ELCA evaluation for keyword search on probabilistic XML data

Rui Zhou · Chengfei Liu · Jianxin Li · Jeffrey Xu Yu

Received: 18 October 2011 / Revised: 13 February 2012 / Accepted: 26 April 2012 / Published online: 16 May 2012 © Springer Science+Business Media, LLC 2012

Abstract As probabilistic data management is becoming one of the main research focuses and keyword search is turning into a more popular query means, it is natural to think how to support keyword queries on probabilistic XML data. With regards to keyword query on deterministic XML documents, ELCA (Exclusive Lowest Common Ancestor) semantics allows more relevant fragments rooted at the ELCAs to appear as results and is more popular compared with other keyword query result semantics (such as SLCAs). In this paper, we investigate how to evaluate ELCA results for keyword queries on probabilistic XML documents. After defining probabilistic ELCA semantics in terms of possible world semantics, we propose an approach to compute ELCA probabilities without generating possible worlds. Then we develop an efficient stack-based algorithm that can find all probabilistic ELCA results and their ELCA probabilities for a given keyword query on a probabilistic XML document. Finally, we experimentally evaluate the proposed ELCA algorithm and compare it with its SLCA counterpart in aspects of result probability, time and space efficiency, and scalability.

Keywords ELCA · probabilistic · XML · keyword search · keyword query · uncertain
1 Introduction

Uncertain data management is currently one of the main research focuses in database community. Uncertain data may be generated by different reasons, such as limited observation equipment, unsupervised data integration, conflicting feedbacks. Moreover, uncertainty itself is inherent in nature. This drives the technicians to face the reality and develop specific database solutions to embrace the uncertain world. In many web applications, such as information extraction, a lot of uncertain data are automatically generated by crawlers or mining systems, and most of the time they are from tree-like raw data. In consequence, it is natural to organize the extracted information in a semi-structured way with probabilities attached showing the confidence for the collected information. In addition, dependencies between extracted information can be easily captured by parent-child relationship in a tree-like XML document. Consider another application, users collaborate to edit documentation structured in sections, subsections, paragraphs and so on in the online encyclopedia Wikipedia. In an open environment, some of these contributions may be incorrect, or even spam and vandalism. If we have some way to estimate the trustworthiness of a contributor, we can represent each individual edit as an uncertain operation on a probabilistic XML document that represents the integration of all previous edits [13]. There are other probabilistic XML applications mentioned in [13]. Driven by these applications, research on probabilistic XML data management is extensively under way.

Many probabilistic models [1, 2, 8, 9, 22, 24, 27] have been proposed to describe probabilistic XML data. The expressiveness between different models is discussed in [1]. Beyond the above, querying probabilistic XML data to retrieve useful information is of equal importance. Current studies mainly focused on twig queries [4, 11, 23], with little light [17] shed on keyword queries on probabilistic XML data. However, support for keyword search is important and promising, because users will be relieved from learning complex query languages (such as XPath, XQuery) and are not required to know the schema of the probabilistic XML document. A user only needs to submit a few keywords and the system will automatically find some suitable fragments from the probabilistic XML document.

There has been established works on keyword search over deterministic XML data. One of the most popular semantics to model keyword query results on a deterministic XML document is the ELCA (Exclusive Lowest Common Ancestor) semantics [7, 29, 30]. We introduce the ELCA semantics using an example. Formal definitions will be introduced in Section 2. Figure 1a shows an ordinary XML tree. Nodes \{a_1, a_2, a_3\} directly contain keyword \(a\), and nodes \{b_1, b_2, b_3, b_4\} directly contain keyword \(b\). Node \{x_1, x_2, x_4\} are considered as ELCA of keywords \(a\) and \(b\). An ELCA is firstly an LCA, and after excluding all its children which contain all keywords, the LCA still contains all the keywords. Node \(x_2\) is an ELCA, because after excluding \(x_1\) which contains all the keyword, \(x_2\) still has its own contributors \(a_1\) and \(b_2\). Node \(x_3\) is not an ELCA, because after excluding \(x_4\), \(x_3\) only covers keyword \(b\). Nodes \(x_1\) and \(x_4\) are also ELCA, because they contain both keywords. No children of \(x_1\) or \(x_4\) contain all the keywords, so no need to exclude any child from \(x_1\) or \(x_4\).

Another popular semantics is SLCA (Smallest LCA) semantics [25, 28]. It asks for the LCAs that are not ancestors of other LCAs. For example, node \(x_1\) and \(x_4\) are SLCAs on the tree, but \(x_2\) is not, because it is an ancestor of \(x_1\). It is not difficult to see that the ELCA result is a superset of the SLCA result, so the ELCA semantics...
can provide more interesting information to users. This motivates us to study the ELCA semantics, and particularly on a new type of data, probabilistic XML data. Note that although SLCA semantics is studied on probabilistic XML data in [17], the solution cannot be used to solve ELCA semantics, as readers may notice that the ELCA semantics is indeed more complex than the SLCA semantics.

On a probabilistic XML document, nodes may appear or not, accordingly a node is (usually) not certain to be an ELCA. As a result, we want to find not only those possible ELCA nodes, but also their ELCA probabilities. Before we point out the computation challenge, we briefly introduce the probabilistic XML model used throughout this paper. We consider a popular probabilistic XML model, $\text{PrXML}_{\{\text{ind}, \text{mux}\}}$ [10, 22], where a probabilistic XML document (also called p-document) is regarded as a tree with two types of nodes: ordinary nodes and distributional nodes. Ordinary nodes store the actual data and distributional nodes define the probability distribution for the child nodes. There are two types of distributional nodes: IND and MUX. IND means the child nodes may appear independently and MUX means the child nodes are mutually-exclusive (i.e. only one child can appear among the defined alternative children). 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. A randomly generated document from a p-document is called a possible world. Apparently, each possible world has a probability. The sum of the probabilities of all possible worlds is 1. A probabilistic XML tree is given in Figure 1b, where unweighted edges have the default probability 1.

Given a keyword query, and a p-document, a node may be an ELCA of the keywords in some possible worlds but not in other possible worlds. We cannot ignore the distributional nodes, because the ELCA results on a probabilistic XML tree may be totally different from those on a deterministic XML tree. For example, in Figure 1b, $x_4$ is no longer an ELCA due to the MUX semantics. $x_3$ may become an ELCA if a possible world contains $a_3$ not $b_3$, but on the deterministic version, $x_3$ is never an ELCA node. Furthermore, $x_1$, a 100% ELCA node in Figure 1a, becomes a conditional ELCA with probability $0.8 \times 0.6 \times 0.7$ in Figure 1b. $x_2$ also becomes an 80% ELCA node. As a result, deterministic ELCA solutions [7, 29, 30] are not applicable to the new problem. Furthermore, to find out the possible ELCA nodes is not enough. Users may want to know the ELCA probabilities of the possible ELCA.
To solve the problem, a straightforward and safe method is to generate all possible worlds from the given p-document, evaluate ELCAAs using existing ELCA algorithms on deterministic XML for each possible world, and combine the result finally. However, it is obvious that this method is infeasible, because the computation cost is too high, since the number of possible worlds is exponential. The challenge is how to evaluate the ELCA probability of a node using only the p-document without generating possible worlds. The idea of our approach is to evaluate the ELCA probabilities in a bottom-up manner.

We summarize the contributions of this paper as follows:

- To the best of our knowledge, this is the first work that studies ELCA semantics on probabilistic XML data.
- We have defined probabilistic ELCA semantics for keyword search on probabilistic XML documents. We have proposed an approach on how to evaluate ELCA probabilities without generating possible world and have designed a stack-based algorithm, PrELCA algorithm, to find the probabilistic ELCAAs and their probabilities.
- We have conducted extensive experiments to test the result probability, time and space efficiency, scalability of the PrELCA algorithm.

The rest of this paper is organized as follows. In Section 2, we introduce ELCA semantics on a deterministic XML document and define probabilistic ELCA semantics on a probabilistic XML document. In Section 3, we propose how to compute ELCA probabilities on a probabilistic XML document without generating possible worlds. An algorithm, PrELCA, is introduced in Section 4 to explain how to put the conceptual idea in Section 3 into procedural computation steps. We report the experiment results in Section 5. Related works and Conclusion are in Section 6 and Section 7 respectively.

2 Preliminaries

In this section, we first introduce ELCA semantics on a deterministic XML document, and then define probabilistic ELCA semantics on a probabilistic XML document.

2.1 ELCA semantics on deterministic XML

A deterministic XML document is usually modeled as a labeled ordered tree. Each node of the XML tree corresponds to an XML element, an attribute or a text string. The leaf nodes are all text strings. A keyword may appear in element names, attribute names or text strings. If a keyword $k$ appears in the subtree rooted at a node $v$, we say the node $v$ contains keyword $k$. If $k$ appears in the element name or attribute name of $v$, or $k$ appears in the text value of $v$ when $v$ is a text string, we say node $v$ directly contains keyword $k$. A keyword query on a deterministic XML document often asks for an XML node that contains all the keywords, therefore, for large XML documents, indexes are often built to record which nodes directly contain which keywords. For example, for a keyword $k_i$, all nodes directly contain $k_i$ are stored in a list $S_i$ (called inverted list) and can be retrieved altogether at once.
We adopt the formalized ELCA semantics as the work [29]. We introduce some notions first. Let \( v \prec_a u \) denote \( v \) is an ancestor node of \( u \), and \( v \preceq_a u \) denote \( v \prec_a u \) or \( v = u \). The function \( \text{lca}(v_1, \ldots, v_n) \) computes the Lowest Common Ancestor (LCA) of nodes \( v_1, \ldots, v_n \). The LCA of sets \( S_1, \ldots, S_n \) is the set of LCAs for each combination of nodes in \( S_1 \) through \( S_n \).

\[
\text{lca}(k_1, \ldots, k_n) = \text{lca}(S_1, \ldots, S_n) = \{ \text{lca}(v_1, \ldots, v_n) | v_1 \in S_1, \ldots, v_n \in S_n \}
\]

Given \( n \) keywords \( \{k_1, \ldots, k_n\} \) and their corresponding inverted lists \( S_1, \ldots, S_n \) of an XML tree \( T \), the Exclusive LCA of these keywords on \( T \) is defined as:

\[
elca(k_1, \ldots, k_n) = \text{elca}(S_1, \ldots, S_n)
= \{ v | \exists v_1 \in S_1, \ldots, v_n \in S_n (v = \text{lca}(v_1, \ldots, v_n)
\quad \land \forall i \in [1, n] \exists x (x \in \text{lca}(S_1, \ldots, S_n) \land \text{child}(v, v_i) \preceq_a x)) \}
\]

where \( \text{child}(v, v_i) \) denotes the child node of \( v \) on the path from \( v \) to \( v_i \). The meaning of a node \( v \) to be an ELCA is: \( v \) should contain all the keywords in the subtree rooted at \( v \), and after excluding \( v \)'s children which also contain all the keywords from the subtree, the subtree still contains all the keywords. In other words, for each keyword, node \( v \) should have its own keyword contributors.

### 2.2 ELCA semantics on 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 world. A p-document can be modelled as a labelled tree \( T \) with ordinary and distributional nodes. Ordinary nodes are regular XML nodes that may appear in deterministic documents, while distributional nodes are used for describing a probabilistic process following which possible worlds can be generated. Distributional nodes do not occur in deterministic documents.

We define ELCA semantics on a p-document with the help of possible worlds of the p-document. Given a p-document \( T \) and a keyword query \( \{k_1, k_2, \ldots, k_n\} \), we define probabilistic ELCA of these keywords on \( T \) as a set of node and probability pairs \( (v, \text{Pr}_{\text{elca}}^G(v)) \). Each node \( v \) is an ELCA node in at least one possible world generated by \( T \), and its probability \( \text{Pr}_{\text{elca}}^G(v) \) is the aggregated probability of all possible worlds that have node \( v \) as an ELCA. The formal definition of \( \text{Pr}_{\text{elca}}^G(v) \) is as follows:

\[
\text{Pr}_{\text{elca}}^G(v) = \sum_{i=1}^{m} \{ \text{Pr}(w_i) | \text{elca}(v, w_i) = \text{true} \}
\]

where \( \{w_1, \ldots, w_m\} \) denotes the set of possible worlds implied by \( T \), \( \text{elca}(v, w_i) = \text{true} \) indicates that \( v \) is an ELCA in the possible world \( w_i \), \( \text{Pr}(w_i) \) is the existence probability of the possible world \( w_i \).

To develop the above discussion, \( \text{Pr}_{\text{elca}}^G(v) \) can also be computed with (2). Here, \( \text{Pr}(\text{path}_{v \rightarrow v_i}) \) 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 node $v$. $Pr_{elca}^L(v)$ is the local probability for $v$ being an ELCA in $T_{sub}(v)$, where $T_{sub}(v)$ denotes a subtree of $T$ rooted at $v$.

$$Pr_{elca}^G(v) = Pr(path_{r\rightarrow v}) \times Pr_{elca}^L(v) \quad (2)$$

To compute $Pr_{elca}^L(v)$, we have the following equation similar to (1).

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

where deterministic trees $\{t_1, t_2, ..., t_{m'}\}$ are local possible worlds generated from $T_{sub}(v)$, $Pr(t_i)$ is the probability of generating $t_i$ from $T_{sub}(v)$; $elca(v, t_i) = true$ means $v$ is an ELCA node in $t_i$.

In the following sections, we mainly focus on how to compute the local ELCA probability, $Pr_{elca}^L(v)$ for a node $v$. $Pr(path_{r\rightarrow v})$ is easy to obtain if we have index recording the probabilities from the root to node $v$. Then it is not difficult to have the global probability $Pr_{elca}^G(v)$ using (2).

3 ELCA probability computation

In this section, we introduce how to compute ELCA probabilities for nodes on a p-document without generating possible worlds. We start from introducing keyword distribution probabilities in Section 3.1, and then introduce how to compute the ELCA probability for a node $v$ using keyword distribution probabilities of $v$’s children in Section 3.2.

3.1 Keyword distribution probabilities

As introduced in the Section 2, in a particular possible world of a p-document, a node $v$ is an ELCA with respect to a set of keywords, if some children of $v$ witness strict subsets of the keywords and those children together witness the full set of the keywords. As a result, to compute the ELCA probability of node $v$, we first compute the probabilities of $v$’s children witnessing the subsets.

Given a keyword query $Q = \{k_1, ..., k_n\}$ with $n$ keywords, for each node $v$ in the p-document $T$, we can assign an array $tab_v$ with size $2^n$ to record the keyword distribution probabilities under $v$. For example, let $\{k_1, k_2\}$ be a keyword query, entry $tab_v[11]$ records the probability when $v$ contains both $k_1$ and $k_2$ in all possible worlds produced by $T_{sub}(v)$; similarly, $tab_v[01]$ stores the probability when $v$ contains only $k_2$; $tab_v[10]$ keeps the probability when $v$ contains only $k_1$; and $tab_v[00]$ records the probability when neither of $k_1$ and $k_2$ appears under $v$. Note that the probabilities stored in $tab_v$ of node $v$ are local probabilities, i.e. these probabilities are based on the condition that node $v$ exists in the possible worlds produced by $T$. To implement $tab_v$, we only need to store non-zero entries of $tab_v$ using a HashMap to save space cost, but, for the clearness of discussion, let us describe $tab_v$ as an array with $2^n$ entries.

