A Hybrid Algorithm for Finding Top-*k* Twig Answers in Probabilistic XML

Bo Ning[1] and Chengfei Liu[2]

[1] Dalian Maritime University, Dalian, China
`ningbo@dlmu.edu.cn`
[2] Swinburne University of Technology, Melbourne, Australia
`CLiu@groupwise.swin.edu.au`

Abstract. Uncertainty is inherently ubiquitous in data of real applications, and those uncertain data can be naturally represented by the XML. Matching twig pattern against XML data is a core problem, and on the background of probabilistic XML, each twig answer has a probabilistic value because of the uncertainty of data. The twig answers that have small probabilistic values are useless to the users, and the users only want to get the answers with the largest k probabilistic values. In this paper, we address the problem of finding twig answers with top-k probabilistic values against probabilistic XML documents directly. To cope with this problem, we propose a hybrid algorithm which takes both the probability value constraint and structural relationship constraint into account. The main idea of the algorithm is that the element with larger path probability value will more likely contribute to the twig answers with larger twig probability values, and at the same time lots of useless answers that do not satisfy the structural constraint can be filtered. Therefore the proposed algorithm can avoid lots of intermediate results, and find the top-k answers quickly. Experiments have been conducted to study the performance of the algorithm.

1 Introduction

Nowadays, uncertainty is inherently ubiquitous in data of real applications. For instance, in sensor applications, sensors produce uncertain data since readings of sensors are inherently imprecise. In scientific research, error-prone experimental machinery, polluted samples, and simple human error bring the uncertainty to experimental data. Therefore uncertain data management is becoming a critical issue. The current relational database technologies can not deal with this problem very well, because to store imprecise information in structured data format can lead to high complexity of space and processing time. While the XML data is a natural representation of uncertain data due to its flexible characteristics. XML has hierarchical structure, therefore the probability values can be assigned to elements and subtrees, dependency and independency of elements can be expressed. In addition, XML supports incomplete information gracefully. The data models for representing uncertainty in XML have been studied in [1–6].

As to the query processing on probabilistic XML, the queries on the probabilistic XML are often in the form of twig patterns. Compared with the query on ordinary XML, the matched answers are associated with the probabilistic values when querying probabilistic data. Therefore the answers as well as the probability values need to be returned. Many kinds of twig queries with different semantics were proposed, and their evaluations were studied in [7]. It is obvious that the answers with small probabilistic values are useless to users who submit the queries, and it makes sense to only return the twig answers with top-k probabilistic values. There are also some other works on querying uncertain data. On uncertain relational data, matching the twig answers with probability values above a threshold was investigated in [8], and query ranking was studied in [9, 11, 12]. The paper [10] studied the query ranking in probabilistic XML by possible world model, and a dynamic programming approach was deployed that extends the approach in [9] to deal with the containment relationships in probabilistic XML, and rank the results by the interplay between score and uncertainty. Those works are based on the p-documents generated from probabilistic XML or the relational data model in which the possible worlds are stored.

It is more flexible if the twig answers with top-k probabilistic values can be matched against the probabilistic XML directly. The algorithm *ProTJFast* and *PTopKTwig*[13] belong to this catalog. In those algorithms, by the use of a novel encoding scheme and the effective use of lower bounds, elements or paths with small probabilities can be filtered. Matching a twig query against ordinary XML document only needs the answer to satisfy the structural relationships constraint, while finding top-k twig answers against probabilistic XML also needs the answers to satisfy the constraint that the probabilistic values of twig answers are largest k ones. The algorithm *ProTJFast* uses element streams ordered by document order (pre-order) as input, and the process of algorithm follows the document order, so that the constraint of probabilistic values are not fed as soon as possible. The algorithm *PTopKTwig* is based on the element streams ordered by path probabilistic value, and do not consider about the structural constraint too much, so that, to satisfy the structural constraint, there are lots of times of detection whether the elements of leaf nodes in query can be matched to be an twig answer. Although the use of enhanced lower bound makes algorithm *PTopKTwig* efficient, there are still lots of useless path answers are joined to the candidate twig answers that are not the final top-k answers.

In this paper, we address the problem of efficiently finding top-k twig answers against probabilistic XML directly too. Our algorithm takes both of the structural constraint and probabilistic value constraint into account, and can find the k twig answers which satisfy the structural relationships and their probabilistic values are largest as earlier as possible. In our algorithm, the intermediate path answers which do not satisfy the structural constraint and probabilistic value constraint can be filtered rapidly. Also we improve the encoding scheme that makes the process of calculating the probabilistic values more efficiently.

