Algorithms for XML Tree Pattern Matching and Query Processing

Kamala Challa¹, E.Jhansi Rani²

¹Pursuing M.Tech(CS), Nalanda Institute of Engineering & Technology, Siddharth Nagar, Sattenapalli, Guntur., Affiliated to JNTUK, Kakinada, A.P., India.

²Asst. Professor, Department of Computer Science Engineering, Nalanda Institute of Engineering & Technology, Siddharth Nagar, Sattenapalli, Guntur., Affiliated to JNTUK, Kakinada, A.P., India.

Abstract - XML tree pattern query (TPQ) processing is a research stream within XML data management that focuses on efficient TPQ answering. With the increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a tree-structured data model. Queries on XML data are commonly expressed in the form of tree patterns (or twig patterns), which represent a very useful subset of XPath and XQuery. Efficiently finding all tree pattern matches in an XML database is a major concern of XML query processing. In the past few years, many algorithms have been proposed to match such tree patterns. This paper presents an overview of the state of the art in TPQ processing. This overview shall start by providing some background in holistic approaches to process TPQ and then introduce different algorithms for pattern matching.

Keywords - XML, Query processing, Pattern matching.

1. Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conform to the labelled-tree data model. The idea behind evaluating tree pattern queries, sometimes called the twig queries, is to find all existing ways of embedding the pattern in the data. Since XML data collections can be very large, efficient evaluation techniques for tree pattern matching are needed. From the formal point of view, XML data objects can be seen as ordered labelled trees. Following this model, previous approaches considered also the query trees ordered, so the problem can be characterized as the ordered tree pattern matching. Though there are certainly situations where the ordered tree pattern matching perfectly reflects the information needs of users, there are many other that would prefer to consider query trees as unordered. For example, when searching for a twig of the element person with the subelements first name and last name (possibly with specific values), ordered matching would not consider the case where the order of the first name and the last name is reversed. However, this could exactly be the person we are searching for. The way to solve this problem is to consider the query twig as an unordered tree where only the ancestor-descendant relationships are important – the preceding-following relationships are unimportant.

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a tree-structured data model. Since the data objects in a variety of languages (e.g. XPath [1], XQuery [2]) are typically trees, tree pattern matching is the central issue. For example, the following query:

"Q = //book[author="Chen"]//chapter/title"

can be represented as a twig (small tree) pattern. Intuitively, it returns the title of chapter for a book that has an author named by “Chen”.

In practice, XML data may be very large, complex and have deep nested elements. Thus, efficiently finding all twig patterns in an XML database is a major concern of XML query processing. In the past few years, many algorithms ([3],[4]) have been proposed to match such twig patterns. These approaches (i) first develop a labeling scheme to capture the structural information of XML documents, and then (ii) perform
tree pattern matching based on labels alone without traversing the original XML documents. For solving the first sub-problem of designing a proper labeling scheme, the previous methods use a *tree-traversal* order or textual positions of *start* and *end* tags (e.g. region encoding [5]) or path expressions (e.g. Dewey ID [6]) or prime numbers (e.g. [7]). By applying these labeling schemes, one can determine the relationship (e.g. ancestor-descendant) between two elements in XML documents from their labels alone.

2. XML Pattern Matching Algorithms

In the context of semi-structured and XML databases, tree-based query pattern is a very practical and important class of queries. Lore DBMS [8] and Timber [9] systems have considered various aspects of query processing on such data and queries. XML data and various issues in their storage as well as query processing using relational database systems have recently been considered in [6, 7]. The recent papers (e.g. [10,11]) are proposed to efficiently process an XML twig pattern. In paper [10], a new holistic algorithm, called OrderedTJ, is proposed to process order-based XML tree query. In paper [11], an algorithm called TwigStackListNot is proposed to handle queries with negation function. Chen et al [12] proposed different data streaming schemes to boost the holism of XML tree pattern processing. They showed that larger optimal class can be achieved by refined data streaming schemes. In addition, Twig2Stack [13] is proposed for answering generalized XML tree pattern queries. Note the difference between generalized XML tree pattern and extended XML tree pattern here. Generalized XML tree pattern is defined to include optional axis which models the expression in LET and RETURN clauses of XQuery statements. But extended XML tree pattern is defined to include some complicated conditions like negative function, wildcard and order restriction.