For a leaf node $v$ in $T$, the entries of $tab_v$ are either 1 or 0. Precisely speaking, one entry is “1”, and all the other entries are “0”. When $v$ is an internal node, let $v$’s children be $\{c_1, ..., c_m\}$, let $\lambda_i$ be the conditional probability when $c_i$ appears
under \( v \), then \( \text{tab}_{v} \) can be computed using \( \{ \text{tab}_{c_1}, \ldots, \text{tab}_{c_m} \} \) and \( \{ \lambda_1, \ldots, \lambda_m \} \). We will elaborate the computation for different types of \( v \): ordinary nodes, MUX nodes and IND nodes.

As an astute reader, you may find that keywords may appear on the intermediate nodes themselves in their element/attribute names. It is not sufficient to consider only the children of an intermediate node. In such cases, we can add a virtual text string node to the intermediate node with parent-child probability 1.0 carrying the element/attribute name as the text string. As a result, keywords will only appear in the leaf nodes. If a virtual text node is found as an ELCA, we consider that this ELCA is referring to the parent node of the virtual node. The above special treatment simplifies our following discussions.

### 3.1.1 Node \( v \) is an ordinary node

When \( v \) is an ordinary node, all the children of \( v \) will definitely appear under \( v \), so we have \( \lambda_1 = \ldots = \lambda_m = 1 \). Let \( \text{tab}_{v}[\mu] \) be an entry in \( \text{tab}_{v} \), where \( \mu \) is a binary expression of the entry index (eg. \( \text{tab}_{v}[101] \) refers to \( \text{tab}_{v}[5] \), here \( \mu = \text{“101”} \)), then \( \text{tab}_{v}[\mu] \) can be computed using the following equation:

\[
\text{tab}_{v}[\mu] \leftarrow \sum_{\mu = \mu_1|\mu_2 \ldots |\mu_m} \prod_{i=1}^{m} \text{tab}_{c_i}[\mu_i]
\]  

(4)

Here, \( \text{tab}_{c_i}[\mu_i] \) is an entry in \( \text{tab}_{c_i} \), \( \mu_i \) gives the keyword occurrences under \( v \)’s child \( c_i \), and \( \mu_1|\mu_2 \ldots |\mu_m \) gives the keyword occurrences among all \( v \)’s children, operator “|” stands for bitwise OR. Different \( \{ \mu_1, \ldots, \mu_m \} \) combinations (i.e. \( \mu_1|\mu_2 \ldots |\mu_m \)) may produce the same \( \mu \), so the total probability of these combinations gives \( \text{tab}_{v}[\mu] \).

Figure 2a shows an example, where \( v \) is an ordinary node. \( c_1, c_2 \) are \( v \)’s children. \( v \)’s keyword distribution table can be computed using \( c_1, c_2 \)’s keyword distribution table.

---

**Figure 2** Evaluation of keyword distribution probabilities.
tables. Take entry \( \text{tab}_v[01] \), denoted as \( p_2 \), as an example: \( p_2 \) stands for the case that \( v \) contains keyword \( k_2 \) but does not contain \( k_1 \). It correspondingly implies three cases: (1) \( c_1 \) contains \( k_2 \) and \( c_2 \) contains neither \( k_1 \), \( k_2 \); (2) \( c_2 \) contains \( k_2 \) and \( c_1 \) contains neither; (3) both \( c_1 \), \( c_2 \) only contains keyword \( k_2 \). The probability sum of the three cases gives the local probability \( p_2 \).

The naive way to compute \( \text{tab}_v \) based on (4) results in an \( O(m2^m) \) algorithm, because each \( \text{tab}_c_i \) contains \( 2^n \) entries, and there are \( (2^n)^m = 2^{mn} \) combinations of \( \{\mu_1, ..., \mu_m\} \). For each combination, computing \( \prod_{i=1}^{m} \text{tab}_c_i[\mu_i] \) takes \( O(m) \) time. However, we can compute \( \text{tab}_v \) progressively in \( O(m2^{2n}) \) time. The idea is to use an intermediate array \( \text{tab}'_v \) to record a temporary distribution and then combine the intermediate array \( \text{tab}'_v \) with each \( \text{tab}_c_i \) one by one (not all together). We now illustrate the process: at the beginning, \( \text{tab}'_v \) is initialized using (5), and then each \( \text{tab}_c_i \) \((i \in [1, m]) \) is merged with \( \text{tab}'_v \) based on (6), in the end, after incorporating all \( v \)'s children, \( \text{tab}_v \) is set as \( \text{tab}'_v \). Figure 3 illustrates the whole process.

\[
\text{tab}'_v[\mu] \left\{ \begin{array}{ll}
0 & (\mu \neq 00...0) \\
1 & (\mu = 00...0) 
\end{array} \right. 
\]  

(5)

\[
\text{tab}'_v[\mu] \leftarrow \sum_{\mu = \mu'} \text{tab}'_v[\mu'] \cdot \text{tab}_c_i[\mu_i] 
\]  

(6)

Basically, “(5) and (6)” is the iteration representation of (4). We now prove they are equal using induction.

**Proof** For \( m = 1 \) (\( v \) has single child) and \( m = 2 \) (\( v \) has two children), it is not difficult to verify that (4) equals to “(5) and (6)” . To be specific, when \( m = 1 \), both (4) and “(5) and (6)” give \( \text{tab}_v \leftarrow \text{tab}_c_1 \); when \( m = 2 \), both (4) and “(5) and (6)” give \( \text{tab}_v[\mu] \leftarrow \sum_{\mu = \mu_1, \mu_2} \text{tab}_c_1[\mu_1] \cdot \text{tab}_c_2[\mu_2] \);
Suppose (4) equals to “(5) and (6)” when \( m = j \), we want to prove they are equal when \( m = j + 1 \). When \( m = j + 1 \), (4) is represented as 
\[
\text{tab}_v[\mu] \leftarrow \sum_{\mu = \mu_1|...|\mu_{j+1}} \prod_{i=1}^{j+1} \text{tab}_{c_i}[\mu_i],
\]
then we have:
\[
\text{tab}_v[\mu] = \sum_{\mu = \mu_1|...|\mu_{j+1}} \prod_{i=1}^{j+1} \text{tab}_{c_i}[\mu_i]
\]
\[
= \sum_{\mu = \mu_1|...|\mu_{j+1}} \prod_{i=1}^{j} \text{tab}_{c_i}[\mu_i] \cdot \text{tab}_{c_{j+1}}[\mu_{j+1}]
\]
\[
= \sum_{\mu = \mu_1|...|\mu_{j+1}} \left( \sum_{\mu' = \mu_1|...|\mu_j} \prod_{i=1}^{j} \text{tab}_{c_i}[\mu_i] \right) \cdot \text{tab}_{c_{j+1}}[\mu_{j+1}]
\]

Let \( \mu' = \mu_1|...|\mu_j \), according to the assumption that it holds for \( m = j \), we can replace \( \sum_{\mu' = \mu_1|...|\mu_j} \prod_{i=1}^{j} \text{tab}_{c_i}[\mu_i] \) with \( \text{tab}'_v[\mu'] \). Equation (4) then becomes the following:
\[
\text{tab}_v[\mu] \leftarrow \sum_{\mu' = \mu_1|...|\mu_j} \text{tab}'_v[\mu'] \cdot \text{tab}_{c_{j+1}}[\mu_{j+1}]
\]

It is now easy to see that (4) equals to “(5) and (6)” when \( m = j + 1 \). The proof is over. \( \square \)

Generally speaking, we are expecting a keyword query consisting of less than \( n \leq 5 \) keywords, so \( 2^n \) is not very large, besides the number of none-zero entries is much smaller than the theoretical bound \( 2^n \), and therefore we can consider the complexity of computing \( \text{tab}_v \) as \( O(m) \). When we are facing too many keywords, a situation out of the scope of this paper, some preprocessing techniques may be adopted to cut down the number of keywords, such as correlating a few keywords as one phrase. In this paper, we keep our discussion on queries with only a few keywords.

### 3.1.2 Node \( v \) is an MUX node

For an MUX node, (7) shows how to compute \( \text{tab}_v[\mu] \) under mutually-exclusive semantics. A keyword distribution \( \mu \) appearing at node \( v \) implies that \( \mu \) appears at one of \( v \)'s children, and thus \( \sum_{i=1}^{m} \lambda_i \cdot \text{tab}_{c_i}[\mu] \) gives \( \text{tab}_v[\mu] \). The case \( \mu = \"00...0\" \) is specially treated. An example is given in Figure 2b, consider node \( v \) as an MUX node now, then \( \text{tab}_{v}[01] \) can be computed as \( \lambda_1 \cdot \text{tab}_{c_1}[01] + \lambda_2 \cdot \text{tab}_{c_2}[01] \). Differently, the entry \( \text{tab}_{v}[00] \) includes an extra \( (1 - \lambda_1 - \lambda_2) \) component, because the absence of both \( c_1 \) and \( c_2 \) also implies that node \( v \) does not contain any keywords.

\[
\text{tab}_v[\mu] \leftarrow \begin{cases} 
\sum_{i=1}^{m} \lambda_i \cdot \text{tab}_{c_i}[\mu] & (\mu \neq \text{\"00...0\"}) \\
\sum_{i=1}^{m} \lambda_i \cdot \text{tab}_{c_i}[0] + 1 - \sum_{i=1}^{m} \lambda_i & (\mu = \text{\"00...0\"})
\end{cases}
\] (7)

Similar to the ordinary case, \( \text{tab}_v \) can be progressively computed under mutually-exclusive semantics as well. At the beginning, initialize table \( \text{tab}'_v \) using (5), the same as the ordinary case; and then \( \text{tab}'_v \) is increased by merging with \( \text{tab}_{c_i} \ (i \in [1, m]) \).
progressively using (8). In the end, set \( \text{\textit{tab}}_{v} \) as \( \text{\textit{tab}}'_{v} \). Both the straightforward and the progressive methods take \( O(m^2n) \) complexity.

\[
\text{\textit{tab}}'_{v}[\mu] \leftarrow \begin{cases} 
\text{\textit{tab}}'_{v}[\mu] + \lambda_i \cdot \text{\textit{tab}}_{c_i}[\mu] & (\mu \neq 00\ldots0) \\
\text{\textit{tab}}'_{v}[0] + \lambda_i \cdot \text{\textit{tab}}_{c_i}[0] - \lambda_i & (\mu = 00\ldots0)
\end{cases}
\] (8)

3.1.3 Node \( v \) is an IND node

When \( v \) is an IND node, the computation of \( \text{\textit{tab}}_{v} \) is similar to the ordinary case. Before directly applying (5) and (6), we need to standardize the keyword distribution table. The idea is to transform edge probability \( \lambda_i \) into 1, and make corresponding changes to the keyword distribution table with no side-effects. The modification is based on (9). An example is shown in Figure 2c, \( c_1 \) and \( c_2 \) are two children of IND node \( v \) with probabilities \( \lambda_1, \lambda_2 \), we can equally transform the keyword distribution tables into the right ones and change probabilities on the edges into 1. The \( \text{\textit{tab}}_{c_1}[00] \) and \( \text{\textit{tab}}_{c_2}[00] \) fields have \((1 - \lambda_1)\) and \((1 - \lambda_2)\) components, because the absence of a child also implies that no keyword instances could appear under that child. After the transformation, we can compute \( v \)'s keyword distribution table using the transformed keyword distribution tables of \( c_1 \) and \( c_2 \) following the same way as Section 3.1.1.

\[
\text{\textit{tab}}_{c_i}[\mu] \leftarrow \begin{cases} 
\lambda_i \cdot \text{\textit{tab}}_{c_i}[\mu] & (\mu \neq 00\ldots0) \\
\lambda_i \cdot \text{\textit{tab}}_{c_i}[0] + 1 - \lambda_i & (\mu = 00\ldots0)
\end{cases}
\] (9)

In summary, we can obtain keyword distribution probabilities for every node in the p-document. The computation can be done in a bottom-up manner progressively. In the next section, we will show how to obtain the ELCA probability of node \( v \) using keyword distribution probabilities of \( v \)'s children.

3.2 ELCA probability

We consider ELCA nodes to be ordinary nodes only. However, although IND/MUX nodes cannot be ELCA nodes according to the semantics, we still do a similar ELCA probability computation for these IND/MUX nodes as we do for ordinary nodes, because the ELCA probability value of an IND/MUX node will be carried upwards and set as the ELCA probability of its parent node. Here, we will use “ELCA probability of IND/MUX nodes” for ease of presentation, though the concept is not really meaningful. We first point out two cases in which we do not need to compute the ELCA probability or we can simply reuse the ELCA probability of a child node, after that we discuss when we need to compute ELCA probabilities and how to do it using keyword distribution table.

Case 1: \( v \) is an ordinary node, and \( v \) has a distributional node (either IND or MUX) as a single child. For this case, we do not need to compute ELCA probability for \( v \), because the child distributional node will pass its probability upward to \( v \).

Case 2: \( v \) is an MUX node and has ELCA probability as 0. According to the MUX semantics, \( v \) has a single child. If the child does not contain all the keywords, then \( v \) does not contain all the keywords either, on the other hand, if the
child contains all the keywords, the child will screen the keywords from contributing upwards. Node $v$ still does not contain its own keyword contributors. In both situations, $v$ is not regarded as an ELCA.

In other cases, including $v$ is an ordinary/IND node and $v$ has a set of ordinary nodes as children, we need to compute ELCA probability for $v$. Note that, when $v$ is an IND node, although $v$ cannot be considered as an ELCA result, we still compute its ELCA probability, because this probability will be passed to $v$’s parent according to Case 1. We discuss the ordinary node case first, IND node is similar. We first define a concept, contributing distribution, for the sake of better presenting the idea.

**Definition 1** Let $\mu$ be a binary expression of an entry index representing a keyword-distribution case, we define $\hat{\mu}$ as the contributing distribution of $\mu$ with the value as follows:

$$\hat{\mu} \left\{ \begin{array}{ll} \mu & (\mu \neq 1...1) \\ 00...0 & (\mu = 1...1) \end{array} \right\}$$

(10)

It means that $\hat{\mu}$ remains the same as $\mu$ in the most cases, except that when $\mu$ is “11...1”, $\hat{\mu}$ is set to “00...0”. According to ELCA semantics, if a child $c_i$ of node $v$ has contained all the keywords, $c_i$ will screen the keyword instances from contributing upward to the its parent $v$. This is our motivation to define $\hat{\mu}$. That is to say: when $\mu_i$ is “11...1”, we regard the contributing distribution $\hat{\mu}_i$ of $\mu_i$ (to parent node $v$) as “00...0”.

For an ordinary node $v$, let $\{c_1, ..., c_m\}$ be $v$’s children and $\{tab_{c_1}, ..., tab_{c_m}\}$ be the keyword distribution probability arrays of $\{c_1, ..., c_m\}$ respectively, let $\hat{\mu}_i$ be the corresponding contributing distribution of $\mu$, (11) gives how to compute the local ELCA probability, $Pr_{elca}^L(v)$, for node $v$ using $\{tab_{c_1}, ..., tab_{c_m}\}$.

$$Pr_{elca}^L(v) \left\{ \begin{array}{ll} \sum_{111...1 = \hat{\mu}_1 | ... | \hat{\mu}_m} \prod_{i=1}^{m} tab_{c_i}[\mu_i] \end{array} \right\}$$

(11)

To explain (11), $v$ is an ELCA when the disjunction of $\hat{\mu}_1, ..., \hat{\mu}_m$ is “11...1”, which means after excluding all the children of $v$ containing all the keywords, $v$ still contains all the keywords under other children. All such $\{\hat{\mu}_1, ..., \hat{\mu}_m\}$ combinations contribute to $Pr_{elca}^L(v)$, and hence the right part of (11) gives an intuitive way to compute $Pr_{elca}^L(v)$.