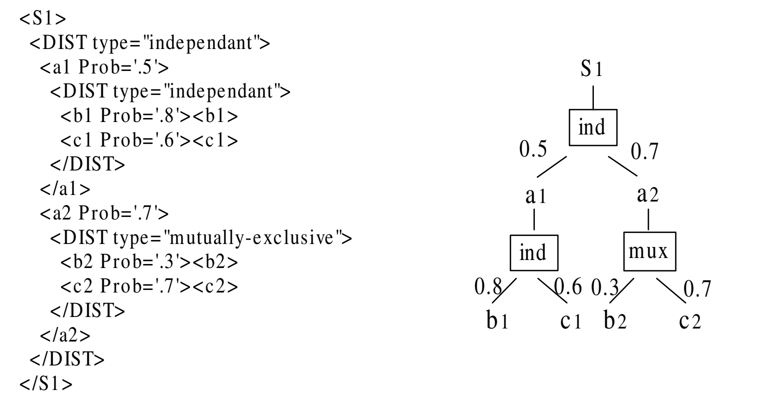
The rest of this paper is organized as follows. Section 2 introduce the background and relate work including the data model twig answers and encoding scheme of probabilistic XML.In Section 3, we improve the encoding scheme by redesign the float vector. In Section 4, we present a hybrid algorithm *HyTopK-Twig* for matching twig answers with top-k probabilities. Section 5 shows our experimental results. Conclusions are included in Section 6.

2 Preliminaries

2.1 Probabilistic XML Model

Nierman et al. proposed the Probabilistic Tree Data Base(ProTDB) [4] to manage uncertain data represented in XML. Actually it belongs to the catalog of PrXML$^{\{\text{ind, mux}\}}$ model [6],in which the independent distribution and mutually-exclusive distribution are considered.

A probabilistic XML document T_P defines a probability distribution over an XML tree $T(V, E)$ and it can be regarded as a weighted XML tree $T_P(V_P,E_P)$. In T_P, $V_P = V_D \cup V$, where V is a set of ordinary elements that appear in T, and V_D is a set of distribution nodes, including independent nodes and mutually-exclusive nodes (*ind* and *mux* for short). An ordinary element, $u \in V_P$, may have different types of distribution nodes as its child elements in T_P that specify the probability distributions over its child elements in T. E_P is a set of edges, and an edge which starts from a distribution node can be associated a positive probability value as weight.



```
<S1>
 <DIST type="independant">
  <a1 Prob='.5'>
   <DIST type="independant">
    <b1 Prob='.8'><b1>
    <c1 Prob='.6'><c1>
   </DIST>
  </a1>
  <a2 Prob='.7'>
   <DIST type="mutually-exclusive">
    <b2 Prob='.3'><b2>
    <c2 Prob='.7'><c2>
   </DIST>
  </a2>
 </DIST>
</S1>
```

(a) Fragment of probabilistic XML (b) Tree model of probabilistic XML

Fig. 1. Example of probabilistic XML

In Figure 1, (a) is the fragment of probabilistic XML document. By using the tag *DIST*, *Prob* and *VAL*, the XML has the ability to express the probabilistic distributions and the probabilistic values of elements. Figure (b) is the tree model for the probabilistic XML, which contains *ind* and *mux* nodes. The element a_1

has an *ind* node as its child, which specifies that its twig child nodes, b_1 and c_1 are independent. The probabilities of having b_1 and c_1 are 0.8 and 0.6, as indicated in the incoming edges to b_1 and c_1 respectively. The element a_2 has a *mux* node as its child, which specifies that b_2 and c_2 cannot appear as a_2's child at the same time. Because of the mutually-exclusive distribution, the sum of probability values of b_2 and c_2 cannot be larger than one.

2.2 Twig Query and Answers

A twig query q is an XPath query with predicates, and it can be modeled as a small tree $T_q(V_q, E_q)$, where V_q is a set of nodes representing types(tag name) and E_q is a set of edges. There are two kinds of edges in E_q including parent-child edge (PC for short) and ancestor-descendant edges (AD for short). Usually, AD edge corresponds to the descendant axis in XPath, and PC edge corresponds to the child axis. The answer of a twig query is a set of tuples, in which there are elements from the probabilistic XML, and those elements match the nodes in q and satisfy all the structural relationships specified in q.

However, to find the answers of a twig query q against a probabilistic XML document, not only the structural relationships specified in q have to be satisfied, but also the types of distribution nodes and weights of edges from distribution nodes have to be considered as well. Actually, a *mux* distribution node u_{mux} can be regarded as constraint to restrict two elements under different child elements of u_{mux} so that they do not contribute to the same result. In contrast, an *ind* distribution node does not affect the existence of an answer, but determine the probability value of the answer.

Given an answer expressed by a tuple $t = (e_1, e_2, ..., e_n)$, there exist a subtree $T_s(V_s, E_s)$ of T_P, which contains all those elements. The probability of t can be computed by the following equation.

$$prob(t) = prob(T_s) = \Pi_{e_i \in E_s} prob(e_i);$$

2.3 Encoding Scheme

For encoding an ordinary XML, the encoding scheme should support the structural relationships and keep the document order, because matching a twig query against ordinary XML document only needs the answer to satisfy the structural relationships constraint, while the formal encoding scheme can not meet the requirements at the background of probabilistic XML, as there are new characteristics of probabilistic XML including the distribution nodes and the probabilistic values. Therefore encoding scheme for probabilistic XML should contains the information of probabilistic values and provides the ability for matching twig answers that satisfy the constraint of probabilistic values.

