International Journal of Computer Network and Security(IJCNS)
Vol 4. No 1. Jan-Mar 2012 ISSN: 0975-8283
www.ijcns.com

A Novel XML Documents Using Clustering Tree Pattern Algorithms

[1]N.Kannaiya Raja, *M.E., (P.hd),*[2]Dr. K.Arulanandam, *Prof and Head,*[3]P. Umadevi, *M.E.,(A/P),* [4]A.Balakrishnan, *M.E*
CSE Department
Arulmigu Meenakshi Amman College of Engg, Thiruvannamalai Dt, India
kanniya13@hotmail.co.in
sathisivamkva@gmail.com
umasri05@yahoo.co.in
drbalaphd1687@yahoo.com

Abstract

In a business enterprises generate and exchange XML data, are used more often for increasing the demand of efficient processing of queries on the XML. The searching for the occurrences of tree pattern query are on XML database is a core operation in XML query process which meets more problems in holistic algorithm, also demonstrated is an efficient technique to suggest XML-tree pattern with parent-child operations. However, XML query have more functions such as negation function, order based axis, and wildcards and also created and invented extended XML tree pattern matching have implement a good relationship between negation function, wildcards function and ordered restriction. In this paper, we research a large set of *xml clustered tree pattern* which may have parent-child relation on top-down approach. We also established framework on multiple matching pattern with strong demonstrate for proof of multiple holistic algorithm based on our theorem, we proposed a set of efficient process for three categories of *xml clustering pattern algorithms* set of experiment on both real life and synthetic dataset demonstrated with effectiveness and efficiency of our proposed theory of algorithms. Index term: *Algorithm, XML, Clustering tree pattern, Query processing.*

1. INTRODUCTION

The growing important of XML data more often inducing the needs for efficient process of XML clustering pattern algorithm on xml data. XML clustering pattern commonly represented has a rooted, labeled XML query tree are used in the XML data for example, Xpath query. The effective matching of XML clustering pattern algorithm has more often as multi operation in XML query processing. In particular, a stack-based algorithm to match binary structural relationship including parent-child (P-C) and ancestor-descendant (A-D) relationship. The limitation of parent-child and ancestor-descendant relationship is the size of intermediate results may become very large, even if the final results are small. A novel holistic Twig join algorithm named *twigstack*, *Twigstack* guarantees there are no "useless" intermediate requests for queries with only (A-D) relationship. In recent works that examine how to enlarge the query class of holistic algorithm. These algorithms have proven highly promising and make their way into XML query processing.

XML clustering pattern algorithm with XML query that prior algorithm focus on XML clustering pattern queries with only (P-C) and (A-D) relationships. The small works are done on XML clustering queries which may contain wildcards, negation function and order restriction. All of the functions used in the XML query language such as Xpath and Xquery.

In this paper, we have call an XML clustering pattern algorithm with XML query which include negation function, wildcards and order restriction as extended XML clustering patterns. Query (a) include a wildcard node such as "α", which can match any single node in an XML database. Query (b) which can include a negative edge, denoted by" β ". This query finds out the "A" that has a child "B", but has no child "C".

In Xpath language, the negative edge can be represented by "not" Boolean function. Query(C) it has the order restriction, it is equal to an "Xpath" the "γ" shows in a children under 'A' are ordered. Finally (d) which is very complexity, which contains wildcards, negation form and order restriction. Opmality of the holistic algorithm prior XML clustering pattern algorithm with XML query do not completely exploit the "optimality" of the holistic algorithms. *Twistack* which guarantees for very useful intermediate result for the queries with only A-D relationship. Another algorithm *twigstack* list which enlarges the optimal query class of the *twigstack* by including P-C relationship. Another important question is whether *twigstack* list can be improved (or) not. Hence, the current problem which includes how to find a large query class which can be processed optimally and also how to effectively answer a query which cannot be guaranteed to process optimally.

Notify that prior works is no algorithm is optimal for queries with (A-D) and (P-C) relationship. In this paper, to explore the framework called "matching" to find out the large optimal query class. Twig pattern queries that the practical application is only part of query nodes belong to the return nodes called output nodes. Take the Xpath "\\A [B]\\C" as an example, only 'C' elements and its sub tree are answers. In this paper, we have to develop a new encoding method to file

the mapping relationship and avoid output non return nodes. Main result that in general, it has given an *extended XML clustering pattern algorithm* which may include (P-C, A-D) relationship, Order restriction, negation function and wildcards. We have to consider the problem effectively matching the extended XML clustering pattern algorithm. The main theme of our algorithm is to identifying a large queries class which can be optimally processed. Like existing papers on XML Tree pattern matching. But in this paper, we can calculate a holistic algorithm "optimal" for the different kind of query class. If it guarantees that output.

Optimality of holistic algorithm that prior XML clustering pattern algorithms with XML query do not completely exploit the "optimality" of holistic algorithms. *Twigstacks* which guarantees for very useful intermediate result for queries with only (A-D) relationship. Another algorithm twig stack list which enlarges the optimal query class of twig stack by including (P-C) relationship. Another important question is whether *twigstack* list can be improved (or) not. Hence the current problem which includes how to find a large query class which can be processed optimally and also how to effectively answer a query which cannot be guaranteed to process optimally. Notify that prior works if there is no algorithm is optimal for queries with (A-D) and (P-C) relationship. In this paper, to explore the framework called "matching" to find out the large optimal query class.

Intermediate results contribute to final result. We can find out the 3 categories of extended XML clustering pattern algorithm (1) queries with (P-C) , (A-D) relationship, wildcards and order restriction, denoted as Z\,\\,α; and (2) queries with (P-C),(A-D) relationship, wildcard, and order restriction, denoted as Z\,\\,α,β; and (3) queries like (P-C) , (A-D) relationships, wildcard, negation functions denoted as Z\, \\,α,β,γ. For each category we have to find out the respective optimal query class.
The technical contribution of this paper is summarized as follows:

- We have created a theoretical framework on optimal processing of XML clustering pattern queries. We can show that "Matching" is the key of result in the sub optimality of the holistic algorithm. *Twigstack* is a fact that optimal for queries with only (A-D) relationship can be explained that no matching cross can be found for any XML document.
- Based upon our theoretical analysis, we can create a series of holistic algorithm match to achieve a large optimal query class for '3'categories of queries (i.e.\, \\, α; Z\, \\, α, β; and Z\, \\, α, β, γ).
- We conducted a set of synthetic and real data set for performance comparison. We compared true match with prior four holistic XML clustering pattern matching algorithms with XML query. The results show that our algorithm can correctly process extended XML clustering pattern. We can develop mainly for the reduction in the site of the intermediate results.

The Extensible Markup Language (XML) has become a standard for data representation. With the continuous growth in the XML data, the ability to manage massive collections of XML data and to discover knowledge from them becomes essential for the Web-based information systems [4,6]. A possible solution is to group the similar XML data based on their context and structure. The clustering of XML data facilitates a number of advanced applications such as improved information retrieval, data and schema integration, document classification analysis, structure summary and indexing, and query processing and optimization [3,5]. In this paper we define a new method for computing the similarity between any two XML documents in terms of their structure. The higher this similarity, the more similar the two documents are in terms of structure, and the more likely they are to have been created from the same DTD.

Crafting a good similarity metric for this setting is somewhat difficult since two documents created from the same DTD can have radically different structures (due to Nesting and repeating elements), