International Journal of Engineering Research and Development
ISSN: 2278-067X, Volume 2, Issue 1 (July 2012), PP. 39-45
www.ijerd.com

XML Tree Pattern Matching Algorithms

Lakshmi Tulasi.Ambati [1], Y.SSR.Murthy[2], D.N.S.B.Kavitha[3]
[1,2,3]*Computer Science Engineering, Shri Vishnu Engineering College For Women,Vishnupur, Bhimavaram, West Godavari District, AndhraPradesh, India.*

Abstract*—In the present day digital world, it is imperative that all organizations and enterprises facilitate efficient processing of queries on XML data. XML queries typically specify patterns of selection predicates on multiple elements that have specified tree structured relationships. The primitive tree-structured relationships are parent-child and ancestor-descendant. Finding all occurrences of these relationships in an XML database is a core operation for XML query processing. In this paper the pattern matching algorithms TwigStack and TwigStackList are discussed. The behavior of TwigStack is analyzed, and a comparison of these two algorithms is attempted. The TwigStack algorithm the initial holistic algorithm, has features of performing simultaneous scan over streams of XML nodes to match their structural relationships holistically, reducing a number of unnecessary intermediate results, and skipping XML nodes that will not contribute to final answers. The family of holistic pattern matching algorithms has appeared as the major important algorithms for processing XML query patterns due to its efficiency and performance advantage. The experimental results show that the query performance is significantly improved especially for queries having relatively more complex structures and/or higher selectivities.

***Keywords*—Xml,TwigStack,TwigStackList;**

I. INTRODUCTION

XML employs a tree-structured model for representing data. In Xml, XPath and XQuery [1] are used for addressing the parts of an xml document and for specifying patterns of selection predicates on multiple elements that have specified tree-structured relationships. For example, the XQuery path expression

Book [author=suciu] // [title=XML]

An XML tree pattern query, represented as a labeled tree, is essentially a complex selection predicate on both structure and content of an XML. Tree pattern matching has been identified as a core operation in querying XML data. The data in Xml is arranged by using the grammar DTD (Document Type Definition) fig 2. In web mining, the data is retrieved from web through XML tree. The XML tree gives all relevant information to the users of the web. Xml allows for structuring of data on the web. The structure of XML data is represented in fig 1. An XML document is made of elements limited by tags and is hierarchically structured.

II. BACKGROUND

The extensible markup language XML has recently emerged as a new standard for information representation and exchange on the Internet. XML allows users to make up any new tags for descriptive markup of their own applications. Since XML data is self-describing, XML is considered one of the most promising means to define semi-structured data, which is expected to be ubiquitous in large volumes from diverse data sources and applications on the web. In Xml Tree there is a Parent-Child (P-C) and Ancestor and Descendant (A-D) relationships which are represented as / and // in fig 3. A tree which is maintained by both Parent-Child (P-C) and Ancestor and Descendant (A-D) relationships is presented in fig.3. There are some pattern matching algorithms [4][5], which are not much efficient than TwigStack [2]discussed in III. Twigstack implementation is discussed in section IV. To overcome some limitations of TwigStack a TwigStack List is discussed in section V. Section VI concludes the paper. TwigStack [2] is one of the pattern matching algorithms, which can efficiently retrieve information much faster than many other algorithms [4][5]. TwigStack [2] is optimal for tree pattern queries with only A-D edges. In other words, TwigStack [2] processes the tree pattern holistically without decomposing into several small binary relationships. TwigStack [2] guarantees that there is no useless intermediate result for queries with only Ancestor-Descendant (A-D) relationships.

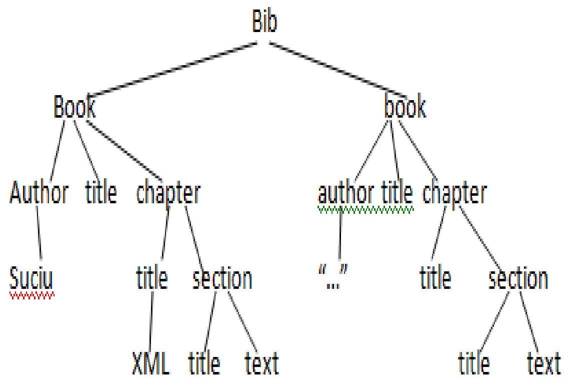


Fig 1, An XML Tree Representation

Algorithm TwigStack operates in two phases. In the first phase (lines 1-11),some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line-12), these solutions are merge-joined to compute answers to the query twig pattern as delineated in fig 4.

```
<!ELEMENT bib (book*)>
<!ELEMENT book (author+, title, chapter*)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>

<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, (text | section)*)>

<!ELEMENT text (#PCDATA | bold | keyword | emph) *>
<!ELEMENT bold (#PCDATA | bold | keyword | emph )*>

<!ELEMENT keyword (#PCDATA | bold | keyword | emph )*>
<!ELEMENT emph  (#PCDATA | bold | keyword | emph )*>
```

Fig 2, An DTD For XML Data

```
//Phase 1
1     while ~end(q)
2         $q_{act}$ = getNext(q)
3         If (~isRoot($q_{act}$ ))
4             cleanStack(parent($q_{act}$ ), nextL($q_{act}$ ))
5         If(isRoot($q_{act}$ ) V ~empty(S parent($q_{act}$ )))
6             cleanStack ($q_{act}$, next($q_{act}$))
7             moveStreamToStack (T $_{qact}$,S $_{qact}$, pointer to
                          top(S $_{parent(q act)}$ ))
8     if (isLeaf(q $_{act}$))
9           showSolution WithBloacking(S $_{qact}$,1)
10          Pop(S$q_{act}$)
11    else advance($T_{qact}$)
//Phase 2
12    mergeAll PathSolutions()
Function getNext(q)
1     if (isLeaf(q))  return q
2     for $q_i$ in children(q)
3         $n_i$=getnext($q_i$)
4         If($n_i$ isnotEqualto $q_i$) return $n_i$
5     $n_{min}$ = minarg $n_i$ ,    nextL(T $_{ni}$)
6     $n_{max}$ = maxarg $n_i$ ,    nextL(T $_{ni}$)
7     while (nextR(Tq) < nextL(T $n_{max}$))
8         advance(Tq)
9     If (nextL(Tq) < nextL(T $n_{min}$)) return q
10    Else return $n_{min}$
```

```
Procedure cleanStack(S, actL)
1       while (~empty(S) and (topR(S) <actL))
2       pop S.
```

Fig 4, TwigStack Algorithm.

III. TWIGSTACK IMPLEMENTATION

TwigStack [2] 1) avoids generating large intermediate results Which do not contribute to the final answer, 2) avoids unnecessary scanning of source documents, 3) avoids unnecessary scanning of irrelevant portions of XML documents. For example, the query is /library/category[@name=*France*]/book/title[@language=*English*]

At each node a stack is maintained by the TwigStack [2] algorithm. A diagrammatic representation of the processing of a query is made in fig.5. And how the data is arranged in the stack in each and every node is presented in fig 6.

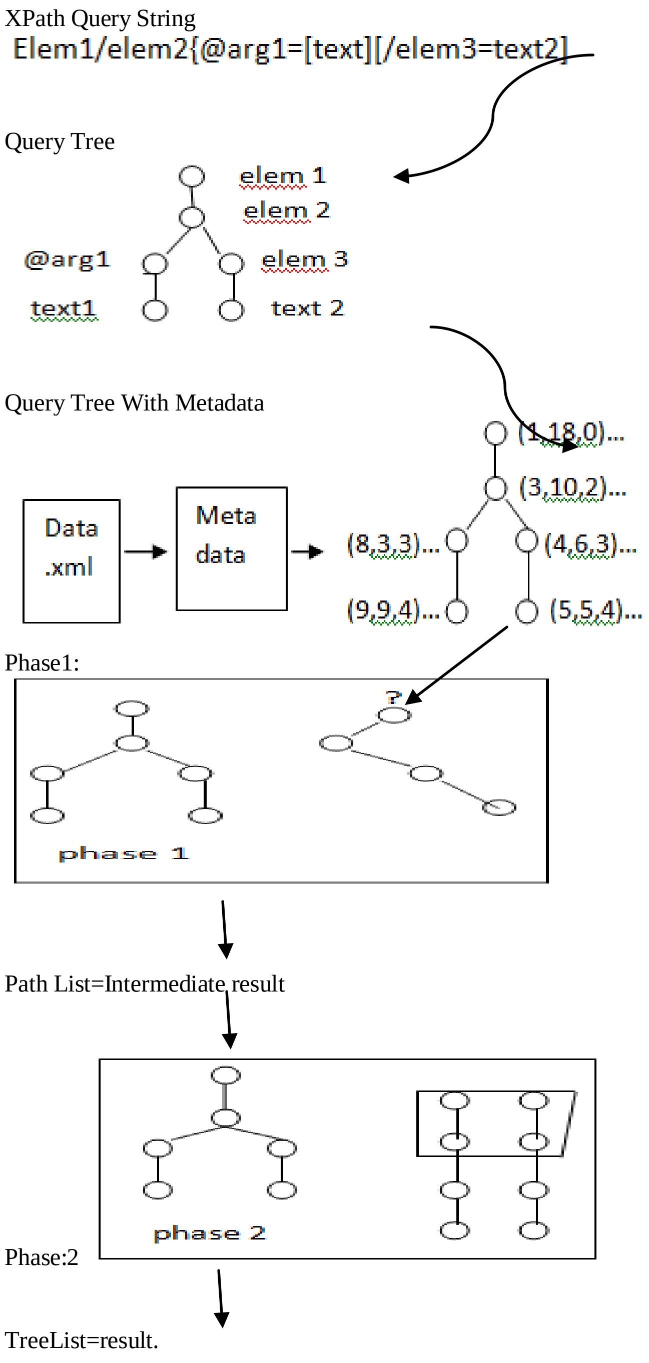
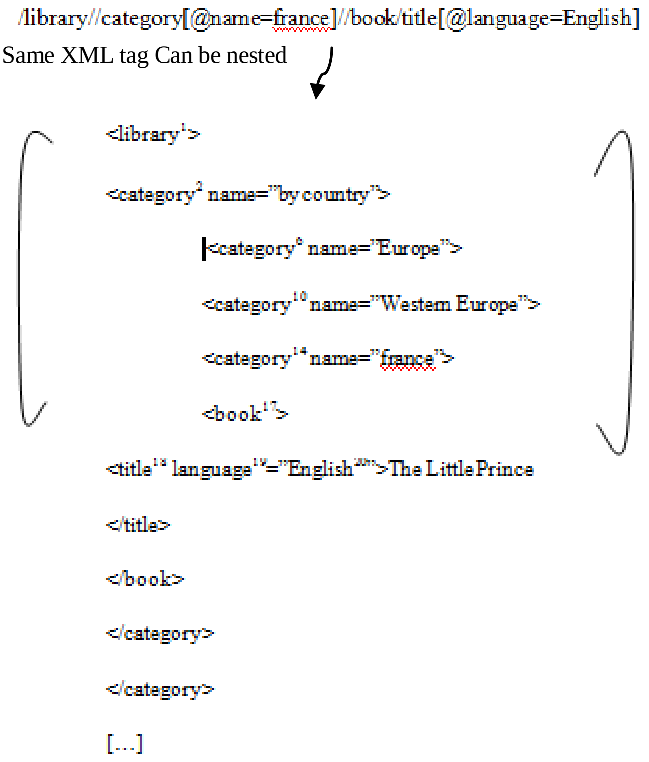


Fig 5, TwigStack implementation.

In fig 5, the TwigStack algorithm comprises two tasks. The first task is to perform query pattern matching against XML data and to generate partial solutions. Meanwhile, the second task is to merge the partial solutions generated by the first task for final solutions.



/library//category[@name=france]//book/title[@language=English]

Same XML tag Can be nested

<library^{1}>

<category^{2} name="by country">

<category^{6} name="Europe">

<category^{10} name="Western Europe">

<category^{14} name="france">

<book^{17}>

<title^{18} language^{19}="English^{20}">The Little Prince

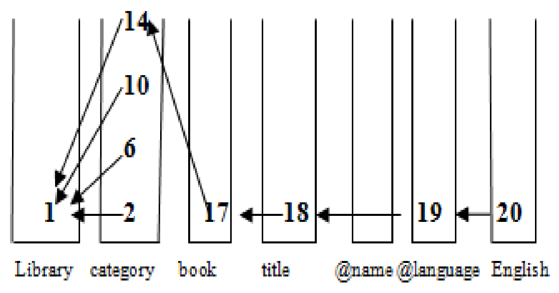
</title>

</book>

</category>

</category>

[...]



Query Node Stack

Fig 6. Arranging of data in stack at each node.

The values of a stack for a query are shown in fig.6 Query Node Stack. The limitations of TwigStack [2] Algorithm are redundancy is maintained, retrieving of data through XML is not much faster than TwigStackList [3], the efficiency of retrieving large queries in XML data is not effective and the intermediate results are not reduced.

IV. TWIGSTACKLIST

TwigStackList [3] is combination of TwigStack [2] and Lists. It improves efficiency of large queries on XML data and overcomes the limitations of redundancy in TwigStack [2]. The tree structure of XML data using TwigStackList [3] is shown in fig 7. At each node the stack and lists are maintained.

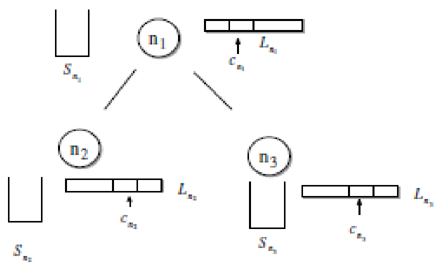


Fig 7, TwigStackList [3] where 'Sn' stands for Stacks and 'Ln' for Lists.

TwigStackList [3] operates in two phases. In the first phase (line 1-11), it repeatedly calls the *getNext* algorithm with the query root as the parameter to get the next node for processing. We output solutions to individual query root-to-leaf paths in this phase. In the second phase (line 12), these solutions are merge-joined to compute the answer to the whole query. The *getNext* algorithm is presented in fig 8 and TwigstackList [3] Algorithm in fig 9.

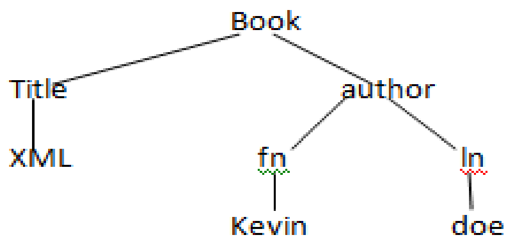


Fig 3, XML tree with A-D and P-C relationships.

At line 2-5, in Algorithm *getNext*, we recursively invoke *getNext* for each n_i *2 children* (*n*). If any returned node g_i is not equal to n_i , we immediately return g_i (line 4). Line 6 and 7 get the *max* and *min* elements for the current head elements in lists or streams, respectively. Line 8 skips elements that do not contribute to results. If no common ancestor for all *C* n_i is found, line 9 returns the child node with the smallest start value, i.e. *gmin* . Line 10 is an important step. Here we look-ahead read some elements in the stream *Tn* and cache elements that are ancestors of *Cnmax* into the list *Ln*. Whenever any element n_i cannot its parent in list *Ln* for n_i *2 children*(*n*), algorithm *getNext* returns node n_i (in line 17). In *TwigStack[2]*, *getNext*(*n*) return *n0* if the head element *en0* in stream *Tn0* has a descendant *e* n_i in each stream Tn_i , for *ni 2 children*(*n0*) and *getNext*(*root*) in *TwigStackList[3]* returns *b*1.

By using TwigStackList Algorithm, we can reduce the intermediate results of a query on xml data, and thereby reduce the redundancy level in TwigStack[2].

Algorithm 1 getNext(n)

```
1    If isLeaf(n) retun n
2    For all node n, n children(n) do
3    $g_i$ = getNext($n_i$)
4    If ($g_i$ isNotEqualTo $n_i$)   return  $g_i$
5    End for
6    $n_{max}$ = maxarg $n_i \leftarrow$ children(n) getStart(ni)
7    $n_{min}$ = minarg $n_i \leftarrow$ children(n) getStart(ni)
8    while ( getEnd(n) < getStart(nmax)) proceed(n)
9    if ( getStart(n) > getStart(nmin)) return $n_{min}$
10   MoveStreamToList(n, $n_{max}$)
11   For all node $n_i$ in PCRchildren(n) do
12   If ( there is an element $e_i$ in listLn such that $e_i$ is the parent of getElement(ni) )  then
13   If($n_i$ is the only child of n)  then
14   Move the cursor $p_n$ of list Ln to point to $e_i$
15   end  if
16   End for
17   Return n
```

Procedure getElement(n)

```
1.   If ~empty(Ln)  then
2.   return Ln.elementAt($p_n$)
3.   Else return cn
```

Procedure getStart(n)

```
1.   return the start attribute of getElement(n)
```

Procedure getEnd(n)

```
1    return the end attribute of getElement(n)
```

Procedure MoveStreamToList(n,g)

```
1    while Cn.start < getStart(g)  do
2    if  Cn.end > getEnd(g) then
2    Ln.append(Cn)
3    end if
4    advance(Tn)
```

```
5    end while
```

Procedure proceed(n)

```
1    if empty(Ln) then
2    advance(Tn)
3    else
4    Ln.delete(Pn)
5    Pn =0  {Move pn to the point to the beginningof Ln}
6    End if
```

fig 8, *getNext* algorithm

Algorithm 2 TwigStackList

```
1    While ~end() do
2    $n_{act}$ = getNext(root)
3    If (~isRoot($n_{act}$)) then
4    cleanparentStack($n_{act}$, getStart($n_{act}$))
5    end if
6    if (isRoot($n_{act}$)V~empty( Sparent ($n_{act}$)) then
7    clearSelfStack($n_{act}$, getEnd($n_{act}$))
8    moveToSatck($n_{act}$,$S_{nact}$,pointertotop($S_{parent(nact)}$)
9    if (isLeaf($n_{act}$) then
10   showSolutionsWithBloacking($S_{nact}$,1)
11   pop($S_{nact}$)
12   endIf
13   else
14   proceed($n_{act}$)
15   endif
16   end while
17   mergeAllPathSolutions
```

Function end()

```
1    return $n_i$   subtreeNodes(n): isLeaf($n_i$) and endC($n_i$)
```

Function moveToStack(n, S_n,p)

```
1    push  (getElement(n),p) toStack $S_n$
2    proceed(n)
```

Procedure clearparentSatck(n, actStart)

```
1    while(~empty$S_{parent(n)}$)
 ^ topEnd($S_{parent(n)}$)<actStart)) do
2    pop($S_{parent(n)}$)
3    end while
```

Procedure clearSelfStack(n, actEnd)

```
1    while (~empty($S_n$) and topEnd($S_n$)<actEnd) do
2    pop($S_n$)
3    end while.
```

Fig 9, TwigStackList Algorithm

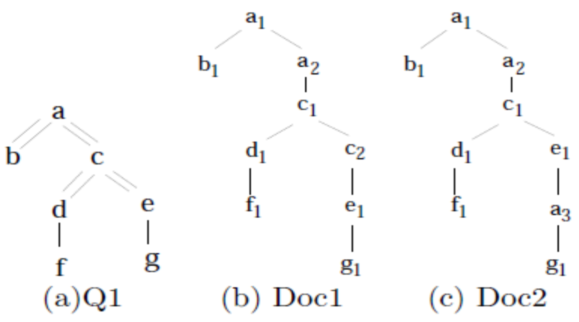


Fig10, Example TwigQuery and Documents

TwigStack [2] pushes *c*1 to *stack Sc* and outputs two *useless*" intermediate path solution <*a*1*; b*1> and <*a*1*; c*1*; d*1*; f*1>. The behavior of TwigStack[2] is also reasonable because based on *region coding* of *g*1, one cannot decide whether *g*1 has the parent tagged with *e*. TwigStackList[3] does not hastily push *c*1 to stack , but first checks the parent-child relationship between *e*1 and *g*1. If *e*1 is not the parent of *g*1, then TwigStackList[3] caches *e*1 in a list and reads more elements in *Te*. In this simple case, *e*1 is the only element in stream *Te*

V. CONCLUSION

The XMl tree construction and importance of pattern matching algorithms for searching the data is discussed. The TwigStack Algorithm has a Time complexity but, the limitation is space complexity. How the TwigStackList overcomes the limitations of TwigStack in reducing the intermediate results in a query on XML data has been elaborated upon.Our experiments have demonstrated that these pattern matching algorithms have an edge over other pattern matching algorithms

REFERENCES

[1]. A. Berglund, S. Boag, and D. Chamberlin, XML Path Language (XPath) 2.0, W3C recommendation, http://www.w3.org/TR/xpath20/, Jan. 2007.

[2]. N. Bruno, D. Srivastava, and N. Koudas, "Holistic Twig Joins: Optimal XML Pattern Matching," Proc. ACM SIGMOD, pp. 310-321, 2002.

[3]. J. Lu, T. Chen, and T.W. Ling, "Efficient Processing of XML TwigPatterns with Parent Child Edges: A Look-Ahead Approach," Proc. 13th ACM Int'l Conf. Information and Knowledge Management (CIKM), pp. 533-542, 2004.

[4]. Q. Li and B. Moon, "Indexing and Querying XML Data For Regular Path Expressions," Proc. Int'l Conf. Very Large Data Bases (VLDB), pp. 361-370, 2001.

[5]. H. Jiang et al., "Holistic Twig Joins on Indexed XML Documents,"Proc. Int'l Conf. Very Large Data Bases (VLDB), pp. 273-284, 2003.