Region-based encoding[14, 15] and prefix encoding are two kinds of encoding schemes for ordinary XML documents. Both encoding schemes support the structural relationships and keep the document order, and these two requirements are essential for evaluating queries against ordinary XML documents. As to the aspect of encoding elements in probabilistic XML, a new requirement needs to be fed, that is how to record the probability values of elements on different levels. Depending on different kinds of processing, the probability value of the current element which is under a distribution node in PXML (node-prob for short) needs to be record, and so does the probability value of the path from the root to the current element (path-prob for short). Ning et al.[13] conclude that both node-prob and path-prob in twig pattern matching against PXML are needed, and property for supporting ancestor vision and ancestor probability vision are also needed. The prefix encoding scheme naturally have the properties above, therefore it is better to encode PXML elements by a prefix encoding scheme.

For efficiently processing twig matching against ordinary XML, Lu et al. proposed a prefix encoding scheme named extended Dewey [16]. Extended Dewey is a kind of Dewey encoding, which use the modulus operation to create a mapping from an integer to an element name, so that given a sequence of integers, it can be converted into the sequence of element names. This characteristic make extended Dewey encoding have the tag name vision of ancestor, that makes the evaluation of twig join efficiently.

For the purpose of supporting twig pattern matching against probabilistic XML, Ning et al.[13] extend Lu's encoding scheme[16] by adding the properties of the *probability vision* and the *ancestor probability vision*, and propose a new encoding scheme called *PEDewey*. Compared with Extended Dewey, *PEDewey* takes the distribution node into account and assigns a float vector to each element, which records the probabilistic value information.

3 Improvement of the *PEDewey* Encoding

In this paper, we just improve the float vector part of *PEDewey* for the efficient calculation of twig answers' probabilistic values.

In *PEDewey*, an additional float vector is assigned to each element. The length of the vector is equal to that of a normal Dewey encoding, and each component holds the node probability value of the element. Given the vector v and each component $node_i$ in v, the path probability value can be calculated by the following equation:

$$prob(path) = \mathit{\Pi}_{node_i \in v} prob(node_i);$$

We can see from the equation that there are lots of multiplication operations for calculating a path probability value, and it is not efficient, so we improve the float vector by recording the natural logarithm of probability value in each component. After that, the path probability value can be calculated by equation:

$$prob_{ln}(path) = \mathit{\Sigma}_{node_i \in v} prob_{ln}(node_i);$$

During the processing of finding top-k twig answers, the probability values are in the form of natural logarithm, and when the final answers are found, we calculate the probability value by the equation $prob(t) = e^{prob_{ln}(t)}$. Notice that the components for elements of ordinary, *ind* and *mux* are all assigned to 0 which is the natural logarithm of 1.

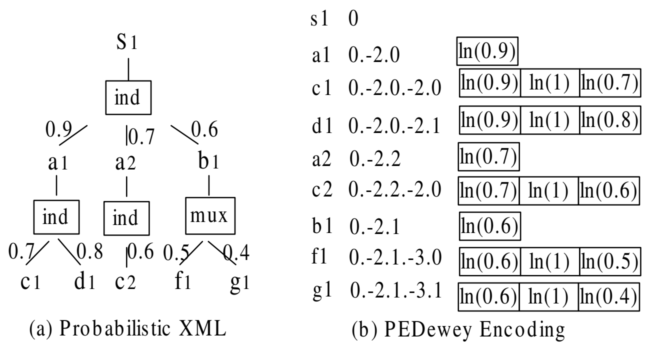

S1
ind
0.9
0.7
0.6
a1
a2
b1
ind
ind
mux
0.7
0.8
0.6
0.5
0.4
c1
d1
c2
f1
g1
(a) Probabilistic XML
s1 0
a1 0.-2.0 ln(0.9)
c1 0.-2.0.-2.0 ln(0.9) ln(1) ln(0.7)
d1 0.-2.0.-2.1 ln(0.9) ln(1) ln(0.8)
a2 0.-2.2 ln(0.7)
c2 0.-2.2.-2.0 ln(0.7) ln(1) ln(0.6)
b1 0.-2.1 ln(0.6)
f1 0.-2.1.-3.0 ln(0.6) ln(1) ln(0.5)
g1 0.-2.1.-3.1 ln(0.6) ln(1) ln(0.4)
(b) PEDewey Encoding

Fig. 2. Example of *PEDewey* encoding

We redefine the operations on the float vector. (1) Given element e, function $pathProb_{ln}(e)$ returns the path-prob of element e in natural logarithm form, which is calculated by adding the node-prob values of all ancestors of e in the float vector (2) Given element e and its ancestor e_a, function $ancPathProb_{ln}(e, e_a)$ returns the path-prob of e_a in natural logarithm form by adding those components from the root to e_a in e's float vector. (3) Given element e and its ancestor e_a, function $leafPathProb_{ln}(e, e_a)$ returns the path-prob of the path from e_a to e in form of natural logarithm by adding those components from e_a to e in e's float vector. (4) Given elements e_i and e_j, function $twigProb_{ln}(e_i, e_j)$, returns the probability of the twig whose leaves are e_i and e_j. Assume the e_i and e_j have common prefix e_c, and the probability of twig answer containing e_i and e_j is:

$$twigProb_{ln}(e_i, e_j) = pathProb_{ln}(e_i) + pathProb_{ln}(e_j) - ancPathProb_{ln}(e_i, e_c);$$

For the PXML shown in Figure 2(a), the encodings of its elements are shown in Figure 2(b). For example, $pathProb_{ln}(c_1)$=-0.462 (0+(-0.10536)+0+(-0.35667)), $ancPathProb_{ln}(c_1, a_1)$= -0.10536 (0+(-0.10536)+0), $leafPathProb_{ln}$ (c_1, a_1) = -0.35667 (0+(-0.35667)) and $twigProb_{ln}(c_1, d_1) = pathProb_{ln}(c_1) + pathProb_{ln}(d_1)$ - $ancPathProb_{ln}(c_1, a_1)$ = -0.6852, where a_1 is the common prefix of c_1 and d_1. At last we can get the probability value of this twig answer is 0.504 ($e^{-0.6852}$).

4 *HyTopKTwig*: a Hybrid Algorithm

In this part, we will propose an algorithm for finding top-k twig answers against the probabilistic XML directly. Firstly, we analyze the characteristics of the problem that finding top-k twig answers, then we propose the algorithm. At last, we analyze the correctness of our algorithm.

4.1 Analysis of the problem

When matching twig pattern against the ordinary XML documents, the only consideration is the structural relationships of query. That means the answers only need to satisfy the structural relationships of twig pattern. However the problem we are going to deal with is to find not only the answers that match the twig pattern, but also the k answers that have the largest probability values among all the twig answers. Therefore how to find top-k answers quickly without large amount of useless intermediate results is the challenge of the problem.

Most of the algorithms for matching twig pattern against the ordinary XML use elements streams ordered by the document order as the input data. Although those algorithms can be easily adjusted to solve the top-k answers problem against probabilistic XML, the efficiency is very low. That is because all the twig answers need to be found out no matter how small their probabilistic values are. Many useless intermediate results are computed, and the algorithm *ProTJFast* is in this case. The algorithm *ProTJFast* uses the elements streams ordered by document order, and the document order is good for matching the structural relationships, and makes the twig matching algorithm holistic. However, at the background of our problem, the document order limits the efficiency.

Intuitively the element with larger path probability value will more likely contribute to the twig answers with larger twig probability values. Based on this idea, algorithm using the elements streams ordered by probability values is proposed, for example the algorithm *PTopKTwig*, to deal with the top-k matching of twig queries against probabilistic XML. The algorithm *PTopKTwig* mainly takes the probability value order into account, and ignores the characteristics of the structural relationships constraints. It needs to compare whether the two leaf elements of leaf nodes in query can be joined to contribute to a final answers lots of times. Although the use of enhanced lower bound makes algorithm *PTopKTwig* efficient, without the documents order, the merge-joins can not be performed, that leads to the low efficiency because lots of comparisons can not be avoided.

From above we can see that there are structural relationships constraint and largest probabilistic values constraint in our problem, and it is a tradeoff between finding top-k probability values and satisfying the structural relationships. So we intend to design a hybrid algorithm which takes both constraints into account, and in our algorithm, the intermediate path answers which do not satisfy the structural constraint and probabilistic value constraint can be filtered rapidly.

4.2 Data Structures and Notations

Firstly we give the definition of skeleton pattern, which is the key point to balance the tradeoff between finding top-k probability values and satisfying the structural relationships.

Definition 1. *The skeleton pattern s is a subtree of twig pattern q, and it can be got by deleting all the subtrees of twig pattern which do not contain any branching node.*

For example, the skeleton pattern of A[//E]//B[//D]//C is A//B.

In our algorithm, we associate each leaf node f in a twig pattern q with a stream T_f, which contains encoding of all elements that match the leaf node f. The elements in the stream are sorted by their path-prob values. It is very fast to sort those elements by using the float vector in our encodings. Notice that in our encoding scheme, the component of float vector is in the form of natural logarithm, so the order of those natural logarithm floating numbers should be ascending, so that the order of real probability values is descending. We maintain *cursorList*, a list pointing to the head elements of all leaf node streams. Using the function cursor(f), we can get the position of the head element in T_f.

A list L_c for keeping top-k candidates is allocated for q. A set S_b is associated with skeleton pattern of query q. Each element cached in S_b are the skeleton part of the candidate answers. Initially set S_b is empty. For each element stream, we assign a signature list. The signature for a element represent whether the current element is a descendant of any skeleton result in set S_b. Initially, all the signatures are zero.

4.3 Algorithm *HyTopKTwig*

There are three phases in main algorithm of *HyTopKTwig*(Algorithm 1). The first phase (Lines 2-6) is to find the initial answers so that we can have k skeleton results in S_b. In the second phase(Lines 7-10), we use the signature list to identify the descendants of skeleton results in set S_b in respective element streams of leaf node in query q. The function of signature list is filtering the elements that can not contribute to the final answers. So in third phase(Line 11), we run algorithm *ProTJFast* against the document ordered element streams where useless elements have been filtered in phase two.

In the first phase, we intend to find the answers whose probability values are as large as possible, so we use the element streams ordered by probability value. The processing of this phase is just like that of algorithm *PTopKTwig*. Algorithm 1 firstly proceeds in the probability order of all the leaf nodes in query q, by calling the function *getNextP()*. This function returns the tag name of the leaf node stream which has the biggest probability value in its head element among all leaf node streams. So that, each processed element will not be processed repetitively. After function *getNextP()* returns a tag q_{act}, we may find the twig answers which the head element in stream $T_{q_{act}}$ contributes to, by invoking function *matchTwig()*. In function *matchTwig()*, the twig answers containing the $e_{q_{act}}$ can all be found. During the process of finding other elements that contribute to the twig answers with $e_{q_{act}}$, there is no duplicated computation of comparing the prefixes, due to the order of probability values and the use of *cursorList*. The *cursorList* records the head elements in respective streams which is next to be processed. The elements before the head elements have

Algorithm 1: *$HyTopkTwig(q)$*

```
Data: Twig query $q$, and streams $T_f$ of the leaf node in $q$.
Result: The matchings of twig pattern $q$ with top-$k$ probabilities.
1  begin
2      while $Sizeof(S_b) < k \wedge \neg\ end(q)$ do
3          $q_{act}$=$getNextP(q)$;
4          $tempTwigResults$=$matchTwig(q_{act}, q)$;
5          $addSkeletonResults(S_b, tempTwigResults)$;
6          $advanceCursor(cursor(q_{act}))$;
7      foreach $q_i \in leafNodes(q)$ do
8          foreach $e_j \in T_{q_i}$ do
9              if $\exists s_k \in S_b$, $s_k$ is the prefix of $e_j$ then
10                 $e_j.signature = 1$;
11     $ProTJFastBySignature(q)$;
12 end
13 Function $end(q)$
14 begin
15     Return $\forall f \in leafNodes(q) \rightarrow eof(T_f)$;
16 end
17 Function $getNextP(n)$
18 begin
19     foreach $q_i \in leafNodes(q)$ do
20         $e_i = get(T_{q_i})$;
21     $max = maxarg_i(e_i)$;
22     return $n_{max}$
23 end
24 Function $ProTJFastBySignature(q)$
25 begin
26     Sort the elements whose signature equals to 1 in respective element streams
       by document order.
27     By using algorithm ProTJFast, output the k answers with largest
       probability values.
28 end
```

been compared with elements in other streams, and the twig answers that these elements might contribute to have been considered. Therefore we only compare $e_{q_{act}}$ with the elements after the head elements in the related streams (Lines 3-4 in Algorithm 2). Once a twig answer are found, we add the skeleton result of this answer to the set S_b, until the size of the S_b equals to k (Line 5 in Algorithm 1). So we can see that the task of phase 1 is to find k skeleton results.

In the second phase of Algorithm 1(Lines 7-10), we update the signature lists of element streams by assigning the signature of element below any skeleton result in set S_b to 1. So that, by the signature lists, we can get a subtree of original probabilistic XML document. We can prove that the final top-k twig answers against the original XML can be found in the subtree.

