Processing XML Queries with Structural and Full-Text Constraints

YA-HUI CHANG, CHENG-YI WU AND CHENG-CHENG LO
Department of Computer Science and Engineering
National Taiwan Ocean University
Keelung, 202 Taiwan
E-mail: yahui@ntou.edu.tw

Efficient query processing on XML data is an important task for querying the data web. In this paper, we consider the XML query which can be represented as a query tree with twig patterns, and also consists of complex full-text constraints. Two approaches are proposed. The structure-first approach will first identify the elements which satisfy the tag constraint, and then process the full-text constraint on the terms represented within each element. The satisfied elements will be combined to meet the complete twig constraints. On the other hand, the keyword-first approach will first identify the elements which represent the required keywords, and then return the elements which satisfy the given full-text predicates and structural constraints. We demonstrate, via an extensive experimental study, that the two approaches have their own merits.

Keywords: XML, query processing, structural constraints, full-text predicates, keyword-based search

1. INTRODUCTION

As the XML (eXtensible Markup Language) technology emerged as the de facto standard for information sharing on the World-Wide-Web (WWW) and for data exchange in e-business, XML data management and query processing have attracted a lot of attention from the academic and business communities.

In general, the nested structure of an XML document is captured by a tree model, so XML queries can be specified based on path expressions to navigate the complex structure of XML data, as seen in the XQuery or XPath query languages. On the other hand, researchers also advocate the style of keyword-based search against XML documents, since it provides a more friendly user environment. Moreover, complex full-text predicates are needed to meet the expressivity demands of increasingly sophisticated XML search applications. Therefore, XQuery and XPath Full-Text (XQFT), an upcoming W3C standard query language for XML data [1], is proposed as an extension of XPath and XQuery to allow using full-text predicates. Consider the following sample query, which is posed against the sample XML tree in Fig. 1:

```
for $p$ in /catalog/item
  where $p$/description ftcontains ("database" ftand "design" ordered) and
  $p$/name ftcontains ("Peter" ftand "Rob" ordered)
return $p$
```

Received August 20, 2010; revised December 30, 2010; accepted February 23, 2011.
Communicated by Tyng-Luh Liu.

221
This query returns a certain item, whose description and name should satisfy the full-text constraints. In the query, the path expressions “catalog/item”, “catalog/item//name” form a twig pattern, which represent the structural constraint on the retrieved data. On the other hand, the operator `ftcontains` is used to enforce the contents of elements to include specific terms. Particularly, the “ordered” expression requires the terms to appear in a certain order. In this query, “database” should occur before “design”, and “Peter” should occur before “Rob”. Note that there are two possible semantics. One requires that the pair of keywords which satisfy the full-text constraint to be represented directly within the content of the same element. The other allows the keywords to be represented indirectly. That is, they could be represented within the content of the descendent elements. Since the latter one incurs more challenges than the former one, we will consider this semantic in this paper.

As seen in literature, there are mainly two tracks of research efforts on processing XML queries. The first one focuses on processing the structural constraints imposed by the path expressions specified in the query [2-12]. They usually first identify the elements which meet the tag or path requirements specified in the query, and apply specially-designed encoding schemes, algorithms, or data structures, e.g., indices or stacks, to expedite the combining process. The second track provides keyword-based querying, where users do not need to explicitly restrict the structure, but the system will ensure that the returning data satisfy the structure of the queried XML document [13-17]. This type of researches usually applies the techniques seen in the information retrieval (IR) field, e.g., inverted lists or ranking schemes.

In this paper, we will address the issues of processing structural and complex full-text constraints on XML data, which is an important task to retrieve information from the data web. Two approaches are proposed to tackle this problem. The first one is called the structure-first approach, which mainly follows the first line of the research direction which was just introduced. That is, we will first identify the elements which satisfy the tag constraint. However, before combining them to meet the whole structural constraints, we will
examine if these elements satisfy the given full-text constraints. On the contrary, the second one is called the keyword-first approach. Similar to the second line of researches, it will first identify the elements which satisfy the keyword constraint, followed by processing the full-text constraints. However, we will ensure that the answers also satisfy the structural constraint specified in the query at the last step. These two approaches are not only different in the processing sequence, but also in the underlying data structures and algorithms. We will implement these two approaches and compare their performance.

In summary, the main contributions of this study are as follows,

1. We propose a structure-first approach to process an XQuery with structural and complex full-text constraints. It follows the framework of the traditional approaches which process the twig constraint. However, we extend the data structure and design several additional functions to process the full-text predicates.
2. We also propose a keyword-first approach to process the same kind of queries. It follows the framework of processing keyword-based querying. We also extend it to assure that the final answers satisfy the structural constraints.
3. We conduct a series of experiments to evaluate the performance of the two proposed approaches. Evaluation results show that they have their own merits.

The remaining of this paper is organized as follows. We first compare related researches in section 2, and introduce the underlying data model, the query representation, and some basic terms in section 3. We then describe the two approaches. The structure-first approach is explained in section 4, and the keyword-first approach is discussed in section 5. We have performed a series of experiments to compare the two approaches. The experimental results will be analyzed in section 6. Finally, conclusion and future works are discussed in section 7.

2. RELATED WORK

There have been a lot of studies on processing structural constraints of XML queries. A common approach is to encode elements for quickly identifying the structural relationship between them. Elements will be first combined into a set of root-to-leaf paths, which are then merged to formulate the complete twig patterns [2, 5]. To reduce the amount of intermediate results, the holistic approach is proposed to use a chain of linked stacks which only represent the elements belonging to part of the correct answers [4]. However, it is an optimal approach for queries with only descendent steps. The Twig2Stack approach then extends the structure of stacks to efficiently process both child steps and descendent steps [10].

The above-mentioned approaches all encode elements using \((start, end, level)\), and there are other encoding schemes being proposed as well, e.g., encoding an XML document as a sequence [7], based on extended Dewey encoding [9], or based on the prufer sequence [8, 11]. In this paper, we utilize the TJFAST algorithm [9] to help processing structural joins, since it is shown to be very efficient, and applies an encoding scheme similar to those used in supporting full-text search on XML data, such as the system in [13, 18].
There are also many works on supporting the type of keyword-based querying for XML data. An important issue first being studied is to efficiently compute the scores of XML elements [15], and to assign appropriate scoring functions for XML data [14, 18]. A novel algorithm to return the Top-K answers is also proposed in [17], and there exist continuous efforts in the series of INEX workshops, which try to increase the precision and recall for keyword-based querying on XML data [19]. Moreover, some researchers investigate how to efficiently process complex full-text constraints [13]. Note that this type of works does not require users to impose structural constraints. We will extend the work of [13], to identify the exact answers which satisfy both the structural and full-text constraints specified by the users.

Another field of researches relevant to our work is the techniques of XML query optimization. The researchers in [20] consider the cost-based query optimization for XML data. They also propose a set of APIs, so that they could utilize any other existing algorithms for processing structural constraints. The researchers in [21] discuss how to perform XML query optimization when side effects are allowed. Particularly, they discuss when the rewriting rules could be still applied. The researchers in [22] propose to perform query optimization during run-time, by using the sampling-based techniques to decide which operator to execute next. We will review different execution strategies in this paper and discuss their performance under different scenarios, which forms the basis of more complex optimization works.

3. PRELIMINARIES

In this section, we discuss how to represent the XML data and a given query. We also define several basic terms and formally describe the problem to solve and the two approaches.

3.1 XML Modeling

For easy explanation, the XML document is represented as a rooted labeled tree, where each node corresponds to an element and the edge represents the nesting relationship between elements. Contexts are represented as child nodes of the associated nodes. The sample XML tree in Fig. 1 represents the information about three books.

To support efficient processing of structural and full-text constraints, we apply two encodings in the XML modeling. First, for each keyword in the context, we assign global positions across elements. For example, the first “Database” under the leftmost title element is encoded as 1, while the first “Database” under the leftmost description element is encoded as 14. This encoding is mainly to support the processing of full-text constraints.

Secondly, to quickly determine the structural relationship between two elements, each of which is associated with the extended Dewey encoding [9]. This encoding scheme basically has the main properties of the original Dewey encoding. That is, the ancestor/descendent or the parent/child relationship between two elements can be determined by comparing their prefixes. For example, element 1.1 is the ancestor of the element 1.1.1 since the former is a proper prefix of the latter. On the other hand, since this encoding shows all of its ancestors, the LCA (lowest common ancestor) between two elements can be obtained.
by computing their common prefixes. For example, the LCA of elements 1.1.1 and 1.1.2 is 1.1. This is an important operation for supporting keyword-based querying, as will be shown later.

A unique property of the extended Dewey encoding applied in our framework is that it can be easily transformed to a labeled path. For example, encoding 1.1.1 can be transformed to the path “/catalog/item/title”. Please see the original paper for the complete encoding mechanism.

3.2 Query Tree

We give one more sample XML query (in XQFT) in this subsection, which will be used as the running example in this paper. This query is expressed based on the XML tree in Fig. 1. It poses two full-text predicates on the content of the element “/catalog/item/description”, which requires an ordering between “database” and “design”, with the distance at least “2” between them. It farther poses an ordering constraint on the content of the element “/catalog/item/name”.

\[ Q_1 \]

\[
\text{for } \texttt{Sp in /catalog/item} \\
\text{where } \texttt{Sp/description ftcontains (“database” ftand “design” ordered) ftand (“database” ftand “design” distance at least 2 words) and Sp/name ftcontains (“Peter” ftand “Rob” ordered)} \\
\text{return } \texttt{Sp}
\]

To clearly show the specified structural and full-text constraint, a query will be transformed into a query tree as illustrated in Fig. 2, which corresponds to the sample query \( Q_1 \). The query tree is based on all the path expressions specified in the query, where the component elements are illustrated by nodes and the location steps are denoted by edges.

![Fig. 2. The sample query tree.](image-url)
Specifically, the “||” edge corresponds to the location step “//”, which represents the AD (ancestor-descendent) relationship between elements, and the “|” edge corresponds to the location step “/”, which represents the PC (parent-child) relationship. On the other hand, the full-text constraint associated with a path expression, is represented as a child node of the path’s end-point, and denoted by the dashed line. We consider two types of complex full-text constraints in this paper. The “ordered” constraint is represented by a pair of square brackets, and the “distance” constraint is represented by a pair of parenthesis. Double circles refer to elements whose contents are to be returned.

