Suggestion of Promising Result Types for XML Keyword Search

Jianxin Li, Chengfei Liu and Rui Zhou
Swinburne University of Technology
Melbourne, Australia
{jianxinli, cliu, rzhou}@swin.edu.au

Wei Wang
University of New South Wales
Sydney, Australia
weiw@cse.unsw.edu.au

ABSTRACT

Although keyword query enables inexperienced users to easily search XML database with no specific knowledge of complex structured query languages or XML data schemas, the ambiguity of keyword query may result in generating a great number of results that may be classified into different types. For users, each result type implies a possible search intention. To improve the performance of keyword query, it is desirable to efficiently work out the most relevant result type from the data to be retrieved.

Several recent research works have focused on this interesting problem by using data schema information or pure IR-style statical information. However, this problem is still open due to some requirements. (1) The data to be retrieved may not contain schema information; (2) Relevant result types should be efficiently computed before keyword query evaluation; (3) The correlation between a result type and a keyword query should be measured by analyzing the distribution of relevant values and structures within the data. As we know, none of existing work satisfies the above three requirements together. To address the problem, we propose an estimation-based approach to compute the promising result types for a keyword query, which can help a user quickly narrow down to her specific information need. To speed up the computation, we designed new algorithms based on the indexes to be built. Finally, we present a set of experimental results that evaluate the proposed algorithms and show the potential of this work.

Keywords

XML keyword query, result type suggestion

1. INTRODUCTION

XML has evolved to be the standard for data representation and exchange on the Internet. Due to the structural flexibility and heterogeneity of XML data, it is difficult for a user to issue a structured query to express her search request. As such, keyword search has emerged as a popular

paradigm for information retrieval over XML data [8, 6, 13, 21, 14, 19, 12]. One of the significant merits of XML keyword search is its simplicity — users do not need to learn a complex query language (i.e., XQuery or XPath), or know the structure of the underlying data. However, this kind of simple query format may not be precise and can potentially return a large number of results that may be classified into different types, i.e., the label paths of results, among them only a few types are interesting to the users.

To address this problem, one possible way is to first compute the query results and then rank them. Two different ranking functions [4, 6] can be applied in this approach. From the ranked list of results, the type of the highest ranked result is usually selected as the promising result type for the query. Obviously, utilizing one specific result for deciding result type may not be a *robust* method. We believe that measuring the correlation between a result type and the issued keyword query should rely on the analysis of the whole set of relevant information, rather than one piece of it.

Example 1. Consider a keyword query "interest art" issued on the sample data in Figure 1. Most likely, it is intended to find students who are interested in art; hence the result type should be "root/students/student".

Ranking methods proposed in [4, 6] will recommend result type as "root/books/book/title" instead. This is because that the two keywords appear in the same book title "dramatic art & interest", which leads to the highest ranking score for this specific result. The problem with this recommendation method is that it is solely dependent on the quality of *one* query result.

Another method [14] infers the return node type by analyzing keyword match patterns. Keywords are classified into two categories: those that specify search predicates and those that specify return nodes. Identification of a return node relies on node categories: entity node, attribute node and connection node. For instance, given a node, if it corresponds to a *-node in the DTD, then the node is an entity node; if it does not correspond to a *-node and only has one child node which is a value, then the node is an attribute node; otherwise, the node is a connection node. A problem with this approach is that it cannot handle well the case where a node's tagname belongs to any of the three categories at the same time.

Example 2. Since the query keyword "interest" appears in both element names and values, [14] will determine the term "interest" as a match of element names, rather than a value. The suggested result type is hence "root/students/student".
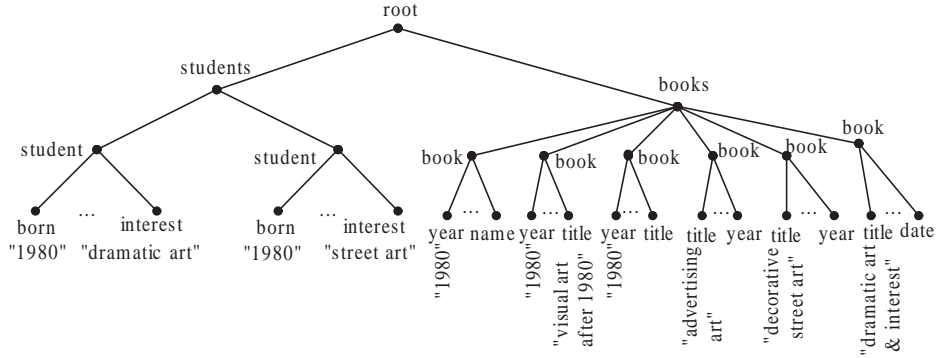
Figure 1: Portion of data tree for a sample of students' reading interests XML database

Another recent proposal relevant to identifying the result type is XReal [3]. It utilizes the statistics of underlying XML data, which summarizes the relationships between element nodes and all tokens in the leaf nodes. Since neither the relationships among the element nodes nor the relationships among values are considered, the precision of this approach may be sub-optimal.

Example 3. Consider another keyword query "1980 art" issued on the sample data in Figure 1. The intension is most likely to find students who are interested in "art" and were born in "1980". Hence, "root/students/student" should be suggested as the result type. XReal will suggest "root/books" instead. This is mainly because their statistic information only considers the relationship between the result type and each independent keyword, not the combination of the query keywords.

In this paper, we propose a new approach to effectively and efficiently identify the result type for keyword search on XML data. Our key idea is to consider all the keyword query answers corresponding to different result types. This avoids the limitations of the previous approaches. In order to save the computation time, we also develop a suggestion method based on a new summarization technique, which takes into consideration both the value and structural distributions of keywords. In addition, our proposed approach does not require a schema of the XML data. We also employ an enhanced ranking function to predict the most relevant result type.

Our contributions in this paper can be summarized as follows:

- We propose a new method of predicting the result type for XML keyword queries. Our method employs a ranking formula that takes into consideration the correlation between a result type and the query keywords based on several types of statistic information.

- We develop a new data structure to estimate accurately the size of keyword query results; the data structure captures both the structural and the value distributions in the data concisely.

- We implement the proposed techniques in a keyword search engine prototype called **XBridge**. Extensive experiments have been conducted demonstrating superior effectiveness, efficiency and scalability of our method against previous methods.

The rest of the paper is organized as follows. Section 2 provides some definitions and introduces the XSKetch index that will be used in this work. Section 3 formally defines the problem of finding promising result types for a keyword query and illustrates the main idea of our approach. We design a general ranking function in Section 4 and propose a set of algorithms that can efficiently work out the promising result types in Section 5. Section 6 presents our experimental evaluation. Section 7 reviews the previous work on XML keyword search. Section 8 concludes the paper.

2. PRELIMINARY

In this section, we first introduce some necessary definitions and then the basics of the XSketch method [18].

Definition 1. [**XML Data Tree**] An XML data tree is defined as $T_t = (V_t, E_t, r)$ where V_t is a finite set of nodes, representing elements and attributes of the data tree T_t; E_t is set of directed edges where each edge $e(v_1, v_2)$ represents the parent-child relationship between the two nodes v_1, v_2 $\in V$; r is the root node of the tree T_t.

We assume that all values appear in the leaf nodes. Given an edge $e = (v_1, v_2)$, we define $P(v_2) = v_1$ and $v_2 \in Ch(v_1)$.

Definition 2. [**Keyword Query**] A keyword query is a set of different terms, denoted by $Q = \{k_1, k_2, \ldots, k_n\}$.

We consider the AND-semantics for the query, i.e., a query result must contain at least one occurrence of each term $k_i \in Q$. For example, Figure 1 represents an XML data tree about student and book information. $Q = \{1980, \text{art}\}$ is a keyword query that is issued by users.

To precisely estimate a keyword query over XML data with a great size, we build upon the ideas of XSketch [18, 17], which is designed to estimate the selectivity of XML twigs.

Definition 3. [**XSketch**] An expanded graph synopsis $G(T_t)$ $= (V, E)$ for an XML data tree T is an edge-labeled directed graph, where V is a set of distinct tag names that occur in V_t; each node in $v \in V$ corresponds to a subset of data nodes in V_t (termed the extent of v, or extent(v)) that have the same label (denoted by label(v)); the label for each edge $(u, v) \in E$ is defined as: (1) label(u, v) = {B}, if v is B-stable w.r.t. u; (2) label(u, v) = {F}, if u is F-stable w.r.t. v; (3) label(u, v) = {F, B}, if both (1) and (2) hold; (4) label(u, v) = {}, otherwise.
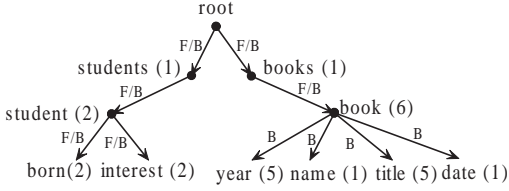
Figure 2: Example of expanded graph synopsis

For example, Figure 2 displays the expanded graph synopsis of the XML data tree in Figure 1. In XSketch, count(v) records the number of elements that map to v in the XML data, i.e., the size of v's extent. For instance, count$(student)$ = 2 and count$(book)$ = 6 in Figure 2. It also captures the localized *backward-* and *forward-stability* conditions across synopsis nodes of an XML data tree. A node u is **B**(ackward)-stable (**F**(orward)-stable) with respect to a parent (resp., child) node v in the synopsis, iff all data elements in *extent*(u) have at least one parent (resp., child) element in *extent*(v). For instance, every *student* element has two child elements *born* and *interest*. At the same time, both *born* and *interest* elements can reach up to their parent *student* elements. So the parent-child edge between *student* and *born* (or *interest*) is marked as F/B. Intuitively, B-stability guarantees that all elements in u descend from v and, therefore, count(u) is an exact estimate for the expression v/u ; similarly, F-stability ensures that all elements in u reach at least one element in v and, therefore, count(u) is an exact estimate for $u[v]$ where v is taken as a predicate of u.

To improve the precision of estimation, XSketch captures two kinds of statistical information: structural and value distribution information. Since the work [17] focuses on the discussion of the relationship between the precision of the estimation and the space budget of building the index XSketch, it only summarizes part of the distribution information so that the space budget can be met. To choose the significant information, it relies on many assumptions. For instance, some values, not all values, on different paths are selected and then the correlations among the selected values are considered. Readers are referred to [18, 17] for more details.

3. OVERVIEW OF OUR WORK

Since it is common for XML data to contain nodes with the same element name in different contexts, we use label paths to denote node types. A label path is a sequence of element names that appear in the path from the root to the node in question.

Definition 4. [**Result Types**] Given a keyword query Q = $\{k_1, k_2, \ldots, k_n\}$ and an XML data tree $T = (V_t, E_t, r)$, result types of Q over T is a list of distinct label paths that start from the root node r and stop at *connected* nodes. Each *connected* node or its descendant nodes should contain at least one instance of the given terms $\{k_1, k_2, \ldots, k_n\}$.

Consider Example 3 again. There are two result types for the keyword query "1980 art": "root/students/student" and "root/books/book".

Given a set of result types and a scoring function, we define the *promising result type* as the one that has the maximum score.

Figure 3: Example of structural distribution

For example, "root/students/student" is the promising result type according to our proposed ranking function (See Equation (3) in Section 4).

The conceptual evaluation process to determine the promising result type from a set of candidates is as follows: we first compute all query results individually and classify them based on their types. Then each type aggregates the local score of each individual result in the same group together where the score is computed by a proposed ranking function. Finally, the type with the highest score is chosen as the promising result type. Obviously, it is time-consuming when the size of the data to be searched is large. Therefore, we would like to propose an efficient approach that can achieve the similar target (i.e., locating the promising result type) with much lower cost (i.e., by estimating the score of each type).

In this paper, we concentrate on the precision of the estimation without considering the usage of space. This is because we find that building an XSketch-like index only requires a small percentage of space compared to the data [18]. Therefore, we can refine the index to the point where all edges in G belong to one of the three types: {F, B, F/B} (See Definition 3). More specifically, we keep splitting the corresponding node sets for the edges that cannot satisfy the stable conditions until one type of stabilities is achieved. In addition, we utilize a tree-style synopsis T, rather than a graph-style synopsis G, to maintain the precise structural and value distribution information, which can reduce more estimation errors. In the rest of the paper, we use T to represent the tree synopsis of the XML data.

Definition 5. [**Structural Distribution**] Consider an internal element node $u \in V_t$ and its child nodes $\{v_i\} \subset V_t$, the structural distribution $H(u)$ of u consists of a set of (1) *distinct distribution types* $(C_{v_1}, C_{v_2}, \ldots, CS_u)$, where C_{v_i} is the count of the child nodes in u, which have the same tag name with v_i, and CS_u is the count of the sibling nodes of u, which have the same tag name with u; and (2) its corresponding *distribution rate* h_u, which is the percentage of *extent*(u), which contains the same set of child nodes and holds the same distribution type.

For example, Figure 3 displays the structural distribution of the internal element nodes *student* and *book* of the XML data tree in Figure 1. Consider first the *student* element node, it has one *born* and one *interest* child node, i.e., C_{born} = 1 and $C_{interest}$ = 1. And it has a sibling node with the

F(born):

$E_{'1980'}$	CS_{born}	$f_{'1980'}$
1	1	100%

F(interest):

$E_{'art'}$	$CS_{interest}$	$f_{'art'}$
1	1	100%

$E_{'dramatic'} \cdot E_{'art'}$		$CS_{interest}$	$f_{'dramatic','art'}$
1	1	1	1/2

...

F(year):

$E_{'1980'}$	CS_{year}	$f_{'1980'}$
1	1	3/5

F(title):

$E_{'art'}$	CS_{title}	$f_{'art'}$
1	1	4/5

$E_{'dramatic'} \cdot E_{'art'}$		CS_{title}	$f_{'dramatic','art'}$
1	1	1	1/5

...

Figure 4: Example of value distribution

same tagname *student* under the parent node *students*, i.e., $CS_{student} = 2$. As such, its distribution type is (1, 1, 2). Because both *student* element nodes have the same distribution type, the distribution rate $h_{student}$ is 100%. For the *book* element node, it has six sibling nodes with *book* as their tag names, i.e., $CS_{book} = 6$. For the first *book* element node, it contains one *year* node and one *name* node, hence its distribution type is (1, 1, 6) and its rate is 1/6 because no other *book* contain the same set of child nodes and satisfy the same distribution type. Similarly, we can work out the type and rate for the last *book* element node as (1, 1, 6) and 1/6. For the rest four *book* nodes, all of them contain the same set of child nodes and hold the same distribution type, hence the rate of the new type is 4/6. All these information can be precomputed and cached for estimating the result size of a keyword query.

After we summarize the structural distribution information H, we start to extract the value distribution based on H from the XML data.

Definition 6. [**Value Distribution**] Given a leaf node $v \in V_t$ and its corresponding size $|extent(v)|$ in the tree synopsis T, the value distribution $F(v)$ of v consists of a set of (1) distinct *distribution types* $(E_{token_1}, E_{token_2}, ..., CS_v)$ where E_{token_i} is 1 (or true) if the informative token $token_i$ exists in the leaf node v and CS_v is the count of the sibling nodes of v where the sibling nodes have the same tag name with the node v; and (2) its corresponding *distribution rate* f is the percentage of $extent(v)$, which contains the same set of distinguished informative tokens and holds the same distribution type.

For example, Figure 4 displays the value distribution of the leaf nodes *born*, *interest*, *year* and *title* of the XML data tree in Figure 1. Consider the *born* element nodes: both of them in $extent(born)$ contain the token '1980', i.e., $E_{'1980'} = 1$; and neither of them has sibling nodes, i.e., $CS_{born} = 1$. In addition, we also know that the size of $extent(born)$ is 2. As such, the value distribution type of $F(born)$ is (1, 1) and its total rate is $f_{'1980'} = 100\%$. Similarly, we can work out the value distributions of *interest-art*, *year-1980* and *title-art*, respectively. Note that the size of $extent(year)$ (or $extent(title)$) is 5 and three of them contain the token '1980' (or 'art'). Furthermore, we can extend the value distribution to multiple tokens. For a leaf node *interest* and two tokens {'dramatic', 'art'}, we find that only the first *interest* element node contains the two tokens together. Therefore,

its type should be $(E_{'dramatic'}, E_{'art'}, CS_{interest})=(1, 1, 1)$ where $CS_{interest}$ is 1 because there is no sibling node that is labeled as *interest*. As one out of two *interest* element nodes has both 'dramatic' and 'art' tokens, its total rate is $f_{'dramatic','art'}=1/2$. All these information can be precomputed and cached for estimating the size of a keyword query.
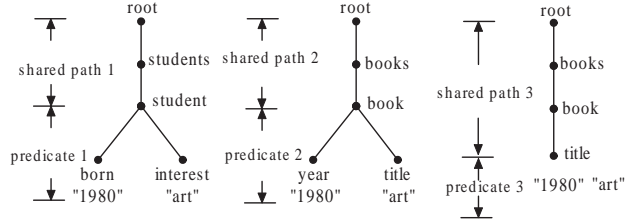

Figure 5: Example of generated query templates

3.1 Our Main Idea

The idea of our approach is as follows. Given a keyword query, we first determine the corresponding distinct paths w.r.t. each term in the query, and generate the query templates in which the given terms are kept. Then, we can obtain all result types where each type is a distinct label path from the root node to the lowest connected node on its query template. Here, the connected node has the similar semantics to the concept of LCA (Lowest Common Ancestor) for keyword queries over XML data. After that, we estimate the query templates over the expanded tree synopsis with the help of distribution information, and compute the score for each result type by calling a ranking function. Finally, the result type with the highest score will be suggested as the promising result type to the query.

Now let's briefly introduce the procedure of our approach by taking Example 3 as an example. For the given keyword query "1980 art", we can generate two query templates by merging their distinct label paths, as shown in Figure 5.

Firstly, we estimate the shared paths over the tree synopsis in Figure 2 using the forward- and backward-stability information. The size of "root/students/student" is 2 while the size of "root/books/book" is 6. After that, we begin to invoke the estimation of the corresponding predicates of the two shared paths. For the predicate of the first shared label path, i.e., predicate 1, we have $2 \cdot h_{student} \cdot c_{born} \cdot c_{interest} \cdot f_{'1980'} \cdot f_{'art'} = 2 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 2$. For the predicate of the second shared label path, i.e., predicate 2, we have $6 \cdot h_{book} \cdot c_{year} \cdot c_{title} \cdot f_{'1980'} \cdot f_{'art'} = 6 \cdot \frac{4}{6} \cdot 1 \cdot 1 \cdot \frac{3}{5} \cdot \frac{4}{5} = 1.92$. Since the total score of "root/students/student" is larger than that of "root/books/book", the former will be recommended as the promising result type to the users. From this example, we can see that our approach utilizes the combination of the relevant nodes to measure the correlations between each result type and the given keyword query.

4. SCORING FUNCTION

In this section, we develop our scoring functions for result type suggestion for XML keyword queries.

4.1 Scoring a Return Type Candidate

Consider a keyword query $Q = \{k_1, k_2, \ldots, k_n\}$. It might corresponds to different interpretations. For example, consider the query "1980 art" in Figure 1, two possible interpretations (also called *query templates*):

1. /root/students/student[born $\sim$ "1980"][interest $\sim$ "art"], and

2. /root/books[book/year $\sim$ "1980"][book/title $\sim$ "art"].

where $\sim$ is a shorthand for testing if the term is contained in a node. The *return types* of the above two interpretations are /root/students/student and /root/books, respectively. Generally, a return type corresponds several query templates. For example, the query template /root/books[book/title $\sim$ "1980"][book/title $\sim$ "art"] also contributes to the latter return type.

Given any interpretion of a keyword query Q, consider one of its result $R = \{N, n_1, n_2, \ldots, n_n\}$. where each n_i is a leaf node and contains the term k_i, and node N is the lowest common ancestor (LCA) of $n_1, \ldots, n_n$.

We consider the following scoring functions to compute the score of R.

Ranking Function 1: This type of ranking function only considers the term information, e.g., the TF-IDF method or its variations. The standard TF-IDF is from the information retrieval field and considers both term frequency (i.e., how many times a term appears in a document) and inverse term frequency (i.e. inverse of how many documents contain the term). In order to apply it to the typical XML database scenarios, we make the following adaptation: (1) we assume term frequency *tf* is always equal to 1 as in [7], and (2) we use *inverse element frequency*, which is defined as the total number of element in the XML data tree over the number of elements that contains the token in the subtree rooted at the element in question. Finally, we can obtain the weight of each node n_i as

$$weight(n_i) = \log_2(1 + tf) \cdot \log_2 ief = \log_2 ief$$

where $tf = 1$ and the final score of R as

$$Score_1(R, Q) = \sum_{i=1}^{n} weight(n_i) \qquad (1)$$

Let's consider Example 3 where the query is $\{1980\ art\}$ over the XML data tree in Figure 1. Based on the formula, for the first "student" element in Figure 1 we have $weight_{1980} = \log_2(27/13) = 0.947$ and $weight_{art} = \log_2(27/15) = 0.847$. The score of the first "student" w.r.t the query $\{1980\ art\}$ is therefore $0.947 + 0.847 = 1.794$.

The obvious problem with this scroing function is that it overlooks the structure of the result.

Ranking Function 2: The second ranking function considers both terms and structures of the result.

$$Score_2(R, Q) = \sum_{i=1}^{n} \frac{weight(n_i)}{dist(N, n_i)} \qquad (2)$$

where $Distance(N, n_i)$ is the length of the path from N to n_i in the XML data tree.

The intuition is that terms appear far from the root of the result subtree should be penalized. This function usually works well to measure the relevance of a result at the element level. However, as it penalizes each term weight $weight(k_i)$ independently using $Distance(N, n_i)$, it may favor the terms with short distance values. For instance, consider a keyword query $Q = \{k_1, k_2\}$ and one of its results $R = \{N, n_1, n_2\}$ in Figure 6(a). The score of R can be calculated: $Score_2(R, Q) = \frac{5}{10} + \frac{1}{2} = 1$. In this case, n_1 is more interesting than n_2 as the weight of k_1 is much larger than that of k_2, yet they have the same contribution to the final score. Therefore, we should reduce the penalty of the term, which are brought by its corresponding long distance.

