Twig Pattern Matching Algorithms for XML

D.BUJJI BABU
Associate Professor, Dept. of CSE, Prakasam Engg. College, Kandukur-523105, A.P., India

Dr. R.SIVA RAMA PRASAD
Research Director, Dept. of CSE, Acharya Nagarjuna University, Guntur-522510, A.P., India

M.SANTHOSH
Dept. of CSE, Prakasam Engg. College, Kandukur-523105, A.P., India

ABSTRACT—The emergence of XML promised significant advances in B2B integration. This is because users can store or transmit structure data using this highly flexible open standard. An effective well-formed XML document structure helps convert data into useful information that can be processed quickly and efficiently. From this point there is need for efficient processing of queries on XML data in XML databases like XML-enabled (MS-SQL Server, Oracle), Native XML (Mark Logic Server, EMC xDB). The research area in XML database is processing of XML tree pattern query (TPQ) called twig with efficient answers. Generally we have parsers that constructs the parse trees for some representation. Similarly, we have XML DOM parser it converts the XML document into XML tree. The XML query languages like XQL/XML Query Language), XML-QL(a query language for XML, Quilt, XPath (Extensible path language), XQuery (Extensible Query language) represent queries on XML data as twigs(small tree patterns). The major operation of XML query processing is to find all the occurrences of twig patterns efficiently on XML database. In the past few years, many algorithms have been proposed to match such tree patterns (twigs). 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 twig pattern matching.

Keywords: XML, XML databases, Pattern Matching Algorithms, XML Tree Pattern (twig), Query processing, XML Parsers

I. INTRODUCTION

There is an increasing need of XML data for data transporting in B2B\(^1\) application. With the rapidly increasing popularity of XML, more and more information is being stored, exchanged and presented in XML format. The ability to efficiently query XML data sources, therefore, becomes increasingly important. This paper gives the query processing of a core subset of query languages: XML twig queries. An XML twig query, represented as a small query tree, is essentially a complex selection on the structure of an XML document. Matching a twig query means finding all the instances of the query tree embedded in the XML data tree (this can be called as twig pattern matching). From this point there is need for efficient pattern matching algorithms on large volume of XML data for evaluating tree patterns (twigs). The outcome of XML DOM parser is XML tree. These XML trees (twigs) are available in two forms, they are ordered (ancestor and left-to-right ordering among siblings relationships significant) and unordered (only ancestor relationship significant) XML trees. In the past, the twigs are unordered. The present approaches considered XML trees as ordered labelled XML trees (twig) with the help of some labelling schemes [5].

The XML query languages (e.g. XPath [1], XQuery [2]) represent the queries as ordered labelled small trees (twigs).

For example, the following twig pattern written in XPath [1] format:

```
//section/title/paragraph/figure ................. (Q1)
```

This query selects figure elements which are descendants of some paragraph elements which in turn are children of section elements having at least one child element title. In the above query (Q1), "/" represents Parent-Child relationship and "/" represents Ancestor-Descendant relationship. Here the twig pattern is a node-labelled tree whose edges are parent-child and ancestor-descendant relationships.

The XML documents are usually modelled as trees and queries in XML query languages and are typically twig (or small tree) patterns with some nodes having value-based predicates. Therefore, finding all distinct matching’s of a twig pattern becomes a core operation in an XML query evaluation. The existing methods for tree pattern matching in XML are typically a decomposition-matching-merging process [3, 8, 9, 5, and 10].

The drawback of the decomposition-matching-merging methods is that the size of intermediate results may be much larger than the final answers. The main reason of having larger intermediate results and repeated matching of sub-patterns is due to the consideration of self-containment XML documents, i.e., an XML element that has the same tag with its sub-elements. However, in the real applications, we seldom find self-containment documents.

We present a fast tree matching algorithm called TreeMatch that can directly find all matching’s of a tree pattern in one step. The only requirement for the data source is that the matching elements of the non-leaf pattern nodes do not contain sub-elements with the same
tag. There are at least two advantages of TreeMatch. First, the TreeMatch algorithm does not need to decompose the query tree pattern, as it matches the pattern against the data source directly. Therefore, it does not produce any intermediate results and does not need the merging process. Second, the final results are compactly encoded in stacks and explicit representation of the results, either as a tree or a relation with each tuple representing one matching, can be generated efficiently.

II. TWIG PATTERN MATCHING IN XML

The existing methods for tree pattern matching (also called twig pattern matching) in XML are typically a decomposition-matching-merging process:

- Decompose the tree pattern (twig) into linear patterns which might be binary (parent-child or ancestor-descendant) relationships between pairs of nodes or root-to-leaf paths;
- Find all matching’s of each linear pattern; and
- Merge-join them to produce the result.

Most research in literature focuses on the second sub-problem: find all matching’s of a linear pattern. It can be classified according to the type of linear pattern they deal with, i.e., matching the binary structural relationships and matching path patterns. To match the binary structural relationships, Zhang et al. [5] proposed the MPMGIN (multi-predicate Merge Join) algorithm and Al-Khalifa et al. [9] gave the Stack-Tree algorithms. The algorithms accept two lists of sorted individual matching nodes and structurally join pairs of nodes from both lists to produce the matching’s of the binary relationships.

The difference between the MPMGIN and Stack-Tree is that the MPMGIN is a variation of the traditional merge-join algorithm, requiring multiple scans of the input lists. The Stack-Tree algorithm is more efficient as it uses stacks to maintain the ancestor or parent nodes and it needs only one scan of the input lists. Li et al. [10] and Chien et al. [8] use an index to facilitate the structural join process and do not require sorted input lists.

Recently, Bruno et al. [3] proposed algorithms called PathStack and TwigStack. The former is for matching path patterns and the latter is claimed to solve the problem of twig pattern matching. Both of them use a chain of stacks to encode the partial result. However, TwigStack does not match the twig pattern directly. It still belongs to the decomposition-matching-merging category.

All the algorithms discussed above use the format 

\[(DocId; Start; End; Level)\]

This format is used to represent the nodes in the database. DocId is the identity of the document the node belongs to, Start : End are the start and end positions of the corresponding element in the document and Level is the depth of the node in the XML tree hierarchy (e.g., for the root node, Level = 0).

The TwigStack algorithm [3] partially solved the problem of larger intermediate results with decomposition-matching-merging methods. When the patterns have only ancestor-descendant edges, the intermediate result of each path matching is guaranteed to be part of the final result. However, TwigStack’s requirement of matching all the root-to-leaf paths leads to repeated matching of the common nodes shared by multiple paths. If the query twig pattern has N leaf nodes, there will be O(N^2) different root-to-leaf paths. The matching of common nodes would be computed up to N times. For example, the tree pattern (twig) Q will be decomposed to two root-to-leaf paths, “journal/article/title” and “journal/article/author/Smith”. The Sub-Path “journal/article” is shared by two paths and will be matched repeatedly. The difficulty of directly matching tree patterns (twigs) comes from the self-containment property of the XML elements, that is, elements have the same tag with their sub-elements. However, self-containment is seldom found in real XML documents. Moreover, the self-containment property is easily identifiable. For an XML document with DTD (Document Type Definition), this property is indicated by the DTD. For an XML document without DTD, it is easily identified during the index construction process.

Other works for XML queries focus on the preprocessing of query patterns before the matching against the XML data source is executed. Amer-Yahia et al. [11] proposed a tree pattern (twig) minimization technique which aims at finding the smallest equivalent twig (tree pattern) of the original pattern by efficiently identifying and eliminating redundant nodes in the pattern. Flesca et al. [12] took one step forward by considering the minimization for general case tree pattern (twig) with wild card operators. Chen et al. [13] proposed the concept of GTP (generalized tree pattern) and presented an algorithm to translate a general XQuery query expression, which consists of more than one tree pattern (twig) and possibly involves quantifiers, aggregation and nesting, into a GTP. Evaluating the query expression reduces to finding distinct matches of the GTP.

III. HOLISTIC ALGORITHMS FOR XML TWIG PATTERN MATCHING

Here we propose an algorithm to evaluate a large XML twig query called extended XML tree pattern (twig).

TreeMatch Algorithm:-

The previous algorithms TwigStack [3], TwigStackList [7], OrderedTJ [4], and TIFast [6] requires bounded main memory for small class of queries with Parent-Child, Ancestor-Descendant relationships. The XML query languages like XPath [1], XQuery [2] defines axes (relationships) and functions such as negation, wildcard, order-based functions. This TreeMatch algorithm defines an extended XML tree pattern (twig) means P.C, A.D, negation, wildcard and/or order restriction. The extended XML tree patterns as shown in below figures:
Here we have three categories of extended XML tree patterns (twigs) in Fig. 1.

