XML Twig Pattern Matching Algorithms and Query Processing

D.BUJJI BABU [1]
Associate Professor,Dept. of CSE,
Prakasam Engg. College,
Kandukur-523105,
A.P., India

Dr. R.SIVA RAMA PRASAD [2]
Research Director, Dept. of CSE,
Acharya Nagarjuna University,
Guntur-522510,
A.P., India

M.SANTHOSH [3]
Dept. of CSE,
Prakasam Engg. College,
Kandukur-523105,
A.P., India

ABSTRACT

XML has emerged as the wire-language of the internet.XML can be used to structure the data and also provide meaning for data. An effective document structure helps convert data into useful information that can be processed quickly and efficiently. The XML data is exchanged and generated in B2B[1] applications. According to this point there is need for efficient processing of queries on XML data. The research stream in XML database is processing of XML tree pattern query (XTPQ) with efficient answer (called pattern matching).The XML document can be converted into tree model by using DOM (Parser).The XML query languages like XPath (Extensible path language), XQuery (Extensible Query language) represent queries on XML data as tree patterns (twigs).The major operation of XML query processing is to find all the occurrences of twig [1] patterns efficiently on XML database. 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 XTPQ processing. This overview shall start by providing some background in holistic approaches to process XTPQ and then introduce different algorithms for twig pattern matching.

Keywords: - XML, Pattern Matching Algorithms, XML Tree Pattern, Query processing, XML Parsers.

1. INTRODUCTION

There is an increasing need of XML data for data transporting application in businesses. The evaluation of XML tree pattern queries (XTPQ) produces output as all matched patterns is called twig patterns (this can be called as pattern matching). The emergence of this point is need for efficient pattern matching algorithms on large volume of XML data for evaluating tree patterns (twigs). The DOM parser represents the XML document as XML tree. The XML trees again two types ordered (ancestor and left-to-right ordering among siblings relationships significant) and unordered (only ancestor relationship significant) XML trees. Some algorithms produces output as unordered XML trees and some produces output as ordered labeled XML trees(twigs).The previous approaches considered XML tree as ordered labeled XML tree(twig). For example, when searching for a twig of the element student with the sub elements 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 student 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*, *preceding-sibling and following-sibling* relationships (axes) 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 (twig) is* the central issue.

For example, the following query:

Query=/book [title='XML']//author [name='Jane']

can be represented as a twig (small tree) pattern. It matches author elements that has sub element name as content the string value “Jane”, and are descendants of book elements that have a child title element whose content is the string value “XML”. In the above query "/" represents the relationship between parent and child

[1] B2B means business to business

"//" represents the relationship between ancestor and descendant.

In practice, XML data may be very large, complex and have nested elements. Thus, efficiently finding all twig patterns in an XML database is a major operation of XML query processing. In the past few years, many algorithms ([3], [4]) have been proposed to match such twig patterns. These approaches

- First develop a labeling scheme to capture the structural information of XML documents, and then
- 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, parent-child) between two elements in XML documents from their labels alone.

2. XML Twig Pattern Matching

The XML databases like Lore [8] and Timber [9] represents the XML query as small tree called twig. XML data and its related issues of 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 (XTPQ). In paper [10], a new holistic algorithm, called ordered, 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.

We have other approaches to match an XML tree pattern are ViST[14] and PRIX[15] ,which converts an XML tree pattern match into sequence match. These two algorithms mainly focus on ordered queries and it is non-trivial to extend those methods to handle unordered queries. The paper [16] gives different XML tree query processing algorithms (including holistic match and sequence match) and concluded that the holistic tree pattern method is robust in nature also guarantees performance. From the theoretical research about the optimality of XML tree pattern matching, Choi et al. [20] developed theorems to prove that it is impossible to devise a holistic algorithm to guarantee the optimality for queries with any combination of Parent-Child and Ancestor-Descendant relationships. Shalem et al. [21] 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 $Q^{/,//,*}$.

The structural relationships are verified with the help of labeling scheme of XML elements. The most commonly used labeling schemes are containment and prefix. The containment labeling scheme was introduced by Zhang et al. [17] for containment queries. The axes like Parent-Child and Ancestor-Descendant relationships have the same complexity according to regional labeling. The example to represent prefix labeling scheme on XML data is Dewey ID. It can be used to preserve the path information during query processing. Recent work of Lu at el. [14] utilizes the *extended Dewey* encoding [18] which encodes path information including not only the element IDs but also the element names.

3. Holistic Algorithms for XML Query Processing

Here we propose two types of algorithms to process an XML twig query. They are

- Two-Phase holistic twig evaluation algorithms
- One-Phase holistic twig evaluation algorithms

a) TwigStack Algorithm:

Based on the containment labeling scheme [17], Bruno et al. [5] proposed a novel holistic XML twig pattern matching method **TwigStack** which avoids storing intermediate results unless they contribute to the final results. The method, unlike the decomposition based method, avoids computing large redundant intermediate results. But the main limitation of **TwigStack** is that it may produce a large set of “useless” intermediate results when queries contain any parent-child relationships. **TwigStack** has been proved to be I/O optimal in terms of output sizes for queries with only A-D edges, their algorithms still cannot control the size of intermediate results for queries with parent-child (P-C) edges. **TwigStack** operates in two steps:

- A list of intermediate path solutions is output as intermediate results; and
- The intermediate path solutions in the first step are merge-joined to produce the final solutions.

Algorithm for TwigStack (q):

```
// Phase 1
1: while notEnd (q)
2: $q_{act}$ = getNext(q)
3: if (isNotRoot($q_{act}$)) then
4: cleanStack(parent($q_{act}$), nextL($q_{act}$))
5: end if
6: if (isRoot($q_{act}$) or isNotEmpty($S_{parent(qact)}$)) then
7: cleanStack($q_{act}$, next($q_{act}$))
8:moveStreamToStack($T_{qact}$,$S_{qact}$ ,pointertotop($S_{parent (qact)}$))
9: if (isLeaf($q_{act}$)) then
10: showSolutionsWithBlocking($S_{qact}$, 1)
11: pop ($S_{qact}$)
12: end if
13: else
14: advance ($T_{qact}$)
15: end if
16: end while
// Phase 2
17: mergeAllPathSolutions()
```

Algorithm **TwigStack** operates in two phases. In the first phase (lines 1-16), some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line 17), these solutions are merge-joined to compute the answers to the query twig pattern.

b) TwigStackList Algorithm:

Unlike the previous Algorithm **TwigStack** [5], our approach takes into account the level information of elements and consequently output much less intermediate paths for query twig patterns with parent-child edges. We have analytically shown that when all edges below branching nodes (nodes that has more than one child) in the query pattern are ancestor-descendant relationships, the I/O cost of **TwigStackList** is only equal to the sum of sizes of the input and the final output. In other words, **TwigStackList [19]** identifies a larger query class to guarantee the I/O optimality than **TwigStack**, which only guarantee the optimality for queries with entirely A-D relationships. Experimental results showed that our method achieves the similar performance with **TwigStack** for queries with only ancestor-descendant relationships, but is much more efficient than **TwigStack** for queries with parent-child relationships, especially for deep data sets with complicated recursive structure.

Algorithm for TwigStackList:

```
1: while notEnd() do
2: $q_{act}$= getNext(root)
3: if (isNotRoot($q_{act}$)) then
4: cleanParentStack($q_{act}$,getStart($q_{act}$))
5: end if
6: if (isRoot($q_{act}$ ) or isNotEmpty($S_{parent(qact)}$)) then
7: clearSelfStack($q_{act}$,getEnd($q_{act}$))
8:moveToStack
($q_{act}$,S$q_{act}$,pointertotop(Sparent($q_{act}$)))
9: if (isLeaf($q_{act}$)) then
10: showSolutionsWithBlocking(S$q_{act}$,1)
11: pop (S $q_{act}$ )
12: end if
13: else
14: proceed ($q_{act}$)
15: end if
16: end while
17: mergeAllPathSolutions()
```

First of all, line 2 calls getNext algorithm to identify the node q_{act} to be processed. Line 4 and 7 remove partial answers from the stacks of parent(q_{act}) and q_{act} that cannot be extended to total answer. If q_{act} is not a leaf node, we push element Cq into Sq (line 8); otherwise (line 10), all path solution involving C_q can be output. Note that path solutions should be output in root-leaf order so that they can be easily merged together to form final twig matches (line 17).

c) OrderedTJ Algorithm:

It's an extension of **TwigStackList**. Here

(a)We introduce a new algorithm, called **OrderedTJ[10]**, for holistic ordered twig pattern processing. In **OrderedTJ**, an element contributes to final results only if the order of its children accords with the order of corresponding query nodes.
(b) If we call edges between branching nodes and their children as branching edges and denote the branching edge connecting to the n'th child as the n'th branching edge, we analytically demonstrate that when the ordered query contains only A-D relationship from the second branching edge, **OrderedTJ [10]** is I/O optimal among all sequential algorithms that read the entire input. In other words, the optimality of **OrderedTJ** allows the existence of P-C edges in non-branching edges(a node that has only one child) and the first branching edge(a node that has more than one child).

Algorithm for OrderedTJ:

```
1: while notEnd() do
2: $q_{act}$= getNext(root)
3:if (isRoot($q_{act}$) or isNotEmpty(Sparent($q_{act}$))) then
4: cleanStack($q_{act}$,getEnd($q_{act}$))
5: end if
6: moveStreamToStack($q_{act}$,S $q_{act}$);
```

```
7: if (isLeaf($q_{act}$)) then
8: showPathSolutions(S$q$,getElement($q_{act}$))
9: else
10: proceed ($q_{act}$)
11: end if
12: end while
13: mergeAllPathSolutions();
```

Algorithm **OrderedTJ**, which computes answers to an ordered query twig, operates in two phases. In the first phase (line 1-12), the individual query root-leaf paths are output. In the second phase (line 13), these solutions are merged-joined to compute the answers to the whole query.

d) TJFast Algorithm (a Fast Twig Join algorithm):

The above three algorithms are based on containment labeling scheme[17]. A new algorithm, namely **TJFast**, which exploits the nice property of the extended Dewey labeling scheme [18] and efficiently evaluates XML twig queries. The containment labeling scheme is difficult to answer queries with wildcards in branching nodes(a node that has more than one child). For example, consider an XPath: "//a/*/ [b]/c". Where "*" denotes a wildcard symbol which can match any single element. The containment labels of a, b and c do not provide enough information to determine whether they match the query or not.

Algorithm for TJFast:

```
1: for each f∈ leafNodes(root)
2: locateMatchedLabel(f)
3: end for
4: while (notEnd(root)) do
5: $f_{act}$= getNext(topBranchingNode)
6: outputSolutions($f_{act}$)
7: advance($T_{fact}$)
8: locateMatchedLabel($f_{act}$)
9: end while
10: mergeAllPathSolutions()
```

Algorithm **TJFast,** which computes answers to a query twig pattern Q, is presented in Algorithm 7. **TJFast** operates in two phases. In the first phase (line 1-9), some solutions to individual root-leaf path patterns are computed. In the second phase (line10), these solutions are merge-joined to compute the answers to the whole query.

e) TreeMatch Algorithm:

This 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.

Algorithm TreeMatch for class Q/, //, *.

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

Now we go through Algorithm. 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_{fact} 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.

The below Fig. (a) shows the query and document illustrate the TreeMatch algorithm for class $Q^{/,//,*}$

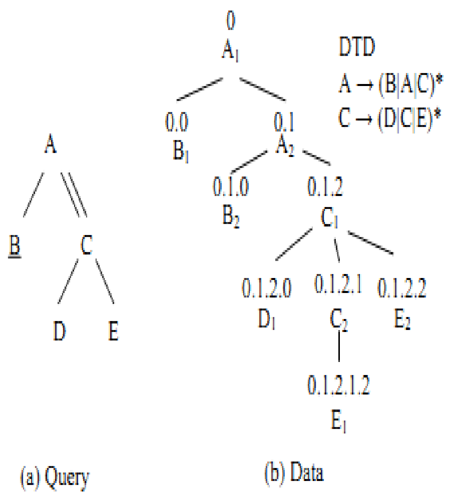


Fig (a) Illustration to Algorithm TreeMatch for class $Q^{/,//,*}$

Current elements	Set encoding S_A	Set encoding S_C
B_1,D_1,E_1	<0,"10",∅>	<0.1.2,"10",∅>, <0.1.2.1,"01",∅>
B_1,D_1,E_2	<0,"11","0.0">	<0.1.2,"11",∅>, <0.1.2.1,"01",∅>
B_2,D_1,E_2	<0,"11","0.0">, <0.1,"11","0.1.0">	<0.1,"11",∅>, <0.1.2.1,"01",∅>

TABLE 1
Set encoding for the example in above Fig (a)

The above TABLE1 demonstrates the current access elements, the sets encoding and the corresponding output elements. There are two branching nodes in the query. First, B1, D1, and E1 are scanned. C1 and C2 are added into the set S_C, but their bitVectors is "10" and "01", which indicate that C1 and C2 have only one child, respectively. In this scenario, recall that **TJFast** may output path solutions A1/A2/C1/D1 and A1/A2/C1/C2/E1, which might be useless to produce final results. Thus, our algorithm **TreeMatch** diminishes the unnecessary I/O cost. Next, E2 is scanned and the bitVector (C1) becomes "11" as C1 has two children now. Similarly, the bitVector (A1) is "11" too. In this moment, we guarantee that A1 matches the whole pattern tree, as all bits in bitVector (A1) is 1. Finally, when B2 is scanned, A2 is added to set S_A. Therefore, Treematch outputs two final results B1 and B2.

Through this example, we illustrate two differences between **TJFast** and **TreeMatch**.

1) **TJFast** outputs one useless intermediate path A1/A2/C1/C2/E1, 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. EXPERIMENTAL STUDY

The experimental study verifies the effectiveness, in terms of accuracy and optimality, of various holistic twig pattern matching algorithms are shown in Fig (b) and Fig (c).

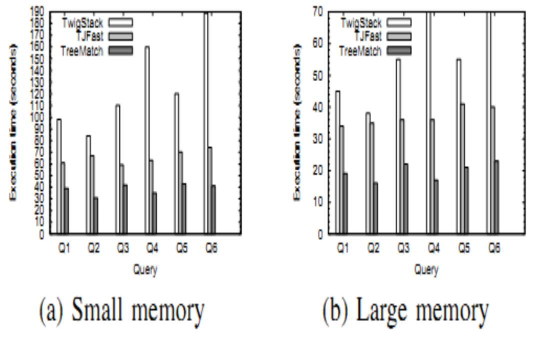
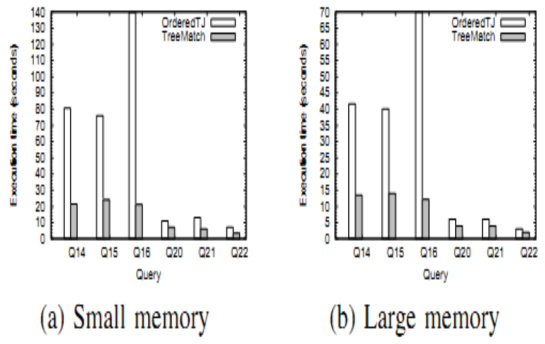


Fig (b) Execution time of $Q^{/,//,*}$ on random data



Fig(c) Execution time of $Q^{/,//,*,<}$ on random data

5. CONCLUSION

In this paper, we proposed the problem of XML twig pattern matching and surveyed some recent works and algorithms. Five algorithms TwigStack [5], TwigStackList [19], OrderedTJ [10], TJFast [18] and TreeMatch are introduced. TreeMatch has an overall good performance in terms of running time and the ability to process generalized tree patterns. From the above algorithms we can observe one point that is first four twig pattern matching algorithms (TwigStack, TwigStackList, OrderedTJ, and TJFast) works on two-phase query evaluation and TreeMatch works on one-phase query evaluation. From this point we can say that TreeMatch twig pattern matching algorithm can answer complicated queries and has good performance.

References

[1] A. Berglund, S. Boag, and D. Chamberlin. XML path language (XPath) 2.0. W3C Recommendation 23 January 2007 http://www.w3.org/TR/xpath20/.
[2] S. Boag, D. Chamberlin, and M. F. Fernandez. Xquery 1.0: An XML query language. W3C Working Draft 22 August 2003.
[3] H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with OR-predicates. In *Proc. of SIGMOD Conference*, pages 274-285, 2004.
[4] H. Jiang et al. Holistic twig joins on indexed XML documents. In *Proc. of VLDB*, pages 273-284, 2003.
[5] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal XML pattern matching. In *Proc. of SIGMOD Conference*, pages 310-321, 2002.
[6] I. Tatarinov, S. Viglas, K. S. Beyer, Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of SIGMOD*, pages 204-215, 2002.
[7] X. Wu, M. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proc. of ICDE*,pages 66-78, 2004.
[8] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, pages 436-445, 1997.
[9] H. V. Jagadish and S. AL-Khalifa. Timber: A native XML database. Technical report, University of Michigan, 2002.
[10] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Effcient processing of ordered XML twig pattern matching. In *DEXA*,pages 300-309, 2005.
[11] T. Yu, T. W. Ling, and J. Lu. Twigstacklistnot: A holistic twig join algorithm for twig query with not-predicates on xml data. In *DASFAA*, pages 249-263, 2006.
[12] T. Chen, J. Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing Techniques. In *SIGMOD*, pages 455-466, 2005.
[13] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D.Agrawal, and K. S. Candan.Twig2stack: Bottom-up processing of generalized-tree-pattern queries over xml document. In *Proc. of VLDB Conference*, pages 19-30, 2006.
[14] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110-121, 2003.
[15] P. Rao and B. Moon. PRIX: Indexing and querying XML using prufer sequences. In *ICDE*, pages 288-300, 2004.
[16] M. Moro, Z. Vagena, and V. J. Tsotras. Tree-pattern queries on a lightweight XML processor. In *VLDB*, pages 205-216, 2005.
[17] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proc. of SIGMOD Conference*, pages 425-436, 2001.
[18] J. Lu, T. W. Ling, C. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching.In VLDB, pages 193–204, 2005.
[19] J. Lu, T. Chen, and T. W. Ling. Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In CIKM, pages 533–542, 2004.
[20]B. Choi, M. Mahoui, and D. Wood. On the optimality of the holistic twig join algorithms. In Proceeding of DEXA, pages 28–37, 2003.
[21]M. Shalem and Z. Bar-Yossef. The space complexity of processing xml twig queries over indexed documents. In ICDE, 2008.

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: bujji_bict@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

3. Kum.M.Santhosh M.Tech (CSE) from Prakasam Engineering College, Kandukur, Prakasam (Dt.), Affiliated by JNTUK, Kakinada, A.P., India.
E-mail: santhosh.maddisetty@gmail.com