Besides the holistic algorithms, there are other approaches to match an XML tree pattern, such as ViST ([14]) and PRIX ([15]), which transform an XML tree pattern match to sequence match. Their algorithms mainly focus on ordered queries, and it is non-trivial to extend those methods to handle unordered queries and extended queries studied in this article. Note that the paper [16] made comprehensive experiments to compare different XML tree query processing algorithms (including sequence match and holistic match) and concluded that the family of holistic processing methods, which provides performance guarantees, is the most robust approach. From the aspect of theoretical research about the optimality of XML tree pattern matching, Choi et al. [6] developed theorems to prove that it is impossible to devise a holistic algorithm to guarantee the optimality for queries with any combination of P-C and A-D relationships. Shalem et al. [11] researched the space complexity of processing XML twig queries. Their paper showed that the upper bound of full-edge queries with parent-child and ancestor-descendant edges are $O(D)$, where $D$ is the document size. In other words, their results also theoretically prove that there exists no algorithm to optimally process an arbitrary query $Q/\ll,*$.

Most of these works build on some labeling scheme of XML elements to facilitate the verification of the structural relationship. The most commonly used labels are the containment and prefix labeling scheme. The containment labeling was introduced by Zhang et al. [17] to facilitate the containment queries. The verification of ancestor-descendant structural relationship is of the same complexity as that of parent-child relationship by using regional labeling. Dewey ID is the first example of using prefix labeling to represent XML data. It can be used to preserve the path information during query processing. Recent work of Lu et al. [14] utilize the extended Dewey encoding which encodes path information including not only the element IDs but also the element names.

3. Holistic Algorithms for XML Query Processing

In this section, we propose two algorithms to evaluate an XML tree query. The following illustrates data structures and notations for query class $Q/\ll,*$.

There is an input list $Tq$ associated with each query node $q$, in which all the elements have the same tag name $q$. Thus, we use $eq$ to refer to these elements. $\text{Cur}(Tq)$ denotes the current element pointed by the cursor of $Tq$. The cursor can be advanced to the next element in $Tq$ with the procedure $\text{advance}(Tq)$. There is a set $Sq$ associated with each branching query node $q$ (not each query node). Each element $eq$ in sets consists of a three-tuple $(\text{label}, \text{bitVector}, \text{outputList})$. $\text{label}$ is the extended Dewey label of $eq$, $\text{bitVector}$ is used to demonstrate whether the current element has the proper children or descendant elements in the document. Specifically, the length of $\text{bitVector}(eq)$ equals to the number of child nodes of $q$. Given a node $q_c2$
children(q), we use vector(eq)[qc] to denote the bit for qc. Specifically, vector(eq)[qc] = “1” if and only if there is an element eqc in the document such that the eq and eqc satisfy the query relationship between q and qc. Finally, outputList contains elements that potentially contribute to final query answers.

In our algorithm, we will frequently use the following two notations. 1) NAB(q) denotes the Nearest Ancestor Branching node of q in the query pattern Q. Formally, \(q^1 = NAB(q)\) if and only if \(q^1\) is a branching node and \(q^1\) is an ancestor of q and there is no other branching node \(q^1\) s.t. \(q^1\) is in the path from \(q^1\) to q. If there is no such ancestor of q, then NAB(q) denotes the top branching node in query. 2) NDB(q) denotes the nearest descendants branching (or leaf) nodes of q. Formally, \(q^1 = 2 \text{NDB}(q)\) if and only if \(q^1\) is a branching or leaf node and \(q^1\) is a descendant of q and there is no other branching or leaf node \(q^1\) s.t. \(q^1\) is in the path from \(q^1\) to q.