Another problem with this ranking function is that it cannot differentiate a tightly-coupled result from a loosely-coupled one. An example is given in Figure 6(b). According to Equation (2), two given query templates will have the same score. Obviously, the template on the right side should be better than the one on the left side.
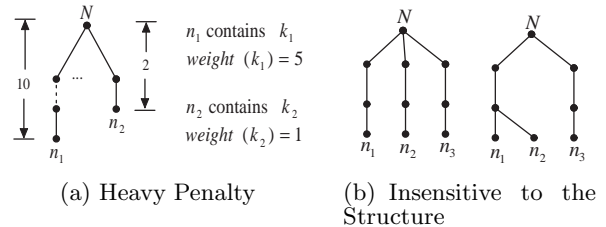

(a) Heavy Penalty   (b) Insensitive to the Structure

Figure 6: Problems with Ranking Function 2

Ranking Function 3: Motivated by the above analyses, we propose a new ranking function that embraces the principles of the above two functions, i.e., it is proportional to the total weights of the terms as Function 1 does; At the same time, it also takes into account the effects of structures of XML. More specifically, it overcomes the two issues identified from Ranking Function 2.

$$Score_3(R, Q) = \begin{cases} \sum_{i=1}^{n} weight(k_i), & \text{if } \widetilde{dist}(N, n_i) = 0 \\ \frac{\sum_{i=1}^{n} weight(k_i)}{(\sum_{i=1}^{n} dist(N, n_i) - \theta)^{\gamma}}, & \text{Otherwise.} \end{cases} \qquad (3)$$

where θ is the total times of the edges that are repeated on the path from N to each n_i in the XML data tree; γ is a parameter to balance the impact of the structure to the score (its default value is 2); and $\widetilde{dist}$ is defined as

$$\widetilde{dist}(N, n_i) = \begin{cases} dist(N, n_i), & dist(N, n_i) \leq avg\text{-}depth \\ avg\text{-}depth + (dist(N, n_i) - avg\text{-}depth) \cdot \eta, & \text{else} \end{cases}$$

where *avg-depth* is the average depth of the XML data tree; and $\eta \in [0, 1]$ is a tunable parameter.

Example 4. Consider the three XML documents in Figure 7 and a keyword query $Q = \{k_1, k_2\}$. There are three query templates and they correspond to different return types. Denote any of the query results for the three query templates as r_1, r_2, and r_3, respectively, and assume the weights for both keywords are 1 and $\gamma = 2$, we calculate the

(a) Document d_1 (b) Document d_2 (c) Document d_3

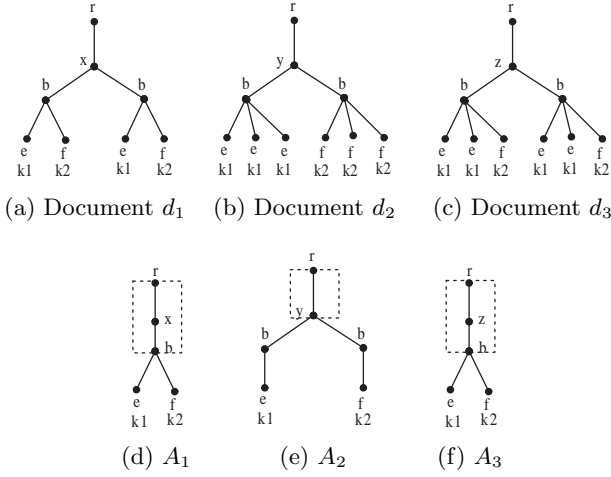(d) A_1 (e) A_2 (f) A_3

Figure 7: Three Query Templates for the Query over Three Documents where Their Corresponding Return Types are Indicated in Dotted Boxes.

scores of the individual results as:

$$Score_3(r_1, Q) = \frac{1+1}{(1+1)^\gamma} = \frac{1}{2}$$

$$Score_3(r_2, Q) = \frac{1+1}{(2+2)^\gamma} = \frac{1}{8}$$

$$Score_3(r_3, Q) = Score_3(r_1, Q)$$

In the rest of the paper, we will use the Equation (3) as our scoring function.

4.2 Aggregating Scores

We know that using only one (or a few) query results to predict the promising result type may not be a good idea, as illustrated in Section 1. Instead, we would like to aggregate the score of all query results in each return type to determine the promising result type. This involves summing up scores of all results of all query templates associated with each return type.

However, a technical issue is that different return types usually correspond to very different numbers of results, hence this method will be inevitably biased towards return types with many query results. Therefore, it is desirable to consider at most top-K results for each return type when performing the score aggregation. We suggest that a reasonable way to determine K is to set it to the average number of results per return type. Consider m result type candidates RTC_i ($1 \le i \le m$) exist in the data to be retrieved and each type candidate RTC_i has $c(RTC_i)$ query results.[1] The average return type query result is

$$avg\text{-}results = \frac{1}{m} \sum_{i=1}^{m} c(RTC_i)$$

Note that if the number of query results of a return type is less than *avg-results*, we will use all its results to aggregate the scores.

---

[1] Note that in case a return type has several associated query templates, the numbers of results of all associated query templates need to be summed up.

Example 5. Continuing the three return types in Figure 7, they have 2, 9, and 4 query results in the corresponding documents, respectively. Therefore, *avg-results* is $\frac{2+4+9}{3} = 5$. Given that each return type in this example has only one query templates and the fact that all results in the same query templates have the same score, we can calculate the score for each result types as:

$$Score(\text{r/x/b}) = \frac{1}{2} \cdot 2 = 1$$

$$Score(\text{r/y}) = \frac{1}{8} \cdot 5 = 0.625$$

$$Score(\text{r/z/b}) = \frac{1}{2} \cdot 4 = 2$$

Finally, the top-scoring return type, r/z/b, will be chosen as the promising return type.

5. ALGORITHMS OF FINDING PROMISING RESULT TYPES

In this section, we first introduce a general keyword search algorithm that uses the actual keyword search results to work out the promising result types. Then we propose a more efficient algorithm that utilizes the offline summarized statistics to predict the promising result types.

5.1 Inverted Node List based Algorithm (INL)

The basic idea of INL is to first retrieve all relevant node lists w.r.t. a keyword query, and then merge the node lists from the shortest one, i.e., the node with the lowest document frequencies will be processed first. To increase the efficiency, we deploy the Dewey number scheme to encode the nodes before-hand, as it enables us to efficiently compute the lowest common ancestor node given a set of nodes identified by their Dewey codes. During the computation, we record the shared paths as the result type candidates. For each type candidate, we maintain a data structure (See Table 1) where we take the individual result score as a key and the number of this kind of results are added together as a value. Furthermore, the table is always sorted by the individual result score in a descending order. At the same time, we also record the total number of query results and the number of distinct result types, which are later used to compute the threshold *avg-results*. After we process all the node lists, we calculate the score for each return type candidate by using its data structure and the threshold *avg-results*. The type candidates with the highest scores will be selected as the promising result types.

Table 1: Data structure for a type candidate RTC_i

Individual Score	#Results
3.0	20
2.5	18
1.5	10
⋮	⋮

This approach obviously will incur substantial amount of computation as it needs to access and merge all the occurrences of keywords. The complexity of *INL* is $O(\prod_{i=1}^{n} |L_i|)$ A more detailed discussion is provided in Section 6.

5.2 Statistic Distribution Information-based Algorithm (SDI)

In this section, we first show the procedures of building structural and value distribution indexes. Then we generate a set of query templates for a keyword query. Finally, an efficient algorithm is proposed to estimate the constructed query templates based on the pre-built distribution indexes.

5.2.1 Building Structural Distribution

Algorithm 1 Building structural distribution index

---

input: An XML data tree T_t
output: An expanded tree synopsis T with the distribution rate $H(v)$ of each synopsis node $v \in V$

1: initiate an empty tree synopsis T' and an empty set $H()$;
2: $r_t \leftarrow$ the root node of T_t;
3: Q.enqueue(r_t)
4: **while** Q is not empty **do**
5:     $v \leftarrow Q$.dequeue()
6:     $Ch_v \leftarrow v$.children();
7:     **for all** node $v_c \in Ch_v$ **do**
8:       **if** T' contains edge (v, v_c) at the current level **then**
9:         IncreaseCount(v, v_c);
10:       **else**
11:         AddNewEdge(v, v_c, T');
12:       **end if**
13:       Q.enqueue(v_c)
14:     **end for**
15: **end while**
16: $T \leftarrow$ MarkFBStability(T', T_t);
17: GenerateDistribution(T, T_t);
18: **return** (T, H);

In this section, we introduce the procedure of summarizing the structural information of an XML data tree; this helps to improve the precision of the estimation as it considers the correlations among the outgoing edges of the tree nodes. It consists of three main steps: generating the intermediate tree synopsis T' from an XML data tree; marking the synopsis nodes of T' with *Forward-* and *Backward-*stability conditions; and making the statistics about the distribution of outgoing edges across the synopsis nodes of the XML data tree.

Generating tree synopsis The basic idea here is similar to the breadth-first search (BFS) over a tree that begins at the root node and explores all its neighboring nodes. Then for each of these nodes, it explores their unexplored neighboring nodes until all nodes in an XML tree are explored. During the tree traversal, the structure of the data tree is abstracted by using AddNewEdge(v, v_c, T') and the *extent* information for each synopsis node is obtained by using IncreaseCount(v, v_c). Once the traversal is completed, the intermediate tree synopsis T' of the XML data tree T_t is built fully. The detailed procedure is provided from Lines 1-Line 15 in Algorithm 1. After that, we need to mark the stability for each edge over the tree synopsis by using the function MarkFBStability() and then call GenerateDistribution() to produce the structural distribution types and their rates.

Marking edges with MarkFBStability() To capture the localized *Forward-* and *Backward-*stability conditions across the synopsis nodes of an XML data tree, it is required to traverse the intermediate tree synopsis T' and check the stability of each synopsis edge(v_1, v_2). For the edge, we first re-

---

Algorithm 2 Function GenerateDistribution(T, T_t)

---

input: The expanded tree synopsis T and the XML tree T_t
output: The expanded tree synopsis T with the structural distribution information H

1: **for all** internal synopsis node $v \in V$ **do**
2:     Initialize $H(v, h) = 0$, $\forall (v, h)$
3:     **for all** instance $v_t \in extent(v)$ **do**
4:       $sibSize \leftarrow$ |GetSiblingByTagname(v_t)|;
5:       $h \leftarrow$ SummarizeChildDist(v_t);
6:       h.append($sibSize$);
7:       $H(v, h) \leftarrow H(v, h) + \frac{1}{|extent(v)|}$;
8:     **end for**
9: **end for**
10: **return** the structural distribution information H;

---

trieve the corresponding groups, $extent(v_1)$ and $extent(v_2)$, of nodes that are labeled as v_1 and v_2, respectively. Then we need to compare the two sets of nodes, which leads to three cases. Case 1: edge (v_1, v_2) satisfies F-Stability because every instance node of v_1 in $extent(v_1)$ has at least one child instance node of v_2 in $extent(v_2)$, which means that we can walk from v_1 to v_2 in a forward step. Similarly, Case 2 means that we can walk from v_2 to v_1 in a backward step. If neither conditions holds, it is required to call a special operation Split($extent(v_2)$) that partitions the group nodes $extent(v_2)$ into some *Maximal* subgroups. All the subgroup nodes should keep the *Forward-* and *Backward-*stability with the nodes $extent(v_1)$. Here, the word *Maximal* means that given any two subgroups $extent_1(v_2)$ and $extent_2(v_2)$, if we move a node from $extent_1(v_2)$ (or $extent_2(v_2)$) to $extent_2(v_2)$ (or $extent_1(v_2)$), the resulting group nodes $extent_2(v_2)$ (or $extent_1(v_2)$) and the nodes $extent(v_1)$ do not satisfy either *Forward-* or *Backward-*stability.

Summarizing edges' distribution: To summarize the outgoing edges' distribution for an internal node v, it is required to find out its child nodes $Ch(v)$, its sibling nodes that have the same tagname with v, and the size $|extent(v)|$ of its extent, respectively. And then we summarize the count C_{v_i} from the child nodes $v_i \in Ch(v)$ and the count CS_v from its sibling nodes, which is used to generate a distribution type. Hence, the rate of this distribution could be increased by $1/|extent(v)|$. The detailed procedure is illustrated in Algorithm 2 where Function GetSiblingByTagname() is used to calculate the number of the sibling nodes that share the same tag name with the given internal node and Function SummarizeChildDist() is used to compute the number of the child nodes with different tag names.

5.2.2 Building Value Distribution Index

In this section, we introduce the procedure of deriving the value distribution for each synopsis node v at the leaf level of expanded tree synopsis T. Each instance node $v_t \in extent(v)$ may contain a single token, e.g., "year" or multiple tokens, e.g., "title" (See Figure 1). If we build an index to record all possible combinations of the terms in the whole document, the index would be too large to be practical.

Nevertheless, recent studies [2, 9] suggest that the typical keyword query length is within two to four tokens and that the possibility of longer queries decreases greatly with the query length. In addition, we also find that for a keyword query, the possibility of all its tokens that are located at one specific instance node is typically low. That is to say, even if a keyword query contains more than four tokens, only a few

Algorithm 3 Building the value distribution index

input: An XML tree T_t and its expanded tree synopsis T
output: the distribution rate $F(v)$ where $l = r/.../v$ is a path of T and v is a leaf-level synopsis node

1: **for all** leaf synopsis node $v \in V$ **do**
2:    Initialize $F(v,l) = 0$, $\forall(v,l)$
3:    **for all** $v_t \in extent(v)$ **do**
4:       $sibSize \leftarrow |v_t.\text{parent}.\text{getChildNodesByTagname}(v_t)|$;
5:       $tokens \leftarrow \text{FilterGetTerms}(v_t)$;
6:       $\mathcal{L} \leftarrow \text{GenCombination}(tokens, sibSize)$;
7:       **for all** token combinations $l \in \mathcal{L}$ **do**
8:          $F(v,l) \leftarrow F(v,l) + \frac{1}{|extent(v)|}$
9:       **end for**
10:    **end for**
11: **end for**
12: **return** the value distribution information F;

---

of them belong to the same leaf node. Therefore, for each instance leaf node v_t, we build four kinds of combinations: 1-Com, 2-Com, 3-Com, 4-Com, where 1-Com represents each single token in the v_t's value content; i-Com represents all combinations that have i tokens ($2 \le i \le 4$).

As shown in Algorithm 3, for each leaf synopsis node $v \in V$, we can obtain a subgroup of instance nodes $extent(v) \subset V_t$ and derive the value distribution by processing each instance node $v_t \in extent(v)$. Firstly, we extract informative tokens from the value content of v_t by calling the function $\text{FilterGetTerms}(v_t)$ where the content is split into tokens and non-informative tokens are filtered out by a stop-word list that can be edited or imported by the DB designer. Then, we calculate the count of v_t's sibling nodes that have the same tag name with v_t. Finally, the function GenCombination() is invoked to generate all subset of size no larger than four from tokens in *elements*.

For each combination l, we append the count of v_t's sibling nodes that have the same tag name with v_t to the end of the combination ℓ. Once a new combination ℓ is generated, we insert it into the value distribution set $F(v)$ if ℓ exists beforehand or update the rate of $F(v,\ell)$.

5.2.3 Finding Result Type Candidates

To predict the promising result type, we have to find all result type candidates first. Typically metadata is far less than the data itself. Therefore, the number of distinct label paths is far less than that of the path instances. Given a keyword query $Q = \{k_1, k_2, \ldots, k_n\}$, we can quickly locate the distinct label paths for each k_i and then work out the result type candidates by applying a "tight" merge operation to these paths. By a "tight" join, we mean that for any two paths, they are connected with the longest common prefix.

Firstly, we retrieve the distinct label paths for each keyword by accessing the index *Token2Path*. Then we check the number of the given keywords. If the query only contains one term, we take all distinct label paths as the result type candidates where we directly generate the query templates by combining each label path and the term together. And we take the weight of the term as the score of a single result. If a user's query contains more than one term and each of them corresponds to a list of label paths, we need to consider all combinations of the label paths that come from the different path lists. This task is divided into two steps, which is shown in Algorithm 4. In Step 1: Line 12 - Line 17, we merge the two shortest lists of label paths that

---

Algorithm 4 Collecting result type candidates & predicates

input: A token-to-path index *Token2Path* and a keyword query $Q = \{k_1, k_2, \ldots, k_n\}$
output: A set of query templates Φ organized as a hash map from *sharedPath* to a list of query templates t_i, where each t_i is in the form of $(score, predicates)$

1: Initialize $\Phi[x] = \emptyset$, $\forall x$;
2: **for all** query keyword k_i **do**
3:    $LP_{k_i} \leftarrow$ the set of label paths k_i appears in the XML data (via *Token2Path* index)
4: **end for**
5: **if** $|Q| = 1$ **then**
6:    **for all** label path $p \in LP_{k_1}$ **do**
7:       $sp \leftarrow p$; $preds \leftarrow \emptyset$; $score \leftarrow \text{CalcScore}(sp, preds)$
8:       $\Phi[sp] \leftarrow \Phi[sp] \cup (score, preds)$
9:    **end for**
10: **else**
11:    let $q_1, q_2, \ldots, q_n$ be a permutation of $k_1, k_2, \ldots, k_n$ such that LP_{k_i} is in increasing order of length;
12:    **for all** label path $u \in LP_{k_1}$ **do**
13:       **for all** label path $v \in LP_{k_2}$ **do**
14:          $(sp, preds, score) \leftarrow \text{MergePaths}(u, v, q_1, q_2)$;
15:          $\Phi[sp] \leftarrow \Phi[sp] \cup (score, preds)$
16:       **end for**
17:    **end for**
18:    **for** $j = 3$ **to** n **do**
19:       **for all** label path $p \in LP_{k_j}$ **do**
20:          $\text{AddPath}(p, q_j, \Phi)$;
21:       **end for**
22:    **end for**
23: **end if**
24: **return** Φ;

---

correspond to two of the given keywords where a function $MergePaths(p_1, p_2, k_1, k_2)$ is designed to combine any two paths $p_1 \in LP_{k_1}$ and $p_2 \in LP_{k_2}$. It starts from the comparison of the first node of each path. If the two nodes have the same label, we then continue to compare the next node. The comparison is done recursively until we find the nodes that have different labels or one of the path does not have the next node. At this point, we denote the part that can be shared by the two paths as the shared path *sp* while the rest of the two paths as the predicates *preds* that takes the two keywords k_1 and k_2. At the same time, the weight of the predicate is recorded as *score*. All of the output will be taken as the intermediate query templates Φ. In Step 2: Line 18 - Line 22, we merge the next shortest list of label paths into the above intermediate query templates Φ if the query contains three or more keywords, which is implemented by a function $AddPath(p, k_j, \Phi)$. In other words, for each template $t_i \in \Phi$, we first compare path p and *sp* of t_i. If p is equal to *sp*, then we only merge the new keyword into the predicate *preds* of t_i; If p covers *sp*, then we need to do the comparison of the rest of p and *preds*; Otherwise, we will generate a new shared path and a new predicate because p and *sp* only share partially. In all three conditions, the weight of the new predicate should be updated and the new keyword k_j should also be inserted into the updated predicate.

5.2.4 Method SDI

Our SDI algorithm is based on <u>S</u>tatistical <u>D</u>istribution <u>I</u>nformation stored in the tree synopses introduced in Sections 5.2.1 and 5.2.2. The key idea is to use the synopsis to estimate the score for each candidate query templates rather than actually executing it.

Algorithm 5 Computing results with SDI index

input: A set of query templates Φ organized as a hash map from *sharedPath* to a list of query templates t_i, where each t_i is in the form of $(score, predicates)$
output: Top-K most promising result types

1: **for all** shared label path $sp \in \Phi$ **do**
2:    $t_i \triangleq (sp, score, predicates) \leftarrow \Phi[sp]$;
3:    $est \leftarrow \mathsf{EvaluateSinglePath}(sp)$;
4:    $v \leftarrow$ the end node of sp;
5:    $H.\mathsf{enqueue}(v)$;
6:    **for all** predicate $pred \in predicates$ **do**
7:      **while** H is not empty **do**
8:        $v \leftarrow H.\mathsf{dequeue}()$;
9:        **if** v is a leaf node **then**
10:          $\ell \leftarrow \mathsf{DistStyle}(v.\mathsf{content}(), v)$;
11:          $est \leftarrow est \cdot \mathsf{VDRate}(F, v, \ell)$;
12:        **else**
13:          $h \leftarrow \mathsf{DistStyle}(Ch(v), v)$;
14:          $est \leftarrow est \cdot \mathsf{SDRate}(H, v, h) \cdot \prod\{C_{v_c} | v_c \in Ch(v)\}$;
15:          **for all** $v_c \in Ch(v)$ **do**
16:            $H.\mathsf{enqueue}(v_c)$;
17:          **end for**
18:        **end if**
19:      **end while**
20:    **end for**
21:    **if** $est \neq 0$ **then**
22:      $rscore[sp] \leftarrow rscore[sp] + est \cdot score(t_i)$;
23:    **else**
24:      Modify the current query template t_i by moving the last node in its shared path to the predicates;
25:      **goto** Line 2
26:    **end if**
27: **end for**
28: **return** the top-K sp in the $rscore$;

Given a query template that contains a shared label path, i.e., a result type, and a list of predicates, we first estimate the path and get the approximate size as the number of the search results. Then we estimate the corresponding predicates where we take into account the counts C_{v_i}, CS_v and their distribution rates. By multiplying the maximal number, the counts, and the rates together, we can obtain the approximate number of the results that match with one case in the query template. If the estimate is zero (e.g., the structural or value distribution synopsis indicates no match for a particular query template), we consider a variant of the current query template by moving the connected node up to its parent node (Line 24).

The detailed procedure is given in Algorithm 5. We consider all result type candidate $sp \in \Phi$, and we retrieve the corresponding query template consisting of a list of predicates *predicates* and the score of the template *score*. And then we estimate the size of the current query template without considering any of its predicates using the function $\mathsf{EvaluateSinglePath}()$; the function considers the F- and B-stability across the nodes of the path. After that, we begin to estimate the predicates in *predicates*. Every predicate is traversed and processed as a small tree and its root node is the end node of RTC. At the same time, the score of each predicate is cached as in Table 1. If the explored node v in the predicate tree is a leaf node, we call the function $\mathsf{VDRate}(F, v, \ell)$ to compute the value distribution rate of ℓ in v by using the statistic information in F. If the explored node v is not a leaf node, i.e., it is an internal node, we call the function $\mathsf{SDRate}(H, v, h)$ to obtain the structural

distribution rate of h in v by using the information in H. To probe F or H, another function $\mathsf{DistStyle}()$ is desirable to produce the distribution types ℓ or h from the predicate tree where the nodes in distribution types are sorted by their tag names. Using the distribution type and rate, we calculate the expansion number of the approximate answers by multiplying the count C_{v_c} where $v_c \in Ch(v)$ and C_{v_c}. Then the computational value will be cached with the corresponding score of the specific predicate in Table 1. Similarly, we repeatedly process for other query templates and record their results in Table 1. Finally, we calculate the aggregated score for each result type candidate and predict the promising ones where *avg-sample* can be computed during the estimation.

6. EXPERIMENTS

To verify the effectiveness and efficiency of our proposed approach , we implemented the **XBridge** system which includes the INL and SDI algorithms, and compared it with XReal [3]. All algorithms were implemented in Java and run on a 3.0GHz Intel Pentium 4 machine with 1GB RAM running Windows XP. We do not consider XSeek [14], which was shown to be outperformed by XReal [3].

6.1 Dataset and Queries

Table 2: Statistics of the Datasets

Dataset	#Elements	Max Depth	Avg Depth
NASA	476,646	8	5.58
UWM	66,729	5	3.95
DBLP	3332,130	6	2.90

To test the applicability of each approach, we choose the real datasets: Nasa 23MB, UWM 2MB and DBLP 127MB from Washington XML Data Repository [1]. The criterion of selection is based on the different depths and sizes of the datasets. If the depth of a document is small, which means that the document is too flat, it is easy for a keyword query to choose the root of the document or the nodes at the higher level as its promising result types. In this case, the result types are straightforward. Most of time, the larger the average depth of a dataset is, the more complex the structure of the dataset may become. It is highly possible for this kind of dataset that contains multiple result types for a keyword query. Therefore, we select three datasets that have the different maximal depths and average depths, which is shown in Table 2.

To extensively demonstrate the performance of each method, we randomly select 6 keyword queries with less than 4 terms for each dataset, which is shown in Table 3. These queries are chosen with different frequencies. Furthermore, we put two noise terms *dipersion* and *Opticaly* in the keyword queries that should generate empty results.

6.2 Quality of Suggestion

To measure the quality of the suggested promising result types, we evaluate all queries in Table 3 over the corresponding datasets. The suggested promising result types for each keyword query are shown in Table 4. In addition, if two suggested promising result types hold the ancestor-descendant relations, we only show the types at the lower level, i.e., the more specific result types.

Table 3: Keyword Queries for Each Dataset

Queries	NASA	UWM	DBLP
Q_{1i}	{magnitude}	{level}	{evaluation}
Q_{2i}	{photographic}	{archeology}	{object oriented}
Q_{3i}	{photographic magnitude}	{Najoom}	{Frank Manola}
Q_{4i}	{rotation dipersion}	{individual supervision}	{concepts applications}
Q_{5i}	{cape photographic}	{building technologies}	{multimedia data type}
Q_{6i}	{Opticaly proper motion}	{approved performance organization}	{Frank database 1983}

Table 4: Promising Result Types for Each Query

Dataset	Queries	INL	XReal	SDI
NASA	Q_{11}	{para, title, definition}	{tableHead}	{para, title, definition}
	Q_{21}	{para, title, definition}	{tableHead}	{title, para, definition}
	Q_{31}	{fields}	{tableHead}	{para, descriptions}
	Q_{41}	{}	{}	{}
	Q_{51}	{fields}	{tableHead}	{fields, source}
	Q_{61}	{}	{}	{}
UWM	Q_{12}	{restrictions, title, comments}	{level}	{restrictions, title, comments}
	Q_{22}	{title}	{title}	{title}
	Q_{32}	{instructor}	{section_listing}	{instructor}
	Q_{42}	{root}	{title}	{root}
	Q_{52}	{title}	{title}	{title}
	Q_{62}	{restrictions}	{restrictions}	{restrictions}
DBLP	Q_{13}	{title}	{title}	{title}
	Q_{23}	{title}	{title}	{title}
	Q_{33}	{author}	{author}	{author}
	Q_{43}	{title}	{title}	{title}
	Q_{53}	{title}	{title}	{title}
	Q_{63}	{proceedings}	{proceedings}	{proceedings}

From the results over NASA in Table 4, we find that XReal only focuses on one node at the higher level while INL and SDI can reach to the more detailed nodes. For the keyword query Q_{11}, the users cannot guess the real meaning from the result type *tableHead* suggested by XReal, but they can easily classify their search intentions from {para, title, definition} recommended by INL and SDI where *para* represents a paragraph of *description*, *title* represents a title of *journal* and *definition* represents a definition of a table *tableHead*. For the keyword query Q_{31}, INL takes *fields* as the promising result type, which is the child node of *tableHead* that is suggested by XReal. However, SDI recommends two more specific types *para, description* that are the descendant nodes of *fields*. For another two keyword queries Q_{41} and Q_{61}, no suggestions are generated by all approaches because of the given two noise keywords {dipersion, Optically}. In this paper, we don't consider the processing of the spelling errors. From the above results, we find that the users can select a better and more meaningful result type from a suggestion of SDI, INL than that of XReal.

From the results over UWM in Table 4, we find that for the keyword queries Q_{12}, Q_{32} and Q_{42}, XReal generates different promising result types from the other approaches and for the other queries, the same suggestions are obtained. For Q_{12}, XReal works out the tagname *level* while the other approaches work out more meaningful search intentions, e.g., the title, the restrictions and the comments of a course. For Q_{32}, INL and SDI can directly suggest *Najoom* as an *instructor*, rather than its ancestor node's tagname *section_listing*.

For Q_{42}, XReal can get a better result type than INL and SDI, which illustrates that the precision of our methods becomes lower when the statistic information only comes from a few relevant nodes. This is because five *individual* nodes and three *supervision* nodes are distributed over the different *title* nodes where only one *title* node contains the two terms *individual* and *supervision* at the same time. In this case, INL and SDI may suggest a node type at the upper level as the result type. However, if we delete the term *individual* from the specific *title* node, XReal still predicts the same result type - *title*. Obviously, no *title* node satisfies the query Q_{42}. Therefore, XReal may suggest wrong result types because it does not consider the structural correlations between the given terms.

From the results over DBLP in Table 4, we find that XReal, INL, and SDI can produce the same suggestions for the five keyword queries. This is because the structure of DBLP dataset is flat. Therefore, the given keyword queries are very easy to be used to locate the corresponding meaningful result types.

From the three sets of experimental results and discussions, we can find that SDI and INL can work well in the datasets with the complex or simple structures. That is to say, they can predict the result types with the more detailed semantics, which can guide users to find their interested information easily. However, XReal can only work well in the dataset with the relative simple structure.

6.3 Processing Time

To evaluate the efficiency of our algorithms, we measure the response times that all three approaches are required to find the top 3 promising result types for the queries in Table 3. As we can see, SDI can achieve greater efficiency than both XReal and INL in all conditions. In addition, we find that in most cases, XReal needs shorter time to process the corresponding keyword queries than INL.
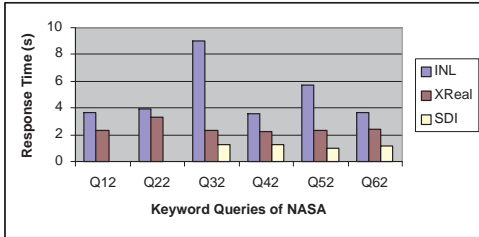

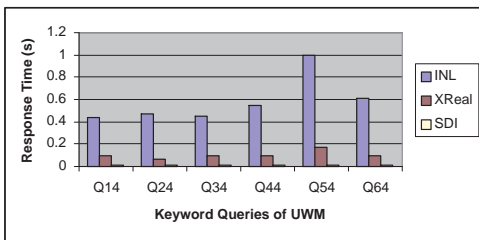**Figure 8: Response Time over NASA Dataset**