3.3 Problem Definition

We provide several basic definitions to formally describe the problem to solve in this paper and the proposed two approaches. The XML tree in Fig. 1 and the query in Fig. 2 will be used as the running example.

Definition 1 \( T_{\text{match}}(T) \) (\( Tm(T) \) in short): an element is a \( T_{\text{match}}(T) \) if its tag satisfies the tag constraint \( T \) specified in the query tree.

Definition 2 \( P_{\text{match}}(P) \) (\( Pm(P) \) in short): an element is a \( P_{\text{match}}(P) \) if its labeled path satisfies the path constraint \( P \) specified in the query tree.

Definition 3 \( K_{\text{match}}(K) \) (\( Km(K) \) in short): an element is a \( K_{\text{match}}(K) \) if its content immediately consists of the required keyword \( K \) specified in the query tree.

Definition 4 \( FT_{\text{match}}(N) \) (\( FTm(N) \) in short): an element is an \( FT_{\text{match}}(N) \) if it is a \( Tm(N) \) and satisfies all the required full-text predicates specified for the node \( N \) in the query tree.

Definition 5 Match tree: a set of elements in the XML tree is a match tree, if each component element is either a \( P_{\text{match}} \) or an \( FT_{\text{match}} \), and they satisfy the whole structural constraint imposed by the query tree.

Definition 6 Answer: an element in a match tree is an answer if it satisfies the returning path.

Example 1: Elements 1.1.8, 1.2.8, and 1.3.4 are \( Tm(description) \). They are also \( Pm(catalog/item/description) \). On the other hand, the following six elements are \( Km(database): 1.1.1, 1.1.8, 1.2.1, 1.2.8, 1.3.1, and 1.3.4, but only two of them, i.e., 1.1.8 and 1.3.4 are \( FTm(description) \). Note element 1.2.8 is not qualified since the distance between “database” and “design” is only one. Finally, we can get two match trees: \{1.1.1, 1.1.2.1, 1.1.8\} and \{1.1.3, 1.3.2.1, 1.3.4\}, and elements 1.1 and 1.3 are answers.

Given an XQuery with structural and complex full-text constraints, the problem to solve in this paper is to identify the answer from an XML tree. The structure-first architecture is shown in Fig. 3 (a). It will first identify \( T_{\text{matches}} \) for each leaf node of the query tree, find the \( FT_{\text{matches}} \) among them, and pick the \( P_{\text{matches}} \) to form the match tree.
On the other hand, the architecture of the keyword-first approach is shown in Fig. 3 (b). The system will first identify \( K\_matches \) for each keyword specified in the query tree. Based on the \( K\_matches \), we find their LCAs (lowest common ancestors), and identify all the nodes which satisfy the full-text constraint. However, we will further represent the \( FT\_matches \) in the query tree, and combine them to find the final match tree. In the following sections, we will describe these two approaches in detail.

**4. THE STRUCTURE-FIRST APPROACH**

In this section, we describe the structure-first approach. We will first introduce the underlying data structures, and then explain the component algorithms.

**4.1 The Data Structure**

An example of the main data structure applied by this approach is shown in Fig. 4, which is the extension of the query tree, as shown in Fig. 2. Recall that the tree structure represents the structural constraint imposed by the given query. We represent the \( T\_matches \) for each leaf node of the query tree as a stream of the associated node, which are sorted based on the preorder sequence. For each \( T\_match \), we further represent the corresponding keywords along with their positions to support the processing of full-text constraints. They are named as the term lists, and are sorted based on the document order. In Fig. 4, we can see that the stream associated with the node description has three \( T\_matches \), each of which has a term list to represent the terms within the content of the element. Note that the terms represented directly within the element or indirectly within the context of the descendants will be all in the stream.

To efficiently instantiate this data structure, the XML document will be first preprocessed and represented as a set of tuples with four types of information: (extended Dewey encoding, tag-name, keyword, position). An index with the tag-name as the key is also built. When processing a query, the Retrieving Data Module, as shown in Fig. 3 (a), will utilize the index to identify the required \( T\_matches \). If we want to support querying on the heterogenous data web, the document identifier will be also included, and those elements in the stream will be sorted first based on the document identifier and then on the extended Dewey encoding.
Fig. 4. The query tree with streams and termlists.

Fig. 5. The main algorithm of the structure-first approach.
The main algorithm of the structure-first approach is listed in Fig. 5. The input query is first parsed and represented as a query tree, and the streams and term lists are instantiated as described above (L1-L6). The following statements of the algorithm consist of a nested for loop (L7-L20). The outer for loop will process each leaf node of the query tree, and the inner for loop will then examine each element, i.e., \( T_{match} \), in the associated stream. It will determine if the corresponding term list represents the required keywords and satisfy the specified full-text predicates, and remove the \( T_{match} \) from the stream if it is not satisfied. Since it is easier to process the ordering constraint than the distance constraint, we will first process the ordering constraint in both the structure-first and the keyword-first approach. An example will be given below and the corresponding algorithms will be explained in the next subsection. Note that at this stage (L21), the elements left in the stream of the query tree will satisfy both the tag constraint and the full-text predicate, and will be qualified as \( FT_{matches} \).