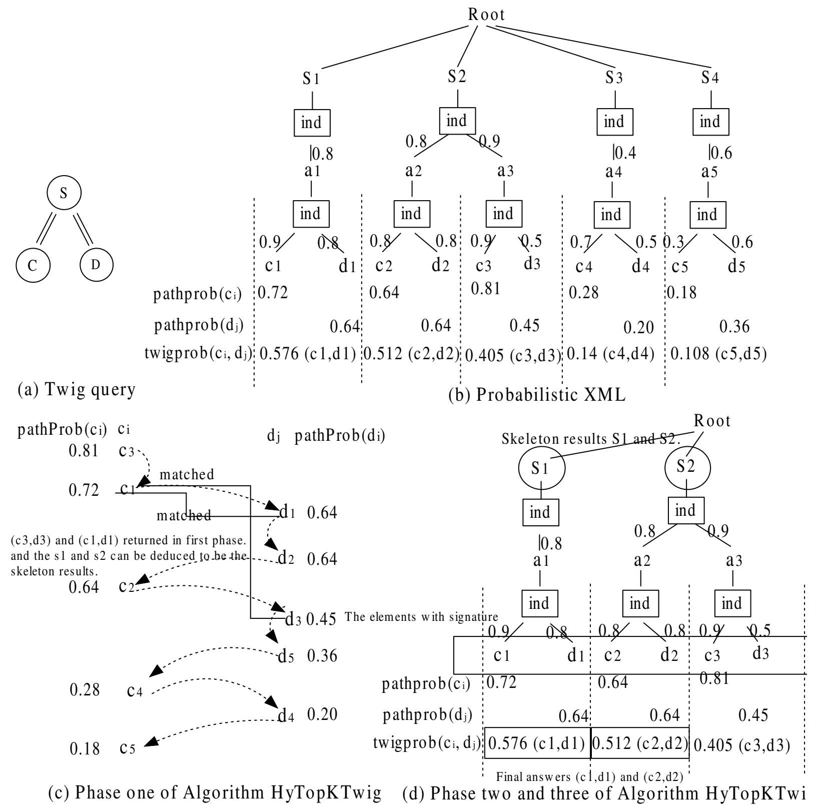
In the last phase of Algorithm 1(Line 11), we firstly sort the elements whose signatures equal to 1 in respective element streams by document order. And then we can perform the algorithm *ProTJFast* to output the k answers with largest probability values. Notice that if the k of top-k query is small, we can use algorithm *TJFast* at the third phase directly, because the number of elements under the k skeleton results is also small, therefore there is no need to use the more complex algorithm *ProTJFast*.

Algorithm 2: $matchTwig(q_{act}, q)$

```
1 begin
2     for any tags pair [$T_{q_a}$,$T_{q_b}$] ($q_a$,$q_a \in leafNodes(q) \wedge q_a$, $q_b \neq q_{act}$) do
3         Advance head element in $T_{q_a}$ to the position of cursor($q_a$);
4         Advance head element in $T_{q_b}$ to the position of cursor($q_b$);
5         while $\neg$ end(q) do
6             if elements $e_{q_a}$,$e_{q_a}$ match the common path pattern with $e_{q_{act}}$ in
              query q, and the common prefix of $e_{q_a}$,$e_{q_a}$ match the common path
              pattern which is from the root to the branching node $q_{bran}$ of $q_a$ and
              $q_b$ in query q, and the common prefix is not a element of mux node.
              then
7                 add $e_{q_a}$,$e_{q_a}$ to the set of intermediate results.
8     return twig answers from the intermediate set.
9 end
```

For the twig query (a) in Figure 3: S[//C]//D against probabilistic XML (b), assume that the answers for top-2 probabilities are required. In the first phase of Algorithm 1, streams T_C and T_D are scanned, and the elements in streams are sorted by path-prob values shown in Figure 3 (c). The processing order of elements in streams are marked by dotted arrow line in Figure 3(c), which is obtained by invoking the $getNextP()$ function. Firstly, $getNextP()$ returns c_3 because its probability value is largest among the elements in respective streams, and c_3 start to join the element in T_D, and find a match (c_3, d_3), next the Algorithm 1 add the skeleton result s_2 of (c_3, d_3) to set S_b. Then $getNextP()$

returns the element c_1, and an answer (c_1, d_1) is found, also the skeleton result s_1 is added to set s_b. At this moment, the size of set s_b equals to the value of k, so the first phase ends. In the second phase, Algorithm 1 marks the signatures of elements that are the descendants of skeleton results in set S_b, so the signatures of elements c_1, d_1, c_2, d_2, c_3 and d_3 are updated. In the last phase, the elements of respective element streams are ordered by document order (In Figure (d)), and then the Algorithm *ProTJFast* can be performed on them. So the final top-2 twig answers is (c_1,d_1) with probability value 0.576 and (c_2,d_2) with probability value 0.512. Notice that, the temporal results (c_3,d_3) in phase one is not the final answer, by the skeleton result s_2 generated from temporal results (c_3,d_3), we can bring the elements c_2 and d_2 to the final phase, and finally they can be merge-joined to be a final answer.



Root
S1
S2
S3
S4
ind
a1
a2
a3
a4
a5
c1 d1 c2 d2 c3 d3 c4 d4 c5 d5
pathprob(ci) 0.72 0.64 0.81 0.28 0.18
pathprob(dj) 0.64 0.64 0.45 0.20 0.36
twigprob(ci, dj) 0.576 (c1,d1) 0.512 (c2,d2) 0.405 (c3,d3) 0.14 (c4,d4) 0.108 (c5,d5)
(a) Twig query
(b) Probabilistic XML
pathProb(ci) ci
dj pathProb(di)
matched
(c3,d3) and (c1,d1) returned in first phase. and the s1 and s2 can be deduced to be the skeleton results.
The elements with signature
Skeleton results S1 and S2.
Final answers (c1,d1) and (c2,d2)
(c) Phase one of Algorithm HyTopKTwig
(d) Phase two and three of Algorithm HyTopKTwi

Fig. 3. Example of HyTopKTwig

4.4 Analysis of Algorithm

We can see that in the first phase of Algorithm *HyTopKTwig*, the element streams are ordered by probability value, so that we can find the answers whose probability value are as larger as possible, while in the last phase, we use Algorithm *ProTJFast* against the element streams ordered by document order, therefore Algorithm *HyTopKTwig* is a hybrid algorithm that takes both probability value constraint and structure constraint into account.

However, are those temporal results in phase one the final answers? In Algorithm *PTopKTwig*, an enhanced lower bound is proposed to get the final answer quickly and to ensure the correctness of algorithm. The *HyTopKTwig* algorithm also needs to ensure the correctness and try to find other candidate answers which may be the final answers, so we use the skeleton results to bound the structural region of those candidate answers. The correctness of Algorithm *HyTopKTwig* is proved below:

Theorem 1. *Given a twig query q and an probabilistic XML database PD, Algorithm* HyTopKTwig *correctly returns all the answers for q with top-k probabilities on PD.*

Proof. If the k skeleton results in set S_b can bound the final answers, then we can prove the correctness of algorithm *HyTopKTwig*, so we change the problem to prove that there is no element, which is not the descendant of any skeleton results in set S_b, can contribute to the final answers.

Assuming that there is an element e which is not the descendant of any skeleton results in set S_b, and can match a twig answer with element e_x whose probability value is larger than the k-*th* twig answer (merge-joined by ek_1 and ek_2). So we can get the inequations:

$$pathProb(ek_1) * preProb(ek_2) < pathProb(e_x) * preProb(e) \qquad (1)$$

equation (1) can be deduced to equation (2)

$$pathProb(e_x) > pathProb(ek_1) * preProb(ek_2)/preProb(e) \qquad (2)$$

In phase one, once the function *getNextP()* returns a tag t, algorithm will regard the tag t as main path and find matching with the other element streams ordered by predicate probability value, so the preProb(ek_2)/preProb(e) must be larger than one because the path answer with larger predicate probability value has been scanned, we can conclude that pathProb(e_x) is larger than pathProb(ek_1), this is contradictory with that the processed main path has larger path probability value than the unprocessed ones. Or if the pathProb(e_x) is really larger than pathProb(ek_1), it means element e_x has been processed before ek_1, and the skeleton result of e_x has existed in set S_b. Because the element e can be merge-joined with e_x, the element e must be the descendant of skeleton result of e_x. It means element e will be dealt with in phase three of algorithm *HyTopKTwig*. Therefore we can conclude that there is no element which is not the descendant of any skeleton results in set S_b, and can match a twig answer whose probability value is larger than the k-*th* twig answer. So the correctness of algorithm *HyTopKTwig* is proved.

5 Experiments

5.1 Experimental Setup

The algorithms *HyTopKTwig*,*ProTJFast* and *PTopKTwig* were implemented in JDK 1.4. Both real-world data set DBLP and synthetic data are used to test the performance of algorithm above, and the synthetic data set was generated by IBM XML generator and a synthetic DTD. We made the corresponding probabilistic XML documents from ordinary ones by inserting distribution nodes and assigning probability values to the child elements of distribution nodes. Table 1 lists the used queries. We take the metrics elapsed time and processed element rate $rate_{proc} = num_{proc}/num_{all}$ to compare the performance among those algorithms, where num_{proc} is the number of processed elements, and num_{all} is the number of all elements.

Table 1. Queries

ID	queries
Q_1	dblp//article[//author]//title
Q_2	S//[//B]//A
Q_3	S//[//B][//C]//A
Q_4	S//[//B][//C][//D]//A
Q_5	S//[//B][//C][//D][//E]//A

5.2 Performance Study

Influence of Document Size We evaluated Q_1 against the DBLP data set of different sizes, ranging from 20MB to 110MB, and the answers with top-20 probability values were returned. From Figure 4, we can see that the elapsed time of *ProTJFast* is linear to the size of documents, while varying size of documents has almost no impact on *HyTopKTwig* and *PTopKTwig*, and the algorithm *HyTopKTwig* performs better than *PTopKTwig*.

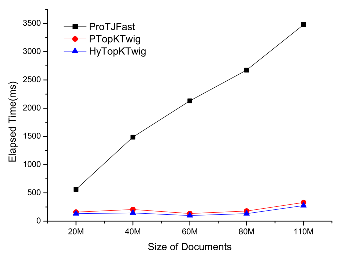


Fig. 4. Varying size

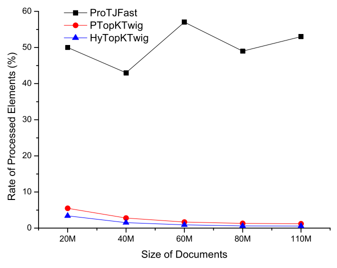


Fig. 5. Varying size

Influence of Number of Answers Query Q_1 was evaluated against the DBLP data set. From Figures 6 and 7, we can see that by varying k from 10 to 50, the elapsed time and the rate of processed elements of those algorithms increase, the algorithm *HyTopKTwig* performs best, and algorithm *ProTJFast* performs worst. When k is small, the performance of *HyTopKTwig* is much better than *ProTJFast* and is better than *PTopKTwig*.

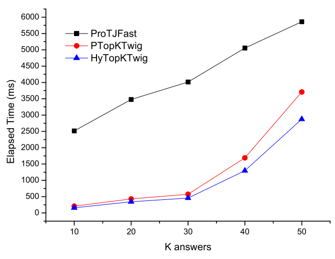


Fig. 6. Varying K

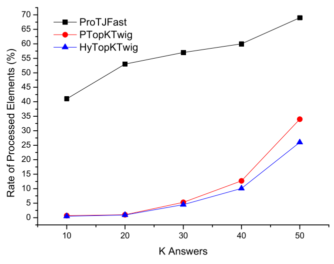


Fig. 7. Varying K

Influence of Multiple Predicates To test the influence of multiple predicates, the queries Q_2 to Q_5 were evaluated on the synthetic data set. By varying the fan-out of query from 2 to 5, from Figures 8 and 9, we can see that the elapsed times of algorithm *PTopKTwig* increases rapidly. The situation is similar in Figure 9 when testing the rate of processed elements. The increasing speed of algorithm *HyTopKTwig* is steady and is slower than the other two ones, especially when the query's fan-out is large.

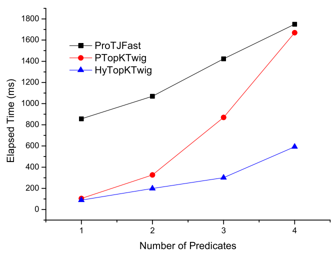


Fig. 8. Varying *pred*

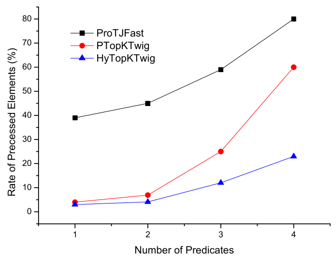


Fig. 9. Varying *pred*

6 Conclusions

In this paper, we addressed the problem of finding top-k matching of a twig pattern against probabilistic XML data. Firstly, we improved the float vector part of *PEDewey* encoding, then we proposed a hybrid algorithm named *HyTopKTwig* that has three phases. The element streams in first phase are ordered by probabilistic value, and the element streams in third phase are ordered by document order, therefore the algorithm *HyTopKTwig* considers about both the probability values constraint and structural relationships constraint. Finally we presented experimental results on a range of real and synthetic data.

Acknowledgement. This work is supported by the Australian Research Council Discovery Project (Grant No. DP0878405, DP110102407), and the Fundamental Research Funds for the Central Universities of China (Grant No. 2009QN030).

References

1. S.Abiteboul and P. Senellart. Queryig and updating probabilistic information in XML. In Prodeeding of EDBT, pp. 1059-1068, 2006.
2. E.Hung, L.Getoor, and V.S.Subrahmanian. Probabilistic interval XML. In Proceeding of ICDT, pp. 358-374, 2003.
3. E.Hung, L.Getoor, and V.S.Subrahmanian. PXML: A probabilistic semistructured data model and algebra. In Proceeding of ICDE, pp. 467-478, 2003.
4. A. Nierman and H.V.Jagasish. ProTDB: Probabilistic data in XML. In Proceeding of VLDB, pp.646-657, 2002.
5. P.Senellart and S.Abiteboul. On the complexity of managing probabilistic XML data. In Proceeding of PODS, pp. 283-292, 2007.
6. B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv. Query efficiency in probabilistic XML models. In Proceeding of SIGMOD, pp. 701-714, 2008.
7. B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic XML. In Proceeding of VLDB, pp. 27-38, 2007.
8. M.Hua, J.Pei, W.Zhang, and X.Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In Proceeding of SIGMOD, pp. 673-686, 2008.
9. M.Hua, J.Pei, W.Zhang, and X.Lin. Efficiently answering probabilistic threshold top-k queries on uncertain data. In Proceeding of ICDE, pp. 1403-1405, 2008.
10. L. Chang, J. X. Yu and L. Qin. Query Ranking in Probabilistic XML Data. In Proceeding of EDBT, pp. 156-167, 2009.
11. K. Yi, F. Li, G.Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases. In Proceeding of ICDE, pp. 1406-1408, 2008.
12. K. Yi, F. Li, G.Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases with x-relations. TKDE 20(12): 1669-1682, 2008.
13. B. Ning, C.Liu, J. X. Yu, and G.Wang : Matching Top-k Answers of Twig Patterns in Probabilistic XML. In Proceeding of DASFAA, pp. 125-139, 2010.
14. T. Grust. Accelerating XPath Location Steps. In Proceeding of SIGMOD, pp. 109-120, 2002.
15. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In Proceeding of SIGMOD, pp. 425-436, 2001.
16. J. Lu, T. W. Ling, C-Y. Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In Proceeding of VLDB, pp. 193-204, 2005.