Similar to keyword distribution probabilities, we can compute $Pr_{elca}^L(v)$ in a progressive way, reducing the computation complexity from $O(m2^{nm})$ to $O(m2^{2n})$. An intermediate array of size $2^n$ is used, denoted as $tab''_v$. Here, the function of $tab''_v$ is similar to that of $tab'_v$ used in the last section. To be specific, at the beginning, $tab''_v$ is initialized by (12). As the computation goes on, $tab''_v$ is continuously merged with $tab_{c_i}$ ($i \in [1, m]$) using (13). In the end, after merging the intermediate table with all $v$’s children one by one, entry $tab''_v[111...1]$ gives $Pr_{elca}^L(v)$. Note that, although only one entry of $tab''_v$, $tab''_v[111...1]$, is required as the final result. In the computation, we
need to store the whole table $tab''$, because other entries are used to compute the final $tab''_{v}[11...1]$ entry.

$$tab''_{v}[\mu] \left\{ \begin{array}{ll}
0 & (\mu \neq 00...0) \\
1 & (\mu = 00...0)
\end{array} \right. \tag{12}$$

$$tab''_{v}[\mu] \left\{ \begin{array}{ll}
\sum_{\mu = \mu'|\mu_i} tab''_{v}[\mu'] \cdot tab_{c_i}[\mu_i] & \\
\end{array} \right. \tag{13}$$

For each child $c_i$, when we compute $Pr_{elca}^L(v)$, the array entry $tab_{c_i}[11...1]$ acts the same as the entry $tab_{c_i}[00...0]$, because it does not contribute any keyword to its parent. In consequence, we can first modify $tab_{c_i}$ with (14), and reuse (6) to compute $Pr_{elca}^L(v)$.

$$tab_{c_i}[\mu] \left\{ \begin{array}{ll}
tab_{c_i}[00...0] + tab_{c_i}[11...1] & (\mu = 00...0) \\
tab_{c_i}[\mu] & (\mu = 11...1) \\
\end{array} \right. \tag{14}$$

For an IND node $v$, we can standardize the keyword distribution table using (9). Then, the computation is the same as the ordinary node case.

In Figure 4b, we give an example to show how to compute the intermediate table $tab''_{v}$. An ordinary node $v$ has two children $c_1$, $c_2$. Their keyword distribution tables have been modified according to (14). The probability of $v$ containing both keywords (coming from different children) is given by $p_3 = x_2 \cdot y_3 + x_3 \cdot y_2$, which implies two cases: (1) $c_1$ contains $k_1$ and $c_2$ contains $k_2$; (2) $c_1$ contains $k_2$ and $c_2$ contains $k_1$. Neither $c_1$, $c_2$ are allowed to solely contain both keywords. In ELCA semantics, if a node contains all the keywords, the node will not make contributions to its parent. The probability is smaller than the probability $p_3 = x_4 + y_4 + x_2 \cdot y_3 + x_3 \cdot y_2$ (given in Figure 4a), which is the keyword distribution probability when node $v$ contains both keywords, but not required to be from different children. Similarly, the calculation of $tab'_{v}[01]$ and $tab''_{v}[01]$ (i.e. $p_2$) are also different.

### 4 Algorithm

In this section, we introduce an algorithm, PrELCA, to put the conceptual idea in the previous section into procedural computation steps. We start with introducing
how to index probabilistic XML data to support probabilistic ELCA computation, and afterwards we introduce the PrELCA algorithm.

4.1 Indexing probabilistic XML data

We use Dewey Encoding Scheme [26] to encode the probabilistic XML document. Generally speaking, we need to have two arrays for a node. One array stores the Dewey code (the same as indexing deterministic XML), and the other array stores the probabilities of the edges from the root node to the current node (special for probabilistic XML). By playing a little trick, we can encode edge probability into Dewey code and save some space cost. We illustrate the idea using Figure 5. 1.3.6.9 is the Dewey code of the node $x_4$, 0.9 $\rightarrow$ 1 $\rightarrow$ 0.7 are the probabilities on the path from the root to node $x_4$. To assist with the encoding, we add a dummy probability 1 before 0.9, and get the probability path as 1 $\rightarrow$ 0.9 $\rightarrow$ 1 $\rightarrow$ 0.7. By performing an addition operation, Dewey code and probability can be combined and stored together as 2 $(1+1) \rightarrow 3.9 (3+0.9) \rightarrow 7 (6+1) \rightarrow 9.7 (9+0.7)$. We name the code as pDewey code. For each field $y$ in the combined pDewey code, the corresponding Dewey code can be decoded as $\lceil y \rceil - 1$, and the probability can be decoded as $y + 1 - \lceil y \rceil$. The correctness can be guaranteed, because edge probabilities always belong to (0, 1]. Apparently, this encoding trick trades time for space, because on one hand we store the Dewey code and edge probabilities in one array so that we save half of the space cost, however, on the other hand, we need some computations to separate Dewey code and edge probabilities from the integrated encoding.

For each keyword, we store a list of nodes that directly contain that keyword using B+-tree. The nodes are identified by their pDewey codes. For each node, we also store the node types (ORD, IND, MUX) on the path from the root to the current node. This node type vector helps to perform different types of calculation for different distribution types. For simplicity, we use the traditional Dewey code and omit pDewey code decoding when we introduce the PrELCA algorithm in the next section.

4.2 PrELCA algorithm

According to the probabilistic ELCA semantics (1) defined in Section 2, a node with non-zero ELCA probability must contain all the keywords in some possible worlds. Therefore, all nodes in the keyword inverted lists and the ancestors of these nodes

\[
\begin{array}{c}
\text{Dewey code: 1} & X1 & \text{X4 Dewey code:} & X4 \text{ path probability from the root:} \\
& 0.9 & 1.3.6.9 & 1 \rightarrow 0.9 \rightarrow 1 \rightarrow 0.7 \\
\text{Dewey code: 1.3} & X2 & \text{Dewey code: 1.3.6} & X3 \\
& 1 & 0.7 & 2 (1+1) \rightarrow 3.9 (3+0.9) \rightarrow 7 (6+1) \rightarrow 9.7 (9+0.7) \\
\text{Dewey code: 1.3.6.9} & X4 & \text{X4 pDewey code:} & \\
\end{array}
\]

Figure 5 pDewey code.
constitute a candidate ELCA set. The idea of the PrELCA algorithm is to mimic a postorder traversal of the original p-document using only the inverted lists. This can be realized by maintaining a stack. We choose to mimic postorder traversal, because it has the feature that a parent node is always visited after all its children have been visited. This feature exactly fits the idea on how to compute ELCA probability conceptually in Section 3. By scanning all the inverted lists once, PrELCA algorithm can find all nodes with non-zero ELCA probabilities without generating possible worlds. Algorithm 1 gives the procedural steps. We first go through the steps, and then give a running example to illustrate the algorithm.