**Example 2:** Consider the sample XML tree (Fig. 1) and the sample query (Fig. 2). We will first identify the five elements in \( Tm(name) \) and the three elements in \( Tm(description) \), as shown in the two streams of Fig. 4. We then apply the full-text constraint on the ordering of the terms “Peter” and “Rob”, and get \( FTm(name) = \{1.1.2.1, 1.2.2.1, 1.3.2.1\} \). Similarly applying the constraint on the ordering and distance of the terms “database” and “design”, we can get \( FTm(description) = \{1.1.8, 1.3.4\} \).

Next, we will need to combine the \( FT_{matches} \) as a set of match trees. The algorithm we applied is the TJFAST algorithm [9]. This algorithm will examine the elements in the stream in the pre-order sequence. It first determines if the element satisfies the path constraint, by transforming the extended Dewey encoding to the labeled path, as discussed in section 3. This labeled path is then matched with the path of the node in the query tree. Note that the query path might consist of the descendant step “//”. It will be represented by the regular expression to allow matching more than one element. After filtering out elements with unqualified paths, the TJFAST algorithm then tries to identify a match tree by determining if the current elements in each stream have a common ancestor in the branch node. We will only show a simple example here to illustrate the main idea, and refer the interested readers to the original paper.

**Example 3:** Continue Example 2 and consider Fig. 6. The tree shown on the left is a simplified version of the original sample XML tree in Fig. 1, and the query tree shown on the right represents the remaining \( FT_{matches} \) in the two streams. Consider the first elements respectively in the two streams, \( i.e., 1.1.2.1 \) and \( 1.1.8 \). Observe that these two elements have a common ancestor 1.1 corresponding to the queried branching node “item” in the sample XML tree. Therefore, a match tree is found, as depicted by the dashed line. Another match tree, \( i.e., \{1, 1.3, 1.3.2.1, 1.3.4\} \) will be also identified.

### 4.3 Processing Full-Text Constraints

We now discuss how to process full-text constraints in the structure-first approach. In
this paper, we consider the \textit{ordered} constraint and the \textit{distance} constraint. Algorithm \texttt{OrderHandle} is first shown in Fig. 7. The parameter \textit{OC} is an ordered constraint with the form \([\text{term}_1, \text{term}_2]\). For each element in the stream, this algorithm will sequentially examine the keywords represented in the associated termlist. If \textit{term}_1 and \textit{term}_2 appear in the termlist in order, the algorithm will return \texttt{TRUE} and this element will be kept in the stream. Otherwise, this element will be deleted.

We then discuss how to process the distance constraint. The complete algorithm is listed in Fig. 8. The parameter \textit{DC} represents a distance constraint and has the form \((T1, T2, \text{operator}, \text{number})\). The basic idea is to use the position of \textit{T1} and \textit{number} to calculate a range which \textit{T2} could appear in. First, we deal with the situation when the operator is \textit{"<"}. The range is first calculated and represented by the variables \textit{LeftBoundary} and \textit{RightBoundary} (L10-L11). In this case, \textit{T2} should appear within this range, as depicted in Fig. 9.
Algorithm DistanceHandle
Input: TermList, DC
Output: True/False
1: T1Cursor = T2Cursor = TermList;
2: T1Position = GetPosition(T1Cursor, DC → T1);
3: T2Position = GetPosition(T2Cursor, DC → T2);
4: if T1Position == 0 or T2Position == 0 then
5:   return False;
6: end if
7: switch(DC → operator)
8: case "<":
9:   while T1Position != 0 and T2Position != 0 do
10:     LeftBoundary = T1Position - DC → number;
11:     RightBoundary = T1Position + DC → number;
12:     if T2Position < LeftBoundary then
13:       T2Position = GetPosition(T2Cursor, DC → T2);
14:     else if T2Position > RightBoundary then
15:       T1Position = GetPosition(T1Cursor, DC → T1);
16:     else
17:       return true;
18:     end if
19:   end while
20: end if
21: case ">":
22: while T1Position != 0 and T2Position != 0 do
23:   T1lastPosition = GetlastPosition(T1Cursor, DC → T1);
24:   T2lastPosition = GetlastPosition(T2Cursor, DC → T2);
25:   if T1Position != 0 and T2Position != 0 then
26:     if T2Position < T1Position + DC → number or T2lastPosition > T1lastPosition + DC → number then
27:       return True;
28:     end if
29:   end if
30: end if
31: case "=":
32: while T1Position != 0 and T2Position != 0 do
33:   LeftBoundary = T1Position - DC → number;
34:   RightBoundary = T1Position + DC → number;
35:   if T2Position == LeftBoundary or T2Position == RightBoundary then
36:     return true;
37: else if T2Position < LeftBoundary then
38:     T2Position = GetPosition(T2Cursor, DC → T2);
39:     continue;
40: else if T2Position > RightBoundary then
41:     T1Position = GetPosition(T1Cursor, DC → T1);
42:     continue;
43: else
44:     T3Cursor = T1Cursor;
45:     while T2Position > LeftBoundary do
46:       if T2Position == LeftBoundary or T2Position == RightBoundary then
47:         return true;
48:       end if
49:     T3Position = GetPosition(T3Cursor, DC → T1);
50:     LeftBoundary = T3Position - DC → number;
51:     RightBoundary = T3Position + DC → number;
52:     end while
53:     LeftBoundary = T1Position - DC → number;
54:     RightBoundary = T1Position + DC → number;
55:     while T2Position < RightBoundary do
56:       T2Position = GetPosition(T2Cursor, DC → T2);
57:       if T2Position == RightBoundary then
58:         return true;
59:       else
60:         continue;
61:       end if
62:     end while
63:     T1Position = GetPosition(T1Cursor, DC → T1);
64:     continue;
65: end if
66: end while

Fig. 8. Algorithm DistanceHandle.

