IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661 Volume 4, Issue 5 (Sep-Oct. 2012), PP 19-26
www.iosrjournals.org

Efficiency of TreeMatch Algorithm in XML Tree Pattern Matching

M.Muthukumaran[1], R.Sudha[2]
[1]*Pursuing M.Tech., (Computer Science and Engineering), PRIST University, Tamilnadu, India*
[2] *Asst. Professor, Department of Computer Science and Engineering, PRIST University, Tamilnadu, India.*

Abstract: *In Recent days exchange XML data more often in organizations and business sectors, so there is an increasing need for effective and efficient processing of queries on XML data. This paper presents a wide analysis to identify the efficiency of XML tree pattern matching algorithms. Previous years many methods have been proposed to match XML tree queries efficiently. In particularly TwigStack, OrderedTJ, TJFast and TreeMatch algorithms. All algorithms to achieve something through these own ways like structural relationship including Parent – Child (P-C) relationship (denoted as '/') and Ancestor-Descendent (A-D) relationships (denoted as '//') and more. Finally, we report our results to show that which algorithm is superior to previous approaches in terms of the performance.*
Keywords *– Efficient TPQ, Efficiency of Tree pattern, XML Tree pattern*

I. Introduction

The extensible markup language XML has recently emerged as a new standard for information representation and exchange on the internet [1]. With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data. Since the data objects in a variety of languages (e.g XPath [2], XQuery [3]) are typically trees, twig (A small tree) pattern matching is the central issue. XML data may be very large, complex and have deep nested elements. Thus, efficiently finding all patterns in an XML database is a major concern of XML query processing.

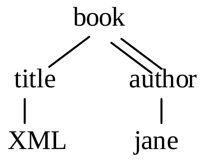


Fig 1: XPath query

An XML query pattern commonly can be represented as a rooted, labeled tree (Twig), for example Fig 1 shows an example XPath query:

Book [title = 'XML'] // author [. = 'jane']

Such a complex query tree pattern can be naturally decomposed into a set of basic P-C and A-D relationship between pairs and nodes [4]. The above example query are the ancestor-descendent relationship (book, author) and the parent-child (book, title) and (title, XML) and (author, jane).

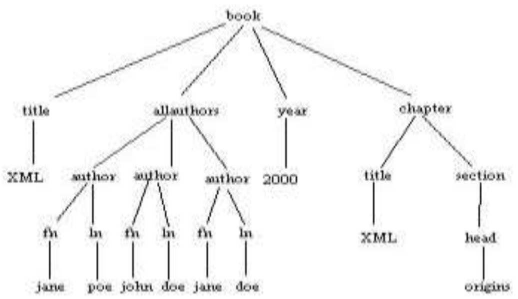


Fig 2: sample XML tree

XML twig pattern algorithms is a selection predicate on multiple elements in an XML document. Such query patterns can generally be represented as node - labeled trees. Matching a twig pattern against an XML database is to find all occurrence of the pattern in the database. For example given a query twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from nodes in Q to nodes in D such that

(i). query node predicates are satisfied by the corresponding database nodes. (ii). The structural relationships [4] between query nodes are satisfied. The query twing pattern in fig 3 and the database tree in fig 2. This query twig pattern has one match in the data tree that maps the nodes in the query to the root of the data and its first and third sub trees.

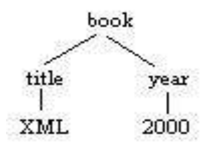


Fig 3: Query twig pattern

II. XML Twig pattern matching algorithms

An XML query contains two parts one is value match and another one is twig match. The above XPath query(fig 1) contains 'XML' is a value match and another is a twig match. Labeling and Computing is the main view of the twig pattern, labeling assign each element in the XML document tree an integer label to capture the structural information of documents and computing use labels to answer the twig pattern without traversing the original document. Mainly there are two labeling schemes, such as containment labeling schemes [5] and Dewey ID labeling schemes [6]. Several algorithms based on the containment labeling scheme have been developed to process twig queries. Prior work on XML twig pattern processing decomposes a twig pattern into a set of binary relationships which can be either parent - child or ancestor - descendant relationships. After that, each binary relationship is processed using structural join techniques and the final match results are obtained by merging individual binary join results together. The main problem with the above solution is that it may generate large and possibly unnecessary intermediate results because the join results of individual binary relationships may not appear in the final results.

The following sections we going to comparative analysis about few existing tree pattern matching techniques in particularly TwigStack, OrderedTJ, TJFast with TreeMatch [5][7][8][9].

III. TwigStack

Based on the containment labeling scheme, Bruno et al. [5] proposed a novel "holistic" XML twig pattern matching method called TwigStack. When all edges in query pattern are ancestor – descendant (A-D) relationships, Twigstack ensures that each root – to – leaf intermediate solution is merge – joinable.

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. To get a better understanding of this limitation, let us take an experimented with TreeBank datasets [10] tested three twig queries patterns (as shown in Table 1), each of which contains at least one P-C edge. TwigStack operates two steps: 1. a list of intermediate path solutions is output as intermediate results and 2. the intermediate path solutions in the first step are merge-joined to produce the final solutions.

Table 1: number of intermediate path solutions produced by TwigStack against treebank data

Query	**Output paths**	**Useful paths**	**Useless paths**
*V P[./DT]//PRP_DOLLAR_*	10663	5	99.9%
S[./JJ]/NP	70988	10	99.9%
S[.//VP/IN]//NP	702391	22565	96.8%

An immediate observation from the table 1 is that TwigStack outputs many intermediate paths that are not merge-joinable [5]. For all three queries, more than 95% intermediate paths produced by TwigStack in the first step are "useless" to final answers [11]. The main reason for such bad performance is that in the TwigStack, it assumes that all edges in queries are A-D relationships and therefore output many useless intermediate results when queries contain P-C relationships. TwigStack cannot answer queries with wildcards in branching nodes. For example in Fig 4, the parent of B should be an ancestor of C

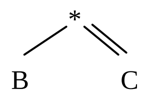


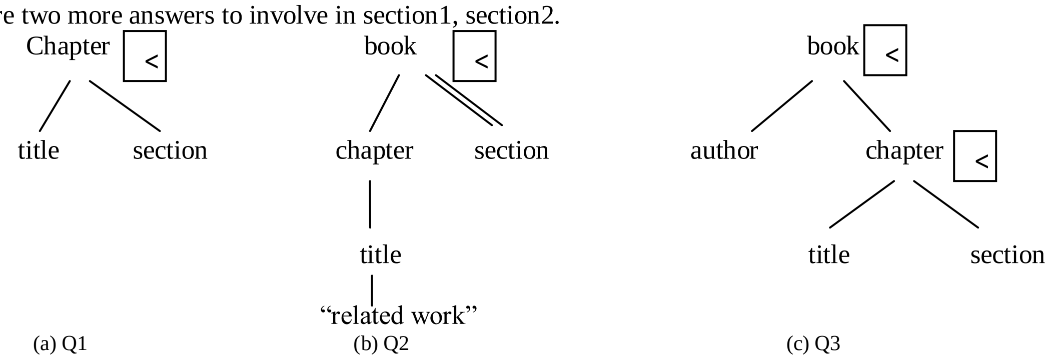
Fig 4: queries with wildcard

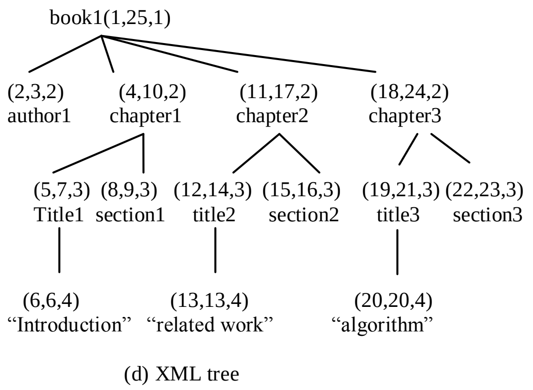
IV. OrderedTJ

In OrderedTJ [7], an element contributes to final results only if the order of its children accords with the order of corresponding query nodes. 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 ancestor - descendant relationship from the second branching edge, OrderedTJ is I/O optimal among all sequential algorithms that read the entire input. In other words, the optimality of OrderedTJ allows the existence of parent-child edges in non-branching edges and the first branching edge. The results show that the effectiveness, scalability and efficiency of holistic twig algorithms for ordered twig pattern.

