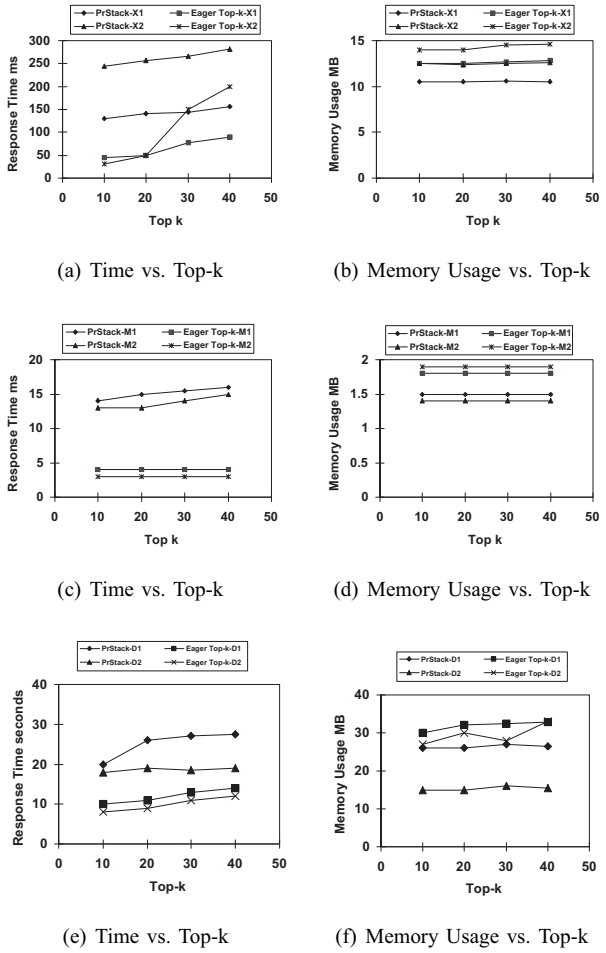
(c) Time vs. Query (d) Memory Usage vs. Query

(e) Time vs. Query (f) Memory Usage vs. Query

Fig. 4. Vary Query over Doc2, 5, 6 where k=10

[1] http://dblp.uni-trier.de/xml/
[2] http://www.dbis.informatik.uni-goettingen.de/Mondial/XML
[3] http://monetdb.cwi.nl/xml/

C. Evaluation of Different Top-k Values



(a) Time vs. Top-k (b) Memory Usage vs. Top-k

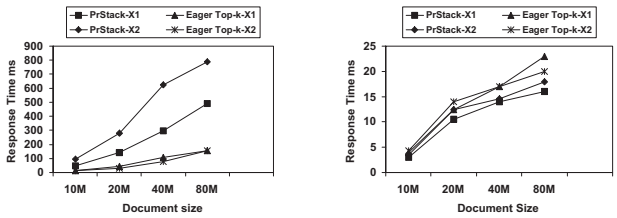
(c) Time vs. Top-k (d) Memory Usage vs. Top-k

(e) Time vs. Top-k (f) Memory Usage vs. Top-k

Fig. 5. Vary Top-k w.r.t. X1-2, M1-2, D1-2 over Doc2, 5, 6

Figure 5 shows the experimental results when we vary the top k values from 10 to 40 and we select the first two keyword queries to be tested for each dataset, i.e., X_1, X_2 over Doc2; M_1, M_2 over Doc5; and D_1, D_2 over Doc6. From the results, we can find that both algorithms will consume more time with the increase of the required top k values. Especially, when processing query X_2 over Doc2, we can see a sharp increase when k is larger than 20 in Figure 5(a). This is because the document contains 10 more or less results that have the larger probabilities than the other candidates. In addition, most of the remaining candidates have similar and small probabilistic values. Therefore, in this case, Eager Top-k algorithm needs to consume much more time when the required number is increased, however, PrStack does not change a lot because it still scans the keyword node lists once. From Figure 5(b) and Figure 5(d), we can find that memory usage nearly does not change when we increase the top k values. The trend of memory usage change a little when we evaluate keyword query D_2 over DBLP in Figure 5(f). This is because we randomly select the SLCA candidates to be computed when we get the SLCA candidate set based on traditional SLCA computation methods. If the highest ones can be found early, it will lead to pruning more nodes.

D. Evaluation of Increasing P-Document Size



(a) Time vs. Doc. Size (b) Memory Usage vs. Doc. Size

Fig. 6. Vary Document Sizes w.r.t. X1,X2, k=10

In this subsection, we take XMark dataset as an example to test the performance of the two algorithms where the top k value is specified as 10. We test all the five queries of XMark dataset, but in this paper, we only show the results of the queries X_1 and X_2. From Figure 6(a), we can see that both PrStack and EagerTopK increase linearly in response time when the document size increases from 10MB to 80MB. However, the increase of Eager TopK is much slower. The comparison illustrates that the second algorithm can obtain much better scalability than the first one when users are only interested in a small number of results from a large data source. Figure 6(b) shows the increasing trend of memory usage when the document size is varied. Both algorithms have the similar increasing trend and PrStack can save a bit more memory than EagerTopK due to the additional maintenance of *UBMap* for potential SLCA candidates in EagerTopK.

VI. Related Work

Keyword Query in Ordinary XML Documents Recently, keyword search has been investigated extensively in XML databases. Given a keyword query and an XML data source, most of related work [14], [15], [12], [16], [13] took LCAs (lowest common ancestor) or SLCAs (smallest LCA) of the matched nodes as the results to be returned. XRANK [14] and Schema-Free XQuery [15] developed stack-based algorithms to compute SLCAs. [12] introduced the Indexed Lookup Eager algorithm when the keywords appear with significantly different frequencies and the Scan Eager algorithm when the keywords have similar frequencies. [16], [17] took the similar approaches as [12]. But they focused on inferring the meaning of returned results and discussed the result differentiations in [18]. To make the results more meaningful, [19] and [20] utilized the statistics of the underlying XML data to identify the return node types. And [21] proposed a number of algorithms for cleaning keyword queries optimally. [13] designed an MS approach to compute SLCAs for keyword queries in multiple ways. [22] took the Valuable LCA (VLCA) as results by avoiding the false positive and false negative of LCA and SLCA. [23] proposed an efficient algorithm called Indexed Stack to find answers based on ELCA (Exclusive

LCA) semantics. In addition, there are other related works that process keyword search by integrating keywords into structured queries. [24] proposed a new query language XML-QL in which the structure of the query and keywords are separated. [15] introduced a method to embed keywords into XQuery to process keyword search.

Probabilistic XML The topic of probabilistic XML has been studied recently. Many models have been proposed, together with structured query evaluations. Nierman et al. [2] first introduced a probabilistic XML model, ProTDB, with the probabilistic types IND - *independant* and MUX - *mutually-exclusive*. Hung et al. [3] modeled the probabilistic XML as directed acyclic graphs, supporting arbitrary distributions over sets of children. Keulen et al. [5] used a probabilistic tree approach for data integration where its probability and possibility nodes are similar to MUX and IND, respectively. Abiteboul et al. [6] proposed a "fuzzy trees" model, where nodes are associated with conjunctions of probabilistic event variables, they also gave a full complexity analysis of query and update on the "fuzzy trees" in [1]. Cohen et al. [25] incorporated a set of constraints to express more complex dependencies among the probabilistic data. They also proposed efficient algorithms to solve the constraint-satisfaction, query evaluation, and sampling problem under a set of constraints. In [9], Kimelfeld et al. summarized and extended the probabilistic XML models previously proposed, the expressiveness and tractability of queries on different models are discussed with the consideration of IND and MUX. [8] studied the problem of evaluating twig queries over probabilistic XML that may return incomplete or partial answers with respect to a probability threshold to users. [10] proposed and addressed the problem of ranking top-k probabilities of answers of a twig query. In summary, all the above work focused on the discussions of probabilistic XML data model and/or structured XML query, e.g., twig query.

Different from all the above work, we addressed the problem of keyword search in probabilistic XML data.