(a). However, if T2 appears at the position (1), we will try to determine if the next T2 appears in the desired range, so we move T2Cursor forward (L12-L14). Otherwise, when T2 appears at the position (3), we will move forward T1Cursor to form the next feasible range.

The second case concerns the operator “>” (L23-L30). Since T1 and T2 should be separated as far as possible, we only consider the extreme cases. In the upper graph of Fig. 9 (b), we use the first position of T1 to form the range, and examine if the first and the last
occurrences of $T_2$ appear outside of the range. We also use the last position of $T_1$ to form another range, as depicted in the lower graph of Fig. 9 (b).

Finally, we discuss the case of the “=” operator. Similar to processing the “<” operator, we first use $T_1$ to calculate the possible range. If $T_2$ is not exactly at the boundary, we move forward the cursor which is behind (L33-L42). The more complex situation happens when $T_2$ is within the range. We need to determine if $T_2$ matches another occurrence of $T_1$ within the range. We therefore use another cursor $T_3$ to compare with $T_2$. If no $T_3$ within this range can match with $T_2$, we will continue to examine another occurrences of $T_1$.

4.4 Time Complexity

To analyze the time complexity of this approach, we suppose that the query consists of $t$ tags and $k$ keywords. It will first take $O(t + k)$ time to build the query tree. This is quite small and will be ignored later. Then, we need to load the data into the query tree. The size of the streams will be bounded by $t \times |T_{\text{match}}|$, where $|T_{\text{match}}|$ is the average size of $T_{\text{matches}}$. However, since the termlist associated with a $T_{\text{match}}$ needs to represent all the underneath keywords, it might include the whole content of the XML document in the worst case. Together, the sizes of the streams and the termlists are bounded by the size of the whole XML document $s$, and therefore this step takes $O(s)$ I/O time. When processing the full-text constraint, it sequentially examines all the termlists, and takes $O(s)$ CPU time. Finally, the time complexity of the TJFAST algorithm is linear in the sum of inputs and outputs. Therefore, this approach totally takes $O(s)$ I/O time and $O(s + t \times |FT_{\text{match}}| + a)$ CPU time, where $a$ is the amount of answers.

5. THE KEYWORD-FIRST APPROACH

In this section, the keyword-first approach is introduced. It will first identify the elements which satisfy the keyword constraints, process the full-text constraints, and try to meet the structural constraints. We will first explain the underlying data structures, and then discuss the component algorithms.

5.1 The Data Structure

For an input query, the Retrieving Data module, as seen in Fig. 3 (b), will first identify the elements that satisfy the keyword constraint. To facilitate such retrieval process,
the inverted lists for each keyword are constructed beforehand. When processing a keyword constraint specified in a query, the matched nodes (denoted by extended Dewey encodings) and the global positions will be returned and represented as an SCU (Smallest Containing Unit) [13]. An SCU consists of a list of items, where each item represents an element and the position which matches the pattern (a keyword or a full-text constraint).

For example, in Figs. 10 (a) and (b), we can see the SCU tables for the keywords “database” and “design” against the XML tree in Fig. 1, where the position is represented by attaching the global position to the extended Dewey encoding of its parent element.

Note that in contrast to the previous approach, an SCU only consists of the elements immediately containing the keyword, to make the table small. Also, the items in an SCU table are sorted based on the postorder of elements. This is to support the bottom-up manner of processing full-text constraints, as will be explained later.
5.2 The Algorithms

The main algorithm of the keyword-first approach is shown in Fig. 11. As in the structure-first approach, it will first build the query tree and process each leaf node in sequence. Then, for each full-text constraint which consists of the keywords K1 and K2, we will create the corresponding SCU tables (L6-L15). Since a keyword may appear in different constraints, we use a hash table to point to the created SCU tables for possible future usage. For the two SCU tables, we invoke the DoingLCA algorithm [13] to obtain those LCAs which have both keywords K1 and K2 within its (or descendants’) contents. Note that the LCA of a pair of elements can be easily obtained by computing the longest common prefixes from the encodings. For example, the LCA of 1.1.1.1 and 1.1.1.3 is 1.1.1. The algorithm in [13] basically uses a specially-designed stack to make the input and the output both in the post-order sequence. Interested readers please see the original paper for the complete listing of this algorithm.

Example 4: We again use the sample XML tree in Fig. 1 and the sample query in Fig. 2 to show the process of the keyword-first approach. Consider the pair of full-text predicates,
i.e., \([\text{database}, \text{design}]\) and \((\text{database}, \text{design}, \geq, 2)\), associated with the leaf node “description”. Based on the first predicate, we create the SCU tables for the two component keywords “database” and “design”, i.e., \(K_m(\text{database})\) and \(K_m(\text{design})\), as shown in Figs. 10 (a)-(b). The DoingLCA algorithm will then operate on these two SCU tables to obtain the corresponding LCAs, as shown in Fig. 10 (c).