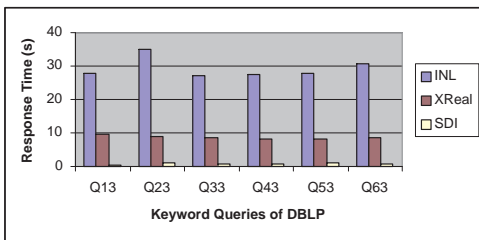**Figure 9: Response Time over UWM Dataset**

Figure 10: Response Time over DBLP Dataset

From Figure 8, we find that SDI can work out the promising result types in 0.02-1.26 seconds. To do the same tasks, INL requires 3.5-9 seconds and XReal requires 2.3-3.3 seconds or so. Therefore, SDI can reduce the response time greatly. In addition, we see that INL needs nearly 9 seconds to process Q_{32} because both keywords {photographic, magnitude} have very higher frequencies in NASA, which results in more computations than that of the other keywords. Therefore, we can say that INL is easy to be affected by the size of the dataset. However, XReal and SDI are relatively stable with regards to the size of the dataset. From Figure 9, we find that all three methods can determine the promising result types within 1 second. This is because UWM is a document with the small size. But SDI is still much better than XReal and INL. From Figure 10, we find that SDI can recommend the promising result types within 1 second. XReal needs 8-10 seconds to determine their result types. But INL performs very bad and needs 30 seconds or so to do the same tasks, which is not acceptable for the users to wait for the suggestions before they continue to do their keyword search.

From the experimental results and discussions, we can conclude that SDI is a stable approach to suggest the promising result types to the users. In addition, SDI can finish the task of recommendation within short time, which is practical and acceptable for the users to wait. Although XReal is also relatively stable, it will need up to 10 seconds sometimes, which is feasible but not acceptable in practice. For INL, however, the response times is very slow in most cases and it is affected by the sizes of the datasets, which is infeasible to be taken as a recommended approach in practice.

7. RELATED WORK

To make the returned results more meaningful, there are several approaches proposed in the literature: supervised result definition, ranking results, and predicting result types.

It is natural in many applications for domain experts or database administrators to provide guidances to keyword queries. [8, 11] considered the problem of identifying return nodes. Both of them require schema information. In addition, [8] requires a system administrator to split the schema graph into pieces, called Target Schema Segments (TSS) for search result presentation. Précis [11] requires users or a system administrator to specify a weight of each edge in the schema graph, and then each user needs to specify a degree constraint and cardinality constraint in the schema to determine the return nodes. [16] is helpful to infer or generate the guidance automatically. It considered the problem of differentiating search results for keyword search on structured data. By defining the differentiability of query results and quantifying the degree of difference, a limited number of valid features in a result can be derived, which can be used to maximally differentiate this result from the others.

Another approach is to allow different interpretations of the query, but strives to rank the more relevant results higher in the list of query results. XKeyword [8] proposed to rank the results according to the distance between different keywords in the document. XRANK [6] extended Google's PageRank to XML element level, to rank among the LCA results, which takes into account result specificity, keyword proximity and hyperlink awareness together. XSEarch [4] employed a ranking scheme that considers factors such as distance, term frequency, and document frequency. [20] proposed *coherency ranking*(CR), a domain- and database design-independent ranking method that is based on an extension of the concept of mutual information.

Yet another approach is to predict the most probable result types for the keyword query and return results only related to this interpretation. XSeek [14] proposed to generate the return nodes which can be explicitly inferred by keyword match pattern and the concept of entities in XML data. It requires to compute the results first and then derive the types of return nodes from the result set. Besides, this approach relies on the concept of entity and considers a node type t in DTD as an entity if t is "*"-annotated in DTD. [3] adopted the statistics of underlying XML data to identify the return node types where the statistic information comes from the number of nodes that contain the given keywords as either values or tag names in their subtrees and share the same path.

There are many works on generating meaningful query results for XML keyword search by inferring the semantics from various perspectives. [6, 13, 21, 14, 19] proposed to first retrieve the relevant nodes matching with every single

keyword from the data source and then compute LCAs or SLCAs of the nodes as the results to be returned. XRANK [6] and Schema-Free XQuery [13] developed stack-based algorithms to compute LCAs as the results. [21] introduced the Indexed Lookup Eager algorithm when the keywords appear with significantly different frequencies and the Scan Eager algorithm when the keywords have similar frequencies. [14, 15] took the similar approaches as [21]. But they focused on the discussions how to infer RETURN clauses for keyword queries w.r.t. XML data. [10] discovered that the filtering mechanism in MaxMatch algorithm in [15] was not sufficient and it committed the false positive problem and redundancy problem. To overcome the problems, [10] proposed a new filtering mechanism based on the concept of Relaxed Tighest Fragment (RTF) as the basic result type. [19] designed a MS approach to compute SLCAs for keyword queries in multiple ways. [12] took the Valuable LCA (VLCA) as results by avoiding the false positive and false negative of LCA and SLCA. [22] proposed an efficient algorithm called Indexed Stack to find answers to keyword queries based on XRank's semantics to LCA, named *Exclusive Lowest Common Ancestor ELCA*. Based on the ELCA semantics, the result of a keyword query is the set of nodes that contain at least one occurrence of all of the query keywords either in their labels or in the labels of their descendant nodes, after *excluding* the occurrences of the keywords in the subtrees that already contain at least one occurrence of all the query keywords.

In addition, there are other related works that process keyword search by integrating keywords into structured queries. [5] proposed a new query language XML-QL in which the structure of the query and keywords are separated. But the users are required to specify partial structures as predicates in XML-QL language. [13] introduced a method to embed keywords into XQuery to process keyword search.

Unlike most existing approaches, our method addresses the problem of automatically predicting promising search intention for XML keyword queries by considering the value and structural distributions of the data, rather than relying on subjective factors (e.g., users' interactive operations and assigned weights). Furthermore, it works well in the absence of database schemas.

8. CONCLUSIONS

In this paper, we have proposed an efficient and effective method to discover the promising result types for a keyword query over XML data sources, with the aim to help users disambiguate possible interpretations of the query. The proposed method is based on a new ranking method taking into consideration the query results for different interpretations and selecting one that has the maximum score. An efficient algorithm based on statistics synopsis build on the XML data has been developed which achieves high precisions with much faster execution speed. Experimental results have demonstrated that our proposed approach has good performance - high precision and low response time, for keyword queries over several real XML datasets.

9. REFERENCES

[1] Washington XML Data Repository. http://www.cs.washington.edu/research/xmldatasets/.

[2] A. T. Arampatzis and J. Kamps. A study of query length. In *SIGIR*, pages 811–812, 2008.

[3] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, pages 517–528, 2009.

[4] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.

[5] D. Florescu, D. Kossmann, and I. Manolescu. Integrating keyword search into XML query processing. *Computer Networks*, 33(1-6):119–135, 2000.

[6] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27, 2003.

[7] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

[8] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *ICDE*, pages 367–378, 2003.

[9] iProspect. iprospect natural seo keyword length study (nuvember 2004). Technical report, iProspect, 2004.

[10] L. Kong, R. Gilleron, and A. Lemay. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *EDBT*, pages 815–826, 2009.

[11] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Précis: The essence of a query answer. In *ICDE*, pages 69–78, 2006.

[12] G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.

[13] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.

[14] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[15] Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.

[16] Z. Liu, P. Sun, and Y. Chen. Structured search result differentiation. *PVLDB*, 2(1):313–324, 2009.

[17] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for xml data graphs. In *VLDB*, pages 466–477, 2002.

[18] N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Selectivity estimation for xml twigs. In *ICDE*, pages 264–275, 2004.

[19] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[20] A. Termehchy and M. Winslett. Effective, design-independent xml keyword search. In *CIKM*, pages 107–116, 2009.

[21] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 537–538, 2005.

[22] Y. Xu and Y. Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.