VII. Conclusions

In this paper, we have addressed the problem of top-k keyword search over a general probabilistic XML model PrXML$^{\{ind,mux\}}$. Given a probabilistic XML tree T, a set of keywords and a number k, we have discussed the challenges to find k SLCA answers with the highest probabilities. A strategy have been proposed to compute the SLCA probabilities without generating possible worlds. Based on the strategy, we have proposed two efficient algorithms. The first algorithm, PrStack, only needs to scan the keyword inverted lists once, and after that the SLCA probabilities for all the nodes in T can be obtained. The second algorithm, EagerTopK, is specially designed to cater for top-k keyword search by effectively pruning unsatisfied SLCA candidates using upper bounds. The experiments have demonstrated efficiency of our algorithms.

VIII. Acknowledgments

Jianxin Li, Chengfei Liu and Rui Zhou are supported by ARC Discovery Projects DP110102407 and DP0878405, and Wei Wang is supported by ARC Discovery Projects DP0987273, DP0881779 and DP0878405.

References

[1] Pierre Senellart and Serge Abiteboul. On the complexity of managing probabilistic xml data. In *PODS*, pages 283–292, 2007.
[2] Andrew Nierman and H. V. Jagadish. ProTDB: Probabilistic data in xml. In *VLDB*, pages 646–657, 2002.
[3] Edward Hung, Lise Getoor, and V. S. Subrahmanian. Pxml: A probabilistic semistructured data model and algebra. In *ICDE*, pages 467–, 2003.
[4] Edward Hung, Lise Getoor, and V. S. Subrahmanian. Probabilistic interval xml. In *ICDT*, pages 358–374, 2003.
[5] Maurice van Keulen, Ander de Keijzer, and Wouter Alink. A probabilistic xml approach to data integration. In *ICDE*, pages 459–470, 2005.
[6] Serge Abiteboul and Pierre Senellart. Querying and updating probabilistic information in xml. In *EDBT*, pages 1059–1068, 2006.
[7] Serge Abiteboul, Benny Kimelfeld, Yehoshua Sagiv, and Pierre Senellart. On the expressiveness of probabilistic xml models. *VLDB J.*, 18(5):1041–1064, 2009.
[8] Benny Kimelfeld and Yehoshua Sagiv. Matching twigs in probabilistic xml. In *VLDB*, pages 27–38, 2007.
[9] Benny Kimelfeld, Yuri Kosharovsky, and Yehoshua Sagiv. Query efficiency in probabilistic xml models. In *SIGMOD Conference*, pages 701–714, 2008.
[10] Lijun Chang, Jeffrey Xu Yu, and Lu Qin. Query ranking in probabilistic xml data. In *EDBT*, pages 156–167, 2009.
[11] Benny Kimelfeld, Yuri Kosharovsky, and Yehoshua Sagiv. Query evaluation over probabilistic xml. *VLDB J.*, 18(5):1117–1140, 2009.
[12] Yu Xu and Yannis Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 537–538, 2005.
[13] Chong Sun, Chee Yong Chan, and Amit K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.
[14] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27, 2003.
[15] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.
[16] Ziyang Liu and Yi Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.
[17] Ziyang Liu and Yi Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.
[18] Ziyang Liu, Peng Sun, and Yi Chen. Structured search result differentiation. *PVLDB*, 2(1):313–324, 2009.
[19] Zhifeng Bao, Tok Wang Ling, Bo Chen, and Jiaheng Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, pages 517–528, 2009.
[20] Jianxin Li, Chengfei Liu, Rui Zhou, and Wei Wang. Suggestion of promising result types for xml keyword search. In *EDBT*, pages 561–572, 2010.
[21] Ken Q. Pu and Xiaohui Yu. Keyword query cleaning. *PVLDB*, 1(1):909–920, 2008.
[22] Guoliang Li, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.
[23] Yu Xu and Yannis Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.
[24] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6):119–135, 2000.
[25] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Incorporating constraints in probabilistic xml. *ACM Trans. Database Syst.*, 34(3), 2009.