PrELCA algorithm takes keyword inverted lists as input, and outputs all probabilistic ELCA nodes with their ELCA probabilities. The memory cost is a stack. Each entry of the stack contains the following information: (1) a visited node \( v \), including the last number of \( v \)'s Dewey code (eg. 3 is recorded if 1.2.3 is the Dewey code of \( v \)), the type of the node \( v \) (i.e. ORD, IND or MUX); (2) an intermediate keyword distribution table of \( v \), denoted as \( disTbl[] \); (3) an intermediate ELCA probability

\[
\text{Algorithm 1 PrELCA algorithm}
\]

**Input:** inverted lists of all keywords, \( S \)

**Output:** a set of \( (r[], f) \) pairs \( R \), where \( r[] \) is a node (represented by its Dewey code), \( f \) is the ELCA probability of the node

1: result set \( R := \emptyset \);
2: stack := empty;
3: while not end of \( S \) do
4:   Read a new node \( v \) from \( S \) according to Dewey order, let array \( \nu[] \) record its Dewey code;
5:   \( p := \text{lcp}(\text{stack, } v) \); \{find the longest common prefix \( p \) such that \( \text{stack}[i].\text{node} = \nu[i], 1 \leq i \leq p \}\n6:   \text{while stack.size} > p \text{ do}
7:      let \( r[] \) be the Dewey code in the current stack;
8:      let \( f = \text{stack.top().elcaTbl}[11...1] \);
9:      add \( (r[], f) \) into the result set \( R \) if \( f \neq 0 \);
10:     \( \text{popEntry} = \text{stack.pop()} \);
11:    \( \text{merge popEntry.disTbl[]} \text{ into stack.top().disTbl[]} \);
12:   calculate a new \( \text{stack.top().elcaTbl}[] \) using the previous \( \text{stack.top().elcaTbl}[] \) and \( \text{popEntry.disTbl}[] \);
13: end while
14: for \( p < j \leq v.length \) do
15:    \( \text{disTbl}[] = \text{new disTable()} \);
16:   \( \text{elcaTbl}[] = \text{new elcaTable()} \);
17:   \( \text{newEntry} = (\text{node} := \nu[j]; \text{disTbl}[]; \text{elcaTbl}[]) \);
18:   \( \text{stack.push(newEntry)} \);
19: end for
20: end while
21: while stack is not empty do
22:     Repeat line 7 to line 13;
23: end while
table of \( v \), denoted as \( elcaTbl[] \). At the beginning, the result set and the stack are initialized as empty (lines 1 and 2). For each new node read from the inverted lists (lines 3–20), the algorithm pops up some nodes whose descendant nodes will not be seen in the future, output their ELCA probabilities (lines 6–13), and pushes some new nodes into the stack (lines 14–19). Line 5 is to calculate how many nodes should be popped from the stack by finding the longest common prefix between the stack and the Dewey code of the new node. Lines 7–9 is to output a result. After that, the top entry will be popped up (line 10), and its keyword distribution table will be merged into the new top entry (which records the parent node of the popped node) based on (6) and (8) at line 11, and its new top entry’s ELCA probability table will also be recalculated based on (13) at line 12. For each newly pushed node, its keyword distribution table \( disTbl[] \) will be initialized using (5) and (12) at lines 15 and 16 respectively. Line 17 constructs a stack entry and line 18 pushes the new entry into the stack. After we finish reading the inverted lists, the remaining nodes in the stack are popped and checked finally (lines 21–23).

In Figure 6, we show some snapshots for running PrELCA algorithm on the probabilistic XML tree in Figure 1b. The left and right arrows indicate push and pop operations respectively. At the beginning (step 1), the first keyword instance \( a_1 \) is read. All the ancestors of \( a_1 \) are pushed into the stack, with the corresponding \( disTbl[] \) and \( elcaTbl[] \) fields initialized. In step 2, \( a_2 \) is read according to the order of Dewey code. The longest common prefix between the stack and the Dewey code of \( a_2 \) is \( r.IND1.x_2 \). So \( a_1 \) is popped up, and \( x_2 \)'s \( disTbl[] \) and \( elcaTbl[] \) are updated into \((0, 1, 0, 0)\) and \((0, 1, 0, 0)\) by merging with \( a_1 \)'s \( disTbl[] \). Node \( a_1 \) is not a result, because the \( a_1 \)'s \( elcaTbl[11] \) is 0. Then, nodes \( IND2 \) and \( a_2 \) are pushed into the stack. In step 3, \( b_1 \) is read afterwards. Similar to step 2, \( a_2 \) is popped up with \( IND2 \)'s \( disTbl[] \) updated, and then \( b_1 \) is pushed into the stack. In step 4, we read a new node \( b_2 \) from the inverted lists. In the stack, node \( b_1 \) is first popped out of the stack. \( IND2 \)'s \( disTbl[] \) is updated into \((0.12, 0.18, 0.28, 0.42)\) by merging with \( b_1 \)'s \( disTbl[] \) \((0.3, 0, 0.7, 0)\) with \( IND2 \)'s current \( disTbl[] \) \((0.4, 0.6, 0, 0)\). Readers may feel free to verify the computation. Similarly, \( x_1 \)'s \( disTbl[] \) is updated as \((0.12, 0.18, 0.28, 0.42)\).

**Figure 6** Stack status for some steps of running PrELCA algorithm on the probabilistic XML tree in Figure 2b.
when IND2 is popped out. \(x_1\)'s \(elcaTbl[]\) is set as IND2's \(elcaTbl[]\), because IND2 is a single child distributional node of \(x_1\) and thus it does not screen keywords from contributing upwards. (Recall Case 1 in Section 3.2). When \(x_1\) is popped out, we find \(x_1\)'s \(elcaTbl[11]\) is non-zero. Therefore, \(x_1\) has local ELCA probability, \(Pr_{elca}^L(x_1) = 0.42\). The global ELCA probability for \(x_1\) can be obtained by multiplying 0.42 with the edge probabilities along the path from the root \(r\) to \(x_1\). In this example, the global ELCA probability \(Pr_{elca}^G(x_1) = 0.42 \times 0.8\). An interesting scene takes place when \(x_1\) is popped out of the stack, \(x_2\)'s \(disTbl[]\) is updated accordingly as \((0, 0.3, 0, 0.7)\) and \(x_2\)'s \(elcaTbl[]\) is updated as \((0, 0.72, 0, 0.28)\). For the first time during the process, \(x_2\)'s \(elcaTbl[]\) is updated into a different value from its \(disTbl[]\). The reason is that \(x_1\) has screened keyword \(a, b\) from contributing upwards when \(x_1\) itself has already contained both keywords. So the local probability that \(x_2\) contains both keywords, represented by \(x_2\)'s \(disTbl[11]\) is 0.7, but the local ELCA probability of \(x_2\), represented by \(x_2\)'s \(elcaTbl[11]\) is only 0.28. At last, \(b_2\) is pushed into the stack.

The following computation is similar, and thus we do not show the whole process with snapshots.

Note that Algorithm PrELCA has the same push/pop procedures as XRank [7], which is designed for computing ELCA's from deterministic XML documents. The differences between PrELCA and XRank is that PrELCA incorporates ELCA probability computations (related steps are lines 11–12) and uses additional data structures serving the probability computations (related steps are lines 8–9 and lines 15–17).

5 Experiments

In this section, we report the performance of the PrELCA algorithm in terms of effectiveness, time and space cost, and scalability. All experiments are done on a laptop with 2.27GHz Intel Pentium 4 CPU and 3GB memory. The operating system is Windows 7, and code is written in Java.

5.1 Datasets and queries

Two real life datasets, DBLP1 (148M) and Mondial2 (1.2M), and one synthetic benchmark dataset, XMark3 have been used. We also generate four test datasets with sizes 10M, 20M, 40M, 80M for XMark data. The three types of datasets are chosen due to the following typical features: DBLP is a large shallow dataset; Modial is a deep, complex, but small dataset; XMark is a balanced dataset, and users can define different depths and sizes to mimic various types of documents.

For each dataset, we generate a corresponding probabilistic XML tree, using the same method in [10]. To be specific, we traverse the original document in preorder, and for each visited node \(v\), we randomly generate some distributional nodes with “IND” or “MUX” types as children of \(v\). Then, for the original children of \(v\), we

---

1http://dblp.uni-trier.de/xml/
2http://www.dbis.informatik.uni-goettingen.de/Mondial/XML
3http://monetdb.cwi.nl/xml/
choose some of them to be the children of the new generated distributional nodes and assign random probability distributions to these children with the restriction that the probability sum under a MUX node is no greater than 1. For each dataset, the percentage of the IND and MUX nodes are controlled around 30% of the total nodes respectively. We also randomly select some terms and construct five keyword queries for different datasets, shown in Table 1.

In Section 5.2 and 5.3, we will compare PrELCA algorithm with a counterpart algorithm, PrStack [17]. We refer PrStack as PrSLCA for the sake of antithesis. Probabilistic SLCA semantics is similar to probabilistic ELCA semantics. The difference is that the probability of a node $v$ being a probabilistic SLCA on a p-document $T$ is the sum of probabilities of the node $v$ being an SLCA in all possible worlds produced by $T$, while the probability of a node $v$ being a probabilistic ELCA on a p-document $T$ is the sum of probabilities of the node $v$ being an ELCA in all possible worlds produced by $T$. PrSLCA is an algorithm to find probabilistic SLCA elements from a probabilistic XML document. In Section 5.2, we will compare search result confidence (probabilities) under the two semantics. In Section 5.3, we will report the run-time performance of both algorithms.

Note that in our experiments, most queries have 2–3 keywords, because we want to keep the results of the keyword queries meaningful. Our algorithm does not have problems to deal with more keywords. However, when facing more keywords (i.e. complicated keyword queries), we suggest to do a keyword correlation step first to reduce both the size of the queries and the size of the input keyword inverted lists. We can discover keyword correlations in advance by analyzing the data and use these keyword correlations to simplify complicated keyword queries. Where complicated queries cannot be simplified, it is likely that the result of these complicated queries have low quality, because a large XML fragment may be required to contain all the loosely-related keywords. To keep the paper focused, we consider only simple queries in our experiments.

5.2 Evaluation of probability

Table 2 shows a comparison of probabilistic ELCA results and probabilistic SLCA results when we run the queries over Mondial dataset and XMark 20MB dataset. For each query and dataset pair, we select top-10 results (with highest probabilities), and

| ID | Keyword query          | ID | Keyword query          |
|----|------------------------|----|------------------------|
| $X_1$ | United States, graduate | $X_2$ | United States, credit, ship |
| $X_3$ | Check, ship            | $X_4$ | Alexas, ship           |
| $X_5$ | Internationally, ship  |     |                        |
| $M_1$ | Muslim, multiparty     | $M_2$ | City, area             |
| $M_3$ | United States, Islands | $M_4$ | Government, area       |
| $M_5$ | 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          |     |                        |
We also count how many results are shared among the results returned by different semantics. For some queries, M2 and M5, ELCA results are almost the same as SLCA results (see the Overlap column), but in most cases, ELCA results and SLCA results are different. Query X5 on XMark even returns totally different results for the two semantics. For other queries, at least 20% results are shared by the two semantics. After examining the returning results, we find that, most of time, PrELCA algorithm will not miss high-ranked results returned by PrSLCA. The reason is that, in an ordinary document, SLCAs are also ELCAs, so probabilistic SLCAs are also probabilistic ELCAs. A node with high SLCA probability is likely to have ELCA probability.

One interesting feature is that, compared with SLCA results, ELCA results always have higher probabilities (except for some queries returning similar results, like M2, M5). For queries M1, M3, M4 on Mondial dataset, the average probability value of ELCA ranges from 0.42 to 0.55, while that of SLCA is about 0.08–0.23. On XMark dataset, we have a similar story, with average ELCA probability from 0.27 to 0.62 and average SLCA probability from 0.08–0.51. ELCA results also have higher Max and Min values. Since the probability reflects the likelihood that a node exists among all possible worlds as an ELCA or an SLCA, it is desirable that returned results have higher probability (or we say confidence). From this point of view, ELCA results are better that SLCA results, because they have higher existence probabilities. Moreover, the Max probabilities of ELCA results are usually high, above 0.5 in all query cases, but for some queries, such as M4, X5, the Max probabilities of SLCA results are below 0.2. If a user issue a threshold query asking results with probability higher than 0.4, there will be no result using SLCA semantics, but ELCA semantics.

| Queries@Mondial | Max    | Min    | Avg    | Overlap |
|-----------------|--------|--------|--------|---------|
| M1 ELCA         | 0.816  | 0.426  | 0.55   | 60%     |
| SLCA            | 0.703  | 0.072  | 0.23   |         |
| M2 ELCA         | 1.000  | 0.980  | 0.99   | 100%    |
| SLCA            | 1.000  | 0.980  | 0.99   |         |
| M3 ELCA         | 0.788  | 0.304  | 0.45   | 40%     |
| SLCA            | 0.582  | 0.073  | 0.13   |         |
| M4 ELCA         | 0.730  | 0.100  | 0.42   | 20%     |
| SLCA            | 0.180  | 0.014  | 0.08   |         |
| M5 ELCA         | 1.000  | 0.890  | 0.94   | 90%     |
| SLCA            | 1.000  | 0.840  | 0.90   |         |

| Queries@XMark  | Max    | Min    | Avg    | Overlap |
|-----------------|--------|--------|--------|---------|
| X1 ELCA         | 0.560  | 0.165  | 0.27   | 20%     |
| SLCA            | 0.209  | 0.054  | 0.15   |         |
| X2 ELCA         | 0.789  | 0.353  | 0.54   | 50%     |
| SLCA            | 0.697  | 0.153  | 0.22   |         |
| X3 ELCA         | 0.970  | 0.553  | 0.62   | 30%     |
| SLCA            | 0.750  | 0.370  | 0.51   |         |
| X4 ELCA         | 0.716  | 0.212  | 0.34   | 20%     |
| SLCA            | 0.236  | 0.014  | 0.13   |         |
| X5 ELCA         | 0.735  | 0.525  | 0.62   | 0%      |
| SLCA            | 0.163  | 0.044  | 0.08   |         |
still gives non-empty results. This could be a reason to use ELCA semantics to return keyword query results.

For the DBLP dataset, we have not listed the results due to paper space limitation, but it is not difficult to understand that probabilistic ELCA results and probabilistic SLCA results are very similar on the DBLP dataset, since it is a flat and shallow dataset.

![Graphs](image-url)

(a) Time cost on XMark20M  
(b) Memory cost on XMark20M  
(c) Time cost on Mondial  
(d) Memory cost on Mondial  
(e) Time cost on DBLP  
(f) Memory cost on DBLP

Figure 7 Time and memory cost on XMark, Mondial and DBLP datasets.
5.3 Evaluation of time cost and space cost

Figure 7 shows the time and space cost when we run the queries $X_1$-$X_5$ on XMak20M, $M_1$-$M_5$ on Mondial, and $D_1$-$D_5$ on DBLP. From Figure 7a, c, e, we can see that both algorithms PrELCA and PrSLCA are efficient. Although ELCA semantics is more complex than SLCA semantics, PrELCA algorithm has a similar performance as PrSLCA algorithm in terms of time cost. The reason may be that both PrELCA and PrSLCA algorithms are stack-based algorithms and access keyword inverted lists in a similar manner. PrELCA algorithm is slightly slower than PrSLCA in most cases, which is acceptable, because ELCA semantics is more complex and needs more computation. The gap is not large, reflecting that PrELCA algorithm is a competent algorithm if users would like to know probabilistic ELCAAs rather than probabilistic SLCAAs. From Figure 7b, d and f, we can see that PrELCA consumes more memory than PrSLCA. This is because besides the keyword distribution tables which are used in both algorithms, PrELCA has to maintain some other intermediate results to compute the final ELCA probabilities, such as the intermediate table mentioned in (12) and (13) in Section 3.2.

5.4 Evaluation of scalability

In this section, we use XMark dataset to test the scalability of the PrELCA algorithm. We test two queries $X_1$, $X_2$ on the XMark dataset ranging from 10M to 80M. Figure 8a shows that the time cost of both queries is going up moderately when the size of the dataset increases. Figure 8b shows that space cost has a similar trend as the time cost, when document size is increasing. The experiment shows that, for various keyword queries, PrELCA algorithm scales well on different documents, although different queries may consume different memories and run for different time, due to different lengths of the inverted lists.

6 Related work

There are two streams of works related to our work: probabilistic XML data management and keyword search on ordinary XML documents.
Uncertain data management draws the attention of database research community recently, including both structured and semi-structured data. In the XML context, the first probabilistic XML model is ProTDB [22]. In ProTDB, two new types of nodes are added into a plain XML document. IND describes independent children and MUX describes mutually-exclusive children. Correspondingly, to answer a twig query on a probabilistic XML document is to find a set of results matching the twig pattern but the results will have existence probabilities. Hung et al. [8] modeled probabilistic XML documents as directed acyclic graphs, explicitly specifying probability distribution over child nodes. In [9], probabilities are defined as intervals, not points. Keulen et al. [27] introduced how to use probabilistic XML in data integration. Their model is a simple model, only considering mutually-exclusive sub-elements. Abiteboul and Senellart [2] proposed a “fuzzy trees” model, where the existence of the nodes in the probabilistic XML document is defined by conjunctive events. They also gave a full complexity analysis of querying and updating on the “fuzzy trees” in [24]. In [1], Abiteboul et al. summarized all the probabilistic XML models in one framework, and studied the expressiveness and translations between different models. ProTDB is represented as PrXML\textsuperscript{\{ind, mux\}} using their framework. Cohen et al. [5] 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. On querying probabilistic XML data, twig query evaluation without index (node lists) and with index are considered in [12] and [23] respectively. Chang et al. [4] addressed a more complex situation where result weight is also considered. The most closest work to ours is [17]. Compared to SLCA semantics in [17], we studied a more complex but reasonable semantics, ELCA semantics.

Keyword search on ordinary XML documents has been extensively investigated in the past few years. Keyword search results are usually considered as fragments from the XML document. Most works use LCA (lowest common ancestor) semantics to find a set of fragments. Each fragment contains all the keywords. These semantics include ELCA [7, 29, 30], SLCA [25, 28], MLCA [18] and Interconnection Relationship [6]. Other LCA-based query result semantics rely more or less on SLCA or ELCA by either imposing further conditions on the LCA nodes [15] or refining the subtrees rooted at the LCA nodes [14, 19, 20]. The works [3] and [16] utilize statistics of the underlying XML data to identify possible query results. More interesting topics related to keyword search on XML can be found in the survey [21]. All the above works consider deterministic XML trees. Algorithms on deterministic documents cannot be directly used on probabilistic documents, because, on probabilistic XML documents, a node may or may not appear, as a result, a node may be an LCA in one possible world, but not in another. How to compute the LCA probability for a node also comes along as a challenge.

7 Conclusions

In this paper, we have studied keyword search on probabilistic XML documents. The probabilistic XML data follows a popular probabilistic XML model, PrXML\textsuperscript{\{ind, mux\}}. We have defined probabilistic ELCA semantics for a keyword query on a probabilistic XML document in terms of possible world semantics. A stacked-based algorithm,
PrELCA, has been proposed to find probabilistic ELCAs and their ELCA probabilities without generating possible worlds. We have conducted extensive experiments to test the performance of the PrELCA algorithm in terms of probability, time and space cost, and scalability. We have compared the results with a previous SLCA based algorithm. The experiments have shown that ELCA semantics gives better keyword queries results with only slight performance sacrifice.

Acknowledgements This work is supported by the grant of Australian Research Council Discovery Project No. DP110102407 and the grant of Research Grants Council of the Hong Kong SAR, China, No. 419109. We would like to thank anonymous reviewers for their helpful comments.

References

1. Abiteboul, S., Kimelfeld, B., Sagiv, Y., Senellart, P.: On the expressiveness of probabilistic xml models. VLDB J. 18(5), 1041–1064 (2009)
2. Abiteboul, S., Senellart, P.: Querying and updating probabilistic information in xml. In: EDBT, pp. 1059–1068 (2006)
3. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective xml keyword search with relevance oriented ranking. In: ICDE, pp. 517–528 (2009)
4. Chang, L., Yu, J.X., Qin, L.: Query ranking in probabilistic xml data. In: EDBT, pp. 156–167 (2009)
5. Cohen, S., Kimelfeld, B., Sagiv, Y.: Incorporating constraints in probabilistic xml. ACM Trans. Database Syst. 34(3), (2009)
6. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: a semantic search engine for XML. In: VLDB, pp. 45–56 (2003)
7. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD Conference, pp. 16–27 (2003)
8. Hung, E., Getoor, L., Subrahmanian, V.S.: Pxml: a probabilistic semistructured data model and algebra. In: ICDE, pp. 467–478 (2003)
9. Hung, E., Getoor, L., Subrahmanian, V.S.: Probabilistic interval xml. ACM Trans. Comput. Log. 8(4), (2007)
10. Kimelfeld, B., Kosharovsky, Y., Sagiv, Y.: Query efficiency in probabilistic xml models. In: SIGMOD Conference, pp. 701–714 (2008)
11. Kimelfeld, B., Kosharovsky, Y., Sagiv, Y.: Query evaluation over probabilistic xml. VLDB J. 18(5), 1117–1140 (2009)
12. Kimelfeld, B., Sagiv, Y.: Matching twigs in probabilistic xml. In: VLDB, pp. 27–38 (2007)
13. Kimelfeld, B., Senellart, P.: Probabilistic XML: models and complexity. http://pierre.senellart.com/publications/kimelfeld2012probabilistic.pdf (2011, preprint)
14. Kong, L., Gilleron, R., Lemay, A.: Retrieving meaningful relaxed tightest fragments for xml keyword search. In: EDBT, pp. 815–826 (2009)
15. Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable lcas over xml documents. In: CIKM, pp. 31–40 (2007)
16. Li, J., Liu, C., Zhou, R., Wang, W.: Suggestion of promising result types for xml keyword search. In: EDBT, pp. 561–572 (2010)
17. Li, J., Liu, C., Zhou, R., Wang, W.: Top-k keyword search over probabilistic xml data. In: ICDE, pp. 673–684 (2011)
18. Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In: VLDB, pp. 72–83 (2004)
19. Liu, Z., Chen, Y.: Identifying meaningful return information for xml keyword search. In: SIGMOD Conference, pp. 329–340 (2007)
20. Liu, Z., Chen, Y.: Reasoning and identifying relevant matches for xml keyword search. PVLDB 1(1), 921–932 (2008)
21. Liu, Z., Chen, Y.: Processing keyword search on xml: a survey. World Wide Web 14(5–6), 671–707 (2011)
22. Nierman, A., Jagadish, H.V.: ProTDB: probabilistic data in xml. In: VLDB, pp. 646–657 (2002)
23. Ning, B., Liu, C., Yu, J.X., Wang, G., Li, J.: Matching top-k answers of twig patterns in probabilistic xml. In: DASFAA (1), pp. 125–139 (2010)
24. Senellart, P., Abiteboul, S.: On the complexity of managing probabilistic xml data. In: PODS, pp. 283–292 (2007)
25. Sun, C., Chan, C.Y., Goenka, A.K.: Multiway slca-based keyword search in xml data. In: WWW, pp. 1043–1052 (2007)
26. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered xml using a relational database system. In: SIGMOD Conference, pp. 204–215 (2002)
27. van Keulen, M., de Keijzer, A., Alink, W.: A probabilistic xml approach to data integration. In: ICDE, pp. 459–470 (2005)
28. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD Conference, pp. 537–538 (2005)
29. Xu, Y., Papakonstantinou, Y.: Efficient lca based keyword search in xml data. In: EDBT, pp. 535–546 (2008)
30. Zhou, R., Liu, C., Li, J.: Fast elca computation for keyword queries on xml data. In: EDBT, pp. 549–560 (2010)