**TreeMatch Algorithm:**

Algorithm TreeMatch for class Q/, //, *.
1: locateMatchLabel(Q);
2: while (¬end(root)) do
3:   actf = getNext(topBranchingNode);
4:   if (actf is a return node)
5:     addToOutputList(NAB(actf), cur(Tactf));
6:   advance(Tactf); // read the next element in Tactf
7:   updateSet(fact); // update set encoding
8: end while
9: emptyAllSets(root);

Now we go through Algorithm . Line 1-3, for each stream, we use Procedure locateMatchedLabel to locate the first element whose path matches the individual root-leaf path pattern. In line 5, we identify the next stream \(T_{fact}\) to be processed by using getNext(topBranchingNode) algorithm, where topBranchingNode is defined as the branching node that is an ancestor of all other branching nodes(if any). In line 6, we output some path matching solutions in which each element that match any branching node b can be found in the corresponding set \(S_b\). We advance \(T_{act}\) in line 7 and locate the next matching element in line 8.

The following example illustrates how TreeMatch and TJfast algorithms work:

We use the query and document in Fig. 1 to illustrate both algorithms.
Fig. 1. Illustration to Algorithm TreeMatch for class $Q$, $\mathcal{L}$. (a) Query. (b) Data.

### Table 1
Set Encoding for the Example in Fig. 1

<table>
<thead>
<tr>
<th>Current elements</th>
<th>Set encoding $S_A$</th>
<th>Set encoding $S_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1, D_1, E_1$</td>
<td>$&lt;0, '10', 0&gt;$</td>
<td>$&lt;0, '12', '10', 0&gt;$, $&lt;0, '12', '1', '01', 0&gt;$</td>
</tr>
<tr>
<td>$B_1, D_1, E_2$</td>
<td>$&lt;0, '11', '00'&gt;$</td>
<td>$&lt;0, '12', '11', '00'&gt;$, $&lt;0, '12', '11', '01', 0&gt;$</td>
</tr>
<tr>
<td>$B_2, D_1, E_2$</td>
<td>$&lt;0, '11', '00'&gt;$, $&lt;0, '11', '10', 0&gt;$</td>
<td>$&lt;0, '11', '00'&gt;$, $&lt;0, '12', '01', 0&gt;$</td>
</tr>
</tbody>
</table>

Table 1 demonstrates the current access elements, the sets encoding and the corresponding output elements. There are two branching nodes in the query. First, $B_1$, $D_1$, and $E_1$ are scanned. $C_1$ and $C_2$ are added into the set $S_C$, but their bitVectors is “10” and “01”, which indicate that $C_1$ and $C_2$ have only one child, respectively. In this scenario, recall that TJFast may output path solutions $A_1/A_2/C_1/D_1$ and $A_1/A_2/C_1/C_2/E_1$, which might be useless to final results. Thus, our algorithm TreeMatch diminishes the unnecessary I/O cost. Next, $E_2$ is scanned and the bitVector($C_1$) becomes “11” as $C_1$ has two children now. Similarly, the bitVector($A_1$) is “11” too. In this moment, we guarantee that $A_1$ matches the whole pattern tree, as all bits in bitVector($A_1$) is 1 . Finally, when $B_2$ is scanned, $A_2$ is added to set $S_A$. Therefore, Treematch outputs two final results $B_1$ and $B_2$.

Through this example, we illustrate two differences between TJFast and TreeMatch. 1) TJFast outputs one useless intermediate path $A_1/A_2/C_1/C_2/E_1$, but TreeMatch uses the bitVector encoding to solve this problem. 2) TJFast outputs the path solution for all nodes in query, but TreeMatch only outputs nodes for return nodes (i.e., node $B$ in the query) to reduce I/O cost.

### 4. Conclusion

In this paper, we proposed the problem of XML tree pattern matching and surveyed some recent works and algorithms. Two algorithms TreeMatch and TJfast are introduced. TreeMatch has an overall good performance in terms of running time and the ability to process generalized tree patterns.

### References


AUTHORS PROFILE

Kamala Challa, Pursuing M.Tech(CS) from Nalanda Institute of Engineering & Technology, Siddharth Nagar, Sattenapalli, Guntur Affiliated to JNTUK, Kakinada, A.P., India. My research Interests are computer networks, Data Mining

E. Jhansi Rani, working as Asst. Professor, Department of Computer Science Engineering at Nalanda Institute of Engineering & Technology, Siddharth Nagar, Sattenapalli, Guntur Affiliated to JNTUK, Kakinada, A.P., India. My research Interests are Mobile Computing, Data Mining.