Algorithm DoingPredicate [13] is then invoked to identify those elements which satisfy the full-text constraint. We also refer the interested readers to the original paper. Basically, an LCA consists of the required keywords, but we need to determine if it satisfies the given full-text constraint. Note that in this approach, it is comparably easy to check the full-text predicates ordered and distance, since we can directly retrieve the matched positions for the keywords from the SCU, and a simple arithmetic calculation will suffice. If an LCA does not satisfy the full-text constraint, the matched position will be sent to its nearest ancestor to check next. For example, as shown by the pointed arrow in Fig. 10, element 1.1 does not satisfy the first ordered constraint in the sample query, because the matched position for “database” (1.1.8.14) is bigger than the matched position for “design” (1.1.1.3). Therefore, these information will be propagated to its ancestor (element 1) for further checking.

After obtaining those elements which satisfy the full-text constraints, we will represent them as a stream of the currently processed node \(N\) in the query tree (L24). Algorithm DoingPmatch will be invoked to filter out those elements which do not have the correct labeled path as the path of \(N\). It is achieved by transforming its extended Dewey encoding to the labeled path, as discussed before.

Example 5: Continue Example 4. Observe that only seven LCAs among nine LCAs remain in the SCU table in Fig. 10 (d), since they represent the queried keywords “database” and “design” in the correct ordering. Note that the following distance constraint is also imposed on the same pair of keywords, so Algorithm DoingPredicate will continue to process that constraint against the same SCU table, and identify five satisfied elements as shown in Fig. 10 (e). Recall that the full-text constrains are specified with the path “catalog/item/description”. Applying Algorithm DoingPmatch on the five elements will identify two \(P\) matches among them, which are also \(FTm(\text{description})\), as shown in Fig. 10 (e).

At the final stage, we will invoke Algorithm TJFAST to find the match trees. However, note that the TJFAST algorithm requires the elements in the stream to be sorted in the preorder sequence, which is different from the ordering in SCU. Therefore, we need to represent them in the correct order before invoking TJFAST.

Example 6: To complete processing this sample query, we need to apply the same process stated in Examples 4-5 to the other full-text constraint \([\text{Peter}, \text{Rob}]\), and we can get \(FT(\text{name}) = \{1.1.2.1, 1.2.2.1, 1.3.2.1\}\). These qualified \(FT\) matches will be represented in the corresponding stream of the query tree, as depicted in Fig. 6 (b). Finally, the TJFAST algorithm is invoked to form the match tree and return the final answer, as in the structure-first approach.
5.3 Analysis of Time Complexity

We now analyze the time complexity of the keyword-first approach. Recall that a query is assumed to consist of $t$ tags and $k$ keywords, and the meanings of $T_{\text{match}}$ and $K_{\text{match}}$ can be seen in Definitions 1 and 3. As in the previous approach, the time to build the query tree is ignored. Next, it takes $O(k \times |K_{\text{match}}|)$ I/O time to sequentially read data from the inverted list. Note that the time complexity of Algorithms DoingLCA and DoingPredicate is both linear in the sum of the input and the output, as discussed in [13]. Therefore, Algorithm DoingLCA will take $O(k \times (|K_{\text{match}}| + L))$, where $L$ is the average number of LCAs being created for a pair of keyword constraints, and Algorithm DoingPredicate will take $O(k \times (L + |FT_{\text{match}}'|))$, where $FT_{\text{match}}'$ represents the element which satisfies the full-text constraint, but may not satisfy the tag constraint.

Finally, the time complexity of Algorithms DoingPmatch and TJFast is also linear in the sum of the input and the output, and it takes $O(t \times (|FT_{\text{match}}' + |FT_{\text{match}}|)) + O(t \times |FT_{\text{match}}'| + a)$, where $a$ is the amount of answers. Therefore, the total time complexity is summarized as: $O(k \times |K_{\text{match}}|) \text{ I/O time} + O(k \times |K_{\text{match}}| + 2kL + (k + t) \times |FT_{\text{match}}'| + 2t \times |FT_{\text{match}}'| + a) \text{ CUP time.}$

Table 1. The test queries.