Given an ordered twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from the nodes in Q to the elements in D, such that: (i) the query node name predicates are satisfied by the corresponding database elements; and (ii) the parent-child and ancestor-descendant relationships between query nodes are satisfied by the corresponding database elements; and (iii) the order of query sibling nodes are satisfied by the corresponding database elements. In particular, based on the containment labeling scheme, given any query node q and its right-sibling r (if any), their corresponding elements, say e_q and e_r, must satisfy that e_q.end<e_r.start. In other words, we do not allow e_q to be an ancestor of e_r. The answers to query Q with n nodes can be represented as a list of n-ary tuples, where each tuple (e_{q1} ,e_{q2},...,e_{qn}) consists of the database elements that identify a distinct match of Q in D.

Fig 5(a-c) show three example ordered twig patterns based on the data tree of Fig 5 (d). For each branching node, we used a symbol "<" in a box to mark its all children ordered. For example, the query solution for Q2 is only (book1, chpater2, title2, "related work", section3). Note that if Q2 were an unordered query, then there are two more answers to involve in section1, section2.





(d) XML tree

Fig 5: example ordered twig query and an XML tree

OrderedTJ output much less intermediate results, OrderedTJ scales linearly with the size of the database, OrderedTJ is not optimal and outputting **less** useless intermediate results.

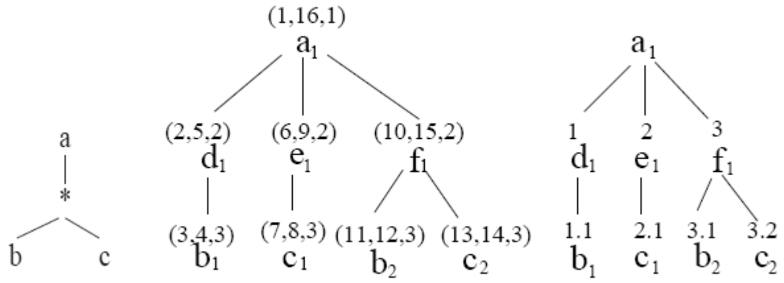
V. TJFast

We have presented two holistic algorithms for answering XML twig queries in previous sections. Interestingly, all these two algorithms use the same containment labeling scheme. While the containment scheme preserves the positional information within the hierarchy of an XML document, we observe that this is not the only labeling scheme that can be used for XML twig query processing. Indeed, there are at least two limitations in the containment scheme.

1. The information contained by a single containment label is very limited. For example, we cannot get the path information from any single containment label.

2. While wildcard steps in XPath are commonly used when element names are unknown or do not matter[12].

The containment labeling scheme is difficult to answer queries with wildcards in branching nodes. 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. This is because even if b and c are descendants of a and their level difference with a is 2, b and c may not be query answers, as they do not have the common parent.



(a) query (b) containment (c) Dewey ID

Fig 6: wildcard query processing

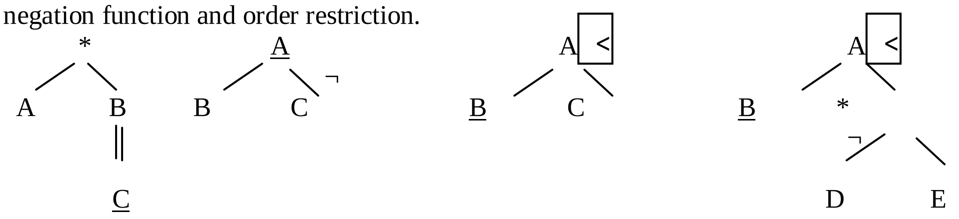
However,Dewey ID [77] labeling scheme can efficiently overcome the above two limitations. In DeweyID [6], each element is labelled by a vector to show the path from the root to this element. Fig 6(c) shows the example XML data with Dewey ID labeling scheme. From this figure, we see that b_1("1.1") and c_1("2.1") have not the same parent, for their prefixes are not the same (i.e. $1 \neq 2$). This example shows that unlike containment, the Dewey ID labeling scheme can provide path information and thus support the evaluation of queries with wildcards in branching nodes.

TJFast outputs one useless intermediate path and it is outputs the path solution for all nodes in query [9]. It does not produce the individual solution for each node when there are multiple return nodes in a query. TJFast cannot work with ordered restriction and negation function [9].

VI. Introduction to TreeMatch

Previous XML tree pattern matching algorithms do not fully exploit the "optimality" [9] of holistic algorithms. TwigStack guarantees that there is no useless intermediate result for queries with only AD relationships. Therefore, TwigStack is *optimal* for queries with only A-D edges.

Previous algorithms focus on XML tree pattern queries with only P-C and A-D relationships. Little work has been done on XML tree queries which may contain wildcards, negation function and order restriction, all of which are frequently used in XML query languages such as XPath and XQuery. In this analysis, we take an XML tree pattern with negation function, wildcards and/or order restriction as *extended XML tree pattern*. Fig 7, for example, shows four extended XML tree patterns. Query (a) includes a wildcard node "*", which can match any single node in an XML database. Query (b) includes a negative edge, denoted by "¬". This query finds *A* that has a child *B*, but has *no* child *C*. In XPath language [2], the semantic of negative edge can be presented with "*not*" Boolean function. Query (c) has the order restriction, which is equivalent to an XPath "//*A*/*B*[following-sibling::*C*]". The "<" in a box shows that all children under *A* are ordered. The semantics of order-base tree pattern is captured by a mapping from the pattern nodes to nodes in an XML database such that the *structural* and *ordered* relationships are satisfied. Finally, Query (d) is more complicated, which contains wildcards, negation function and order restriction.



(a) TQ1 (b) TQ2 (c) TQ3 (d) TQ4

Fig 7: Example extended XML tree pattern queries. " _" denotes the return node in query

Xpath expressions: TQ1: //*[A]/B//C, TQ2://A[B][not (C)], TQ3://A/B[following-sibling::C] and TQ4://A/B[following-sibling::*[not(D)]/E]

Based on the theoretical analysis, We studies a series of holistic algorithms with TreeMatch [9] to achieve a best performance of the tree pattern matching algorithms.

6.1 Analysis of TreeMatch

Now we go through Algorithm 1. 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 line 4, 5 is to insert the potential matching elements to *outputlist*. Line 6 advances the list Tf_{act} and line 7 updates the set encoding.

Algorithm 1: Algorithm TreeMatch for class *Q/,//,**

```
1 locateMatchLabel(Q);
2 while (¬end(root)) do
3    $f_{act}$= getNext(topBranchingNode);
4    if ($f_{act}$ is a return node)
5       addToOutputList(N A B($f_{act}$ ), 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);
```

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.

Algorithm 2: Procedures and Functions in TreeMatch

```
1 Procedure locateMatchLabel(Q)
   1: for each leaf q Є Q, locate the extended Dewey
      label $e_q$ in list $T_q$ such that $e_q$ matches the
      individual root-  leaf path
  Procedure addToOutputList(q,$e_{qi}$)
   1: for each $e_q$ Є  $S_q$ do
   2:         if (satisfyTreePattern($e_{qi}$,$e_q$))
   3:               outputList($e_q$).add($e_{qi}$);
  Function satisfyTreePattern($e_{qi}$,$e_q$)
   1: if (bitVector($e_q$,$q_i$) = '1') return true;
   2: else return false;
  Procedure updateSet(q,e)
   1: cleanSet(q,e);
   2: add e to set $S_q$; //set the proper bitVector(e)
   3: if (¬is Root(q) Λ (bitVector(e)="1…1"))
          updateAncestorSet(q);
  Procedure cleanSet(q,e)
   1: for each element $e_q$ Є $S_q$ do
   2:      if (satisfyTreePattern($e_q$,e))
   3:          if (q is a return node)
   4:             addToOutputList( N A B(q),e);
   5:          if (isTopBranching(q))
   6:              if (there is only one element in $S_q$)
   7:              output all elements in outputList($e_q$);
   8:              else merge all elements in outputList($e_q$)
        To outputList($e_a$), where $e_a$=NAB($e_q$);
   9: delete $e_q$ from set $S_q$;
  Procedure updateAncestorSet(q)
   1: /*Assume that q' = NAB(q)*/
   2: for each e Є $S_{q'}$ do
   3:     if (bitVector(e, q) = 0)
   4:        bitVector(e, q) = 1;
   5:        if (¬is Root(q) Λ (bitVector(e)="1…1"))
   6:              updateAncestorSet(q');
  Procedure emptyAllSets(q)
   1: if (q is not a leaf node)
   2:      for each child c of q do EmptyAllSets(c);
   3: for each element e Є Sq do cleanSet(q,e);
```

In Procedure *addToOutputList*(q; e_{qi}), it add the potential query answer e_{qi} to the set of S_{eq}, 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 *bitVector* values of ancestors of q need update.

Algorithm *getNext*(see Algorithm 3) 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, then suppose element e_i matches node n in the corresponding path solution(if more than one element that matches n, e_i is the *deepest* one by level)(line 7,8), we return f_{min} such that the current element e_{min} in $T_{f_{min}}$ has the minimal label in all e_i by lexicographical order(line 13,20) For the second task of *getNext*, before an element e_b is inserted to the set S_b, we ensure that e_b is an ancestor (or parent) of each other element e_{bi} to match node b in the corresponding path solutions (line 13). If there are more than one element to match the branching node b, e_{bi} is defined as their *deepest*(i.e. maximal) element(line 8).

Algorithm 3: getNext(n)

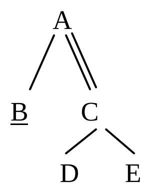
```
1:  if (isLeaf($n$)) then
2:       return $n$
3: else
4:    for each $n_i \in$ NDB($n$) do
5:        $f_i$ = getNext($n_i$)
6:        if ( isBranching($n_i$) $\wedge \neg empty(S_{ni})$ )
7:             return $f_i$
8:        else $e_i = max\{p|p \in MB(n_i, n)\}$
 9:    end for
10:   max = $maxarg_i\{e_i\}$
11:   for each $n_i \in$ NDB($n$) do
12:      if ($\forall e \in MB(n_i, n) : e \notin ancestors(e_{max})$)
13:          return $f_i$;
14:      endif
15:    end for
16:     min = $minarg_i\{f_i | f_i$ is not a return node$\}$
17:     for each $e \in$ MB($n_{min}$, $n$)
18:       if ($e \in$ ancestors($e_{max}$) ) updateSet($S_n$, $e$)
19:     end for
20:      return $f_{min}$
21:  end if
```

Function MB(n; b)

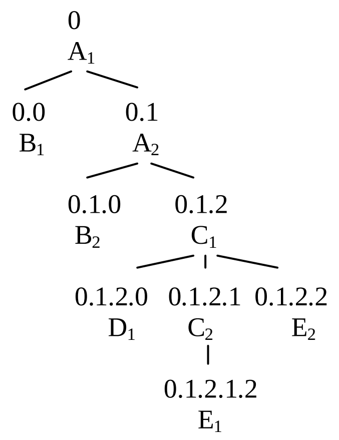
```
1:  if (isBranching($n$)) then
2:     Let $e$ be the maximal element in set $S_n$
3:  else
4:     Let $e$ = cur($T_n$)
5:  end if
6: Return a set of element $a$ that is an ancestor of $e$ such
   that $a$ can match node $b$ in the path solution of $e$ to
   path pattern $p_n$
```

Example : We use the query and document in Fig 8 to illustrate TreeMatch algorithm. Table 2 demonstrates the current access elements, the sets encoding and the corresponding output elements. There are two branching nodes in the query. Firstly, $B1$, $D1$ 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 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 *B*2 is scanned, *A*2 is added to set *SA*. Therefore, Treematch outputs two final results *B*1 and *B*2. Note that there are no useless nodes output here.



(a) query



(b) data

Fig 8: illustration to Algorithm TreeMatch for class Q/,//,*

Through this example, we illustrates 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.

Table 2: set encoding for the example in fig 8

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

6.2 Comparative analysis table of previous algorithms with TreeMatch

Table 3: Summary of algorithm analysis

Algorithms	**Labeling scheme**	**Optimality**	**Query**	**Output list**
TwigStack	Containment	optimal in terms of output sizes and not optimal for PC	Unordered	Many useless intermediate results when queries contain P-C relationships
OrderedTJ	Containment	Not a optimal	Ordered	much less intermediate results
TJFast	Extended Dewey	Not fully optimal	Unordered	one useless intermediate path and it is outputs the path solution for all nodes in query
TreeMatch	Extended Dewey and bitvector	Fully optimal	Ordered restriction, Negation and wildcard	No useless paths

Based on previous detailed discussions, table 3 illustrates the comparative analysis of previous tree pattern matching algorithms with TreeMatch with the key factors of labeling schemes, optimality, query and output list.

VII. Conclusion

In this paper, we proposed the problem of XML twig pattern matching and surveyed some recent works and algorithms. TreeMatch has an overall good performance in terms of labeling schemes, optimality, query processing, outputlist (table 3) and the ability to process extended XML tree patterns (twigs). The previous twig pattern matching algorithms (TwigStack, OrderedTJ and TJFast) requires more features than TreeMatch algorithm. TreeMatch to achieve such optimal query classes so, from this points we can say that TreeMatch twig pattern matching algorithm can answer complicated queries and has good performance.

VIII. Acknowledgments

We feel very humble and happy to thank below mentioned authors (reference paper's) who initiated and motivated us to research and develop knowledge. And also we feel happy to dedicate our work credit to those authors for getting interest on these topics and give enthusiasm to do future enhancements.

References

[1] Tim Bray, Jean Paoli, C.M. Sperberg -McQueen and Eve Maler. *Extensible markup language (XML) 1.0 second edition W3C recommendation. Technical report RSC-XML-20001006, World Wide Web consortium, October 2000.*

[2] W3C. *XML Path Language (XPath) 1.0*. "http://www.w3.org/TR/xpath", 1999.

[3] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. "http://www.w3.org/TR/xquery", November 2003.

[4] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. *Structural joins: A primitive for efficient XML query pattern matching. In ICDE, pages 141–152, February 2002.*

[5] N. Bruno, D. Srivastava, and N. Koudas, *Holistic twig joins: optimal XML pattern matching, In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2002, pp. 310–321.*

[6] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang, *Storing and querying ordered XML using a relational database system, In Proceedings of the ACM SIGMOD International Conference on Management of Data, 2002, pp. 204–215*.

[7] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. *Efficient processing of ordered XML twig pattern matching. In DEXA, pages 300–309, 2005.*

[8] J. Lu, T. Chen, and T. W. Ling. *TJFast: Efficient processing of XML twig pattern matching. Technical report, National university of Singapore, 2004.*

[9] J. Lu, T. W. Ling, Z. Bao, and C. Wang. *Extended xml tree pattern matching: theories and algorithms. IEEE transactions on knowledge and data engineering, vol.23, no. 3, march 2011*

[10] H. V. Jagadish and S. AL-Khalifa, Timber: *A native XML database, Tech. report, University of Michigan, 2002*.

[11] Lu Jiaheng, Efficient Processing Of Xml Twig Pattern Matching, doctoral diss., *Shanghai Jiao Tong University, China, 2006*

[12] C Y Chan, W Fan, and Y Zeng, *Taming xpath queries by minimizing wildcard steps, Proceedings of 30th International Conference on Very Large Data Bases, 2004, pp. 156–167.*

Authors profile

[1]**M.Muthukumaran** pursuing M.Tech (CSE) in PRIST University, Tamilnadu, India. He has received the Master of Computer Applications degree at the University of Madras, Chennai, India. He has more than six years experience in IT industry and two years experienced as Assistant professor. His area of interest is Data warehouse, Data Mining and Data Engineering.

[2]**R.Sudha** received her Master of Engineering in Computer Science and Engineering from Pavendar Bharathidasan College of Engineering Under Anna University, India. She is Currently working as Assistant professor in Prist University,Trichy. She has Published a paper titled " A System Tool for Identification of RAGAS using MIDI (Musical Instrument Digital Interface) for CMIR (Classical Music Information Retrieval)" in International Conference held in Dubai 28-30,2009. Published a paper titled "Adaptive Location Aided Routing in Mobile ad-hoc Network(ALARM)" in a National Conference held in PSNA college, Dindugal 2006. And her area of Interest includes Multimedia (Musical Information Retrieval), Database and Networking.