- \( Q^{P-C} \) means queries with P-C-A-D relationships and wildcards. Here \( \sim / \) denotes Parent-Child (P-C) relationship, \( \sim \) denotes Ancestor-Descendant (A-D) relationship and a wildcard \( \sim * \) means it can match any single node in an XML database.
- \( Q^{P-C,A-D} \) means queries with P-C, A-D relationships, wildcards and order restriction. Here \( \sim \) shows that the nodes are ordered.
- \( Q^{P-C,A-D}* \) means queries with P-C, A-D relationships, wildcards, order restriction and negation function. Here \( \sim \) represents negation function.

The TreeMatch algorithm is proposed to achieve larger optimal query classes. It uses a concise encoding technique to match the results and also reduces the useless intermediate results. Most XML query processing algorithms on XML documents rely on certain labeling schemes, such as region encoding scheme [5], prefix scheme [10], ORDPATH [14], extended Dewey scheme [6]. In this paper, we use the extended Dewey labeling scheme, proposed in paper [6], to assign each node in XML documents a sequence of integers to capture the structure information of documents.

Extended Dewey labeling scheme is a variant scheme of the prefix labeling scheme [10]. In the prefix labeling scheme, the root is labeled by an empty string and for a non-root element \( u \), label (\( u \)) = label (\( v \).\( u \)), where \( u \) is the \( n \)th child of \( v \). In Extended Dewey labeling scheme [6], each label provides complete information about ancestors’ names and labels. For example, given an element \( e \) with label \( 1.2.3 \), prefix labeling schemes can tell us parent(e) = "1.2" and grandparent(e) = "1", but extended Dewey labeling scheme can also tell us the tag name of elements, say, tag(e) = "A", tag(parent(e)) = "B" and tag(grandparent(e)) = "C". In order to achieve this goal, paper [9] uses module function to encode the element tag information to prefix labels, and use finite state transducer (FST) to decode the type’s information for a single extended Dewey label. The complete path information in extended Dewey labels enables holistic algorithms to scan only leaf query nodes to answer an XML query.

Data structures and notations:

For each query node \( q \), the associated input list is \( T_q \) and all elements have same tag name that is \( e_q \) cur (\( T_q \)) function denotes the current element pointed by cursor of \( T_q \). The cursor value is incremented and it points to the next element in \( T_q \) with the procedure advance (\( T_q \)).

For each branching query node (a node has more than one child), the associated set is \( S_q \). Each element \( e_q \) in sets consists of a 3-tuple (label; bitVector; outputList). label is the extended Dewey label of \( e_q \). bitVector is used to demonstrate whether the current element has the proper children or descendant elements in the document. Specifically, the length of bitVector(\( e_q \)) equals to the number of child nodes of \( q \). Given a node \( q \), \( e \) children (\( q \)), we use bitVector (\( e_q \)) to denote the bit for \( q \). Specifically, bitVector (\( e_q \)) \( q_i \) = "1" if and only if there is an element \( e_q \) in the document such that the \( e_q \) and \( e_i \) satisfy the query relationship between \( q \) and \( q_i \). Finally, outputList contains elements that potentially contribute to final query answers.

Consider the data and query in Fig. 2. Here B is a single return node (denoted by underline in figure). Firstly, B1 and C1 are read. Since A1 ("0") has only one child B1 and no child element to match C, bitVector (A1) = "10". In contrast, bitVector (A2) = "11", since A2 ("0.1") has two children B2 and C1, which satisfy the P-C relationships in the query. Since all bits in bitVector (A) are "1", B2 ("0.1.0") is guaranteed to be a query answer.

<table>
<thead>
<tr>
<th>Current elements</th>
<th>Set encoding of ( S_A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_1, C_1 )</td>
<td>( &lt;0.1^\sim,0.0^\sim&gt; )</td>
</tr>
<tr>
<td>( B_2, C_1 )</td>
<td>( &lt;0.1^\sim,1.1^\sim,0.1^\sim&gt;, &lt;0.10^\sim,0.0^\sim&gt; )</td>
</tr>
</tbody>
</table>

Table 1

Set encoding for Fig. 2

Fig. 2 illustrates the set encoding \( S_A \) to query node A for an example document. There are two tuples in set \( S_A \). Since A1 ("0") has only one child B1 and no child element to match C, bitVector (A1) = "10". In contrast, bitVector (A2) = "11", since A2 ("0.1") has two children B2 and C1, which satisfy the P-C relationships in the query. Since all bits in bitVector (A) are "1", B2 ("0.1.0") is guaranteed to be a query answer.
avoiding the output of B1, as bitVector (A1) is “10”. But TwigStack and TJFast would output two “useless” elements A1 and B1 in that case, and therefore, entail more I/O cost.

**Algorithm TreeMatch for Query class Q^{"/nte",}"**

1: locateMatchLabel (Q);  
2: while (notEnd (root)) do 
3: \( f_{act} = \text{getNext (topBranchingNode)}); \  
4: if (\( f_{act} \) is a return node) then 
5: addToOutputList (NearestAncestorBranching (\( f_{act} \)), cur (\( T_{activ} \))); 
6: advance (\( T_{act} \)); // read the next element in \( T_{act} \) 
7: updateSet (\( f_{act} \)); // update set encoding 
8: locateMatchLabel (Q); // locate next element with matching path 
9: emptyAllSets (root); 

**Procedures and Functions used in TreeMatch for Query class Q^{"/nte",}"**

**Procedure locateMatchLabel (Q)**
1: for each leaf \( q \in Q \), locate the extended Dewey label \( e_q \) in list \( \mathcal{T}_q \) such that \( e_q \) matches the individual root-leaf path 

**Procedure addToOutputList (\( q, e_q \))**
1: for each \( e_q \in S_q \) do 
2: if (satisfyTreePattern (\( e_q, e_q \))) then 
3: outputList (\( e_q \)); 

**Function satisfyTreePattern (\( e_q, e_q \))**
1: if (bitVector (\( e_q, q \)) = “1”) then return true; 
2: else return false; 

**Procedure updateSet (\( q, e \))**
1: cleanSet (\( q, e \)); 
2: add to set \( S_q \); // set the proper bitVector (\( e \)) 
3: if (isNotRoot (\( q \)) and (bitVector (\( e \)) = “1...1”)) then updateAncestorSet (\( q \)); 

**Procedure cleanSet (\( q, e \))**
1: for each \( e_q \in S_q \) do 
2: if (satisfyTreePattern (\( e_q, e \))) then 
3: if (\( q \) is a return node) then 
4: addToOutputList (NearestAncestorBranching (\( q \), e), \( e \)); 
5: if (isTopBranching (\( q \))) then 
6: if (there is only one element in \( S_q \)) then 
7: output all elements in outputList (\( e_q \)); 
8: else merge all elements in outputList (\( e_q \)) to outputList (\( e_q \)), where \( e_q = \text{NearestAncestorBranching (} e_q \)\); 
9: delete \( e_q \) from set \( S_q \); 

**Procedure updateAncestorSet (\( q \))**
1: /* assume that \( q^\prime = \text{NearestAncestorBranching (} q \)*/ 
2: for each \( e \in S_q \) do 
3: if (bitVector (\( e, q \)) = 0) then 
4: bitVector (\( e, q \)) = 1; 
5: if (isNotRoot (\( q \)) and (bitVector (\( e \)) = “1...1”)) then 
6: updateAncestorSet (\( q \)); 

**Procedure emptyAllSets (\( q \))**
1: if (\( q \) is non-leaf node) then 
2: for each child \( c \) of \( q \) do emptyAllSets (\( c \)); 
3: for each element \( e \in S_q \) do cleanSet (\( q, e \)); 

**Algorithm for getNext (\( n \))**
1: if (\( isLeaf (n) \)) then 
2: return \( n \) 
3: else 
4: for each \( n_i \in \text{NearestDescendantBranching (} n \)\) do 
5: \( f_i = \text{getNext (} n_i \)\); 
6: if (\( isBranching (n_i) \) and notEmpty (\( S_{min} \)) ) then 
7: return \( f_i \) 
8: else \( e_i = \max \{ \text{p | p \in MB (} n_i, n \)\} \) 
9: end for 
10: \( max = \max \{ e_i \} \) 
11: for each \( n_i \in \text{NearestDescendantBranching (} n \)\) do 
12: if (\( For 
\text{All (} e \) \in MB (} n_i, n : e \) is not belongs to Ancestors (\( e_{max} \)) \) then 
13: return \( f_i \); 
14: end if 
15: end for 
16: \( min = \min \{ e_i | f_i \text{ is not a return node} \} \) 
17: for each \( e \in MB (} n_{max}, n \)\) do 
18: if (\( e \in \text{ancestors (} e_{max} \)) \) then updateSet (\( S_{max}, e \) 
19: end for 
20: return \( f_{max} \) 
21: end if 

**Function MB (\( n, b \))**
1: if (\( isBranching (n) \)) then 
2: let \( e \) be the maximal element in \( S_n \) 
3: else 
4: let \( e = \text{cur (} T_n \)\) 
5: end if 
6: Return a set of element \( a \) that is an ancestor of \( e \) such that a can match node \( b \) in the path solution of \( e \) to path pattern \( p_n \) 

Now we go through Algorithm (TreeMatch for class Q^{"/nte","}). Line 1 locates the first elements whose paths match the individual root-leaf path pattern. In each iteration, a leaf node \( f_{act} \) is selected by getNext function (line 3). The purpose of lines 4 and 5 is to insert the potential matching elements to outputList. Line 6 advances the list \( T_{activ} \) and line 7 updates the set encoding. Line 8 locates the next matching element to the individual path. Finally, when all data have been processed, we need to empty all sets in Procedure emptyAllSets (line 9) to guarantee the completeness of output solutions.

In Procedure addToOutputList (\( q, e_q \)), we add the potential query answer \( e_q \) to the set of \( S_q \), where \( q \) is the nearest ancestor branching node of \( q_i \) (i.e. NAB (\( q_i = q \)). Procedure updateSet accomplishes three tasks. First,
clean the sets according to the current scanned elements. Second, add e into set and calculate the proper bitVector. Finally, we need recursively update the ancestor set of e. Because of the insertion of e, the bit Vector values of ancestors of q need update.

Algorithm getNext (see the above) is the core function called in TreeMatch, in which we accomplish two tasks. For the first task to identify the next processed node, Algorithm getNext (n) returns a query leaf node f according to the following recursive criteria

(i) if n is a leaf node, f = n (line 2); else
(ii) n is a branching node (a node having more than one child), then suppose element e, which matches node n in the corresponding path solution (if more than one element that matches n, e, is the deepest one by level) (line 7,8), we return fmin such that the current element e, in Tmin has the minimal label in all e, by lexicographical order (line 13,20).

For the second task of getNext, before an element e, is inserted to the set Sq, we ensure that e, is an ancestor (or parent) of each other element e, to match node b in the corresponding path solutions (line 13). If there is more than one element to match the branching node b, e, is defined as their deepest (i.e. maximal) element (line 8).

Algorithm TreeMatch for class Q^<"./",">:

Procedure updateSet (q, e)
1: cleanSet (q, e);
2: add e to set S_q; /set the proper bitVector, minChild and maxChild
3: if (isNotRoot (q) and (bitVector (e) = 1...1)) then updateAncestorSet (q);

Function satisfyTreePattern (q, e, c, q, e, c, q, e, c, q, e, c):
1: if (e < minChild (q, q, c, q, e, c, q, e, c, q, e) ) then return false;
2: else if (e > maxChild (q, q, c, q, e, c, q, e, c, q, e)) then return false;
3: else if ((bitVector (e, q, c, q, e, c, q, e, c, q, e) ) = ‘1’ ) and (negBitVector (e, q, c, q, e, c, q, e, c, q, e)) return false;

The above algorithm TreeMatch for Q^<"./","> supports the negative edges. It is similar to Q^<"./",">,<",") but the procedure updateSet and function satisfyTreePattern are changed. We add negBitVector to record the matching information about negative child edge. Given a node q, e, negativeChildren (q, e, c, q, e, c, q, e, c) and only if there is no element e, in the document such that the e, and e, satisfy the query relationship in between q and q, e. In this case, we also have the negateBitVectors are “0”. In line 2 of Procedure updateSet, we need to set the proper negBitVector according to the current elements. In Function satisfyTreePattern, e, is a valid element only if the negateBitVector is ‘0’.

IV. CONCLUSION

In this paper, we proposed the problem of XML twig pattern matching and surveyed some recent works and algorithms. The TreeMatch algorithms with different query classes are introduced. TreeMatch has an overall good performance in terms of running time and the ability to process extended XML tree patterns (twigs). The previous twig pattern matching algorithms (TwigStack, TwigStackList, OrderedTI, and TIFast) requires bounded main memory for small queries. But, the TreeMatch works on one-phase query evaluation and it requires bounded main memory for larger queries. From this point we can say that TreeMatch twig pattern matching algorithm can answer complicated queries and has good performance.

REFERENCES


AUTHORS PROFILE

1. D. Bujji Babu currently working as an Associate professor in the department of Computer Science and Engineering, at Prakasam Engineering College, Kandukur, A.P. India. He is having 4 years of research and 10 years of teaching experience. He is a research scholar in the department of CSE at Acharya Nagarjuna University, India.
E-mail: bujjibict@yahoo.com

2. Dr. Siva Rama Prasad currently working as a head of the International Business Administration department at Acharya Nagarjuna University, India. He has published several research papers in various peer reviewed international journals. Authored seven books and also he is working as a research director in the department of CSE.
E-mail: raminenisivaram@yahoo.co.in

E-mail: santhosh.maddisetti@gmail.com