<table>
<thead>
<tr>
<th>No</th>
<th>Query Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0</td>
<td>/site/item//description/text ftcontains (&quot;master&quot; ftand &quot;master&quot; ordered) and ./mailbox//from ftcontains (&quot;mehrdad&quot; ftand &quot;mehrdad&quot; ordered)]</td>
</tr>
<tr>
<td>Q1</td>
<td>/site/regions/asia/item//emph ftcontains (&quot;master&quot; ftand &quot;attend&quot; distance at most 500 words) and ./bold ftcontains (&quot;ship&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q2</td>
<td>/site/regions/asia/item//payment ftcontains (&quot;master&quot; ftand &quot;attend&quot; distance at most 500 words) and ./shipping ftcontains (&quot;ship&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q3</td>
<td>/site/regions/asia/item//payment ftcontains (&quot;master&quot; ftand &quot;attend&quot; distance at most 500 words) and ./shipping ftcontains (&quot;ship&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q4</td>
<td>/dblp/inproceedings//booktitle ftcontains (&quot;system&quot; ftand &quot;system&quot; ordered) and ./title ftcontains (&quot;program&quot; ftand &quot;program&quot; ordered)]</td>
</tr>
<tr>
<td>Q5</td>
<td>/dblp/inproceedings//booktitle ftcontains (&quot;system&quot; ftand &quot;system&quot; ordered) and ./title ftcontains (&quot;program&quot; ftand &quot;program&quot; ordered)]</td>
</tr>
<tr>
<td>Q6</td>
<td>/dblp/inproceedings//booktitle ftcontains (&quot;system&quot; ftand &quot;system&quot; ordered) and ./title ftcontains (&quot;program&quot; ftand &quot;program&quot; ordered)]</td>
</tr>
<tr>
<td>Q7</td>
<td>/site/regions/asia/item//description//text fcontains (&quot;master&quot; ftand &quot;attend&quot; distance at most 500 words) and ./shipping ftcontains (&quot;ship&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q8</td>
<td>/site/regions/asia/item//description//text fcontains (&quot;master&quot; ftand &quot;attend&quot; distance at most 500 words) and ./shipping ftcontains (&quot;ship&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q9</td>
<td>/site/regions/asia/item//description//text fcontains (&quot;master&quot; ftand &quot;attend&quot; distance at least 5 words) and ./shipping ftcontains (&quot;see&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q10</td>
<td>/site/regions/asia/item//description//text fcontains (&quot;master&quot; ftand &quot;attend&quot; distance at least 50 words) and ./shipping ftcontains (&quot;see&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
<tr>
<td>Q11</td>
<td>/site/regions/asia/item//description//text fcontains (&quot;master&quot; ftand &quot;attend&quot; distance at least 500 words) and ./shipping ftcontains (&quot;see&quot; ftand &quot;see&quot; ordered)]</td>
</tr>
</tbody>
</table>
6. PERFORMANCE EVALUATION

We have designed several experiments to evaluate the performance of the proposed two approaches. All the experiments are performed on a personal computer with an Intel Core 2 1.9 GHz CPU and 1.5 GB memory, with the Microsoft Windows XP operating system. We will compare the performance of the structure-first approach (called SF) and the keyword-first approach (called KF). In addition, we have designed an improved version of the KF approach, which will use the level information of the query tree to filter out unsatisfied LCAs to reduce the amount of the input of Algorithm DoingPredicate. It will be called the KLF approach. The test queries are summarized in Table 1. To save the space, we use the XPath-like syntax.

6.1 Effects of Data Sizes

We have performed a scalability test on the size of the XML document. The first dataset we use is the DBLP collection [23], with sizes ranging from 10 MB to 50 MB. Query Q4 in Table 1 is the test query, which consists of a twig pattern with the ordered constraint on the two leaves. The execution time is shown on Fig. 12 (a). We can see that the amounts of data sizes have a linear effect on the three approaches. Particularly, the KLF approach performs best. Also, the SF approach performs worst because the elements which satisfy the tag constraint (T_match) are a lot more than the elements which satisfy the keyword constraint (K_match), as seen in Fig. 12 (b). We also perform a similar experiment on the XMark dataset [24] using the query Q0, and the results are the same, as shown in Fig. 13.
6.2 Effects of Frequencies

As observed in the previous experiments, the amount of retrieved data has a direct effect on the execution time. We will perform a more detailed analysis on this issue in this subsection. The first experiment is on the frequencies of tags. We use the XMark data set with the 50 MB size. The frequencies of several representative tags are listed in Table 2. We use three queries to run the experiments, which differ on the two leaf nodes. Q1 consists of the tags \textit{emph} and \textit{bold}, Q2 consists of the tags \textit{emph} and \textit{shipping}, and Q3 consists of the tags \textit{payment} and \textit{shipping}. As shown in Fig. 14 (a), we can see that the performance of the KF and KLF approaches are not affected at all, while the SF approach is obviously affected, since the tag frequencies will affect the amount of $T_{match}$.

The second experiment is on the frequencies of keywords. We use the DBLP data set with the 50 MB size. The frequencies of several keywords are listed in Table 3. We similarly apply three queries to run the experiment, where Q4 consists of the keyword \textit{system} and \textit{program}, Q5 consists of the keywords \textit{system} and \textit{support}, and Q6 consists of the keywords \textit{system} and \textit{language}. Fig. 14 (b) shows the execution time of the SF and the KF approaches. However, we can see that the performance does not really reflect the keyword frequencies. The reason is that although the keyword frequencies affect the amount of $K_{match}$, but their distribution will also affect the computation of LCAs, which requires heavy computation efforts. This factor will be studied in the next subsection.

### Table 2. Tag Frequencies in XMark.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>emph</td>
<td>16321</td>
</tr>
<tr>
<td>bold</td>
<td>16562</td>
</tr>
<tr>
<td>shipping</td>
<td>19333</td>
</tr>
<tr>
<td>payment</td>
<td>19333</td>
</tr>
</tbody>
</table>

### Table 3. Keyword Frequencies in DBLP.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>6741</td>
</tr>
<tr>
<td>support</td>
<td>954</td>
</tr>
<tr>
<td>program</td>
<td>1260</td>
</tr>
<tr>
<td>language</td>
<td>2552</td>
</tr>
</tbody>
</table>

6.3 Complex Full-Text Constraints

In this set of experiment, we let queries Q7, Q8, Q9 consist of the same set of full-text constraints, including \textit{ordered} and \textit{distance} constraints, but let the paths of the two leaf nodes of the query tree differ. Here, the path of Q7, “//item/description/text//keyword” is the longest, and the path of Q9, “//item/description” is the shortest. The execution time is shown in Fig. 15 (a). We can see that the KF and the KLF approaches perform almost the same, but the execution time of the SF approach increases a lot. The reason is that in the SF approach, we let each element associated with all the keywords represented within
its own content or within its descendants. Therefore, the sizes of the termlists for Q9 are a lot more than those for Q8 and Q7, and obviously affect its execution time.

In the next experiment, the three queries have the same tag pattern, but their full-text predicates differ on the distance predicates, where Q10 is “≥ 5 words”, Q11 is “≥ 50 words”, and Q12 is “≥ 500 words”. We can see that requiring the keywords more distant will increase the computation of propagating to the ancestor nodes and therefore make the KF and the KLF approaches require more execution time.

6.4 Discussion

We compare the two approaches in different aspects. First, in terms of the system architecture (Figs. 3 (a)-(b)) and the underlying data structure (Figs. 4 and 10), the SF approach seems a more natural and simple one. It retrieves all the T_matches along with the terms represented within its contents, and iterates over the termlists to determine if an element satisfies the full-text constraints. On the other hand, the KF approach involves more steps. After retrieving the K_matches, it should first identify the LCA which contains the required pair of keywords. Next, although it is very easy to determine if an LCA satisfies the full-text constraint or not, we might need to propagate the matched positions to the ancestor for further checking. At last, the survived elements need to be represented in the query tree for final combining. We can see that the time complexity of the KF approach is affected by more factors than the SF approach, as discussed in sections 4.4 and 5.3.

However, empirically, the KF approach is usually more efficient than the SF approach, except in the case that the full-text constraint is quite complex (Fig. 15 (b)). The reason is that the execution time is affected by the amount of retrieved data to a large extent. Since an XML document usually has more keywords than tags, tag frequencies are normally higher than key frequencies. Moreover, the SF approach requires all the keywords underneath the element to be loaded into the query tree. Therefore, the structure-first approach tends to perform worse than the keyword-first approach.

7. CONCLUSION

We discuss how to process an XML query which presents twig patterns and consists
of complex full-text constraints in this paper. The structure-first approach and the key-
word-first approach are proposed. We have implemented the two approaches and per-
fomed a series of experiments to compare their performance. In general, the KLF system,
i.e., the keyword-first approach which applies the level information earlier, performs best.
However, there are many factors which affect their performance. In the future, we plan to
investigate the optimization technique, so that the most efficient approach will be auto-
matically chosen. We will also consider other constructs of an XML query, such as value-
based join expressions or nested returning expressions, to support the full-fledge FLOWR
XQuery expressions.

REFERENCES

2. Q. Li and B. Moon, “Indexing and querying XML data for regular path expressions,”
in Proceedings of the 27th International Conference on Very Large Data Bases, 2001,
pp. 361-370.
3. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu,
“Structural joins: A primitive for efficient XML query pattern matching,” in Proceed-
ings of International Conference on Data Engineering, 2002, pp. 141-152.
matching,” in Proceedings of the ACM SIGMOD International Conference on Man-
joins on indexed XML documents,” in Proceedings of the 28th International Confer-
ence on Very Large Data Bases, 2002, pp. 263-274.
documents,” in Proceedings of the 29th International Conference on Very Large Data
querying XML data by tree structures,” in Proceedings of SIGMOD Conference,
2003, pp. 110-121.
288-299.
dewey: On efficient processing of XML twig pattern matching,” in Proceedings of the
31st International Conference on Very Large Data Bases, 2005, pp. 193-204.
“TwigStack: Bottom-up processing of generalized tree-pattern queries over XML
documents,” in Proceedings of the 32nd International Conference on Very Large Data
meets XML indexing and querying,” in Proceedings of the 33rd International Con-
ference on Very Large Data Bases, 2007, pp. 63-74.


**Ya-Hui Chang** (張雅惠) is an Associate Professor of Computer Science Department at National Taiwan Ocean University. She received the B.S. degree in Computer Science and Information Engineering from National Taiwan University, and the M.S. and Ph.D. degree in Computer Science from University of Maryland at College Park, U.S.A. Her research interests include database query processing, information integration, and XML techniques.
Cheng-Yi Wu (吳政儀) was born in Taipei, Taiwan, on October 8, 1981. He received the M.S. degree in Computer Science from National Taiwan Ocean University in 2009. His research interests include XML query processing, full-text constraint processing and Information retrieval techniques.

Cheng-Cheng Lo (羅誠正) received the M.S. in Computer Science Department from National Taiwan Ocean University in 2011 and B.S. degree in Department of Computer and Communication Engineering from National Kaohsiung First University of Science and Technology in 2008. His research interests include XML query optimization, XML query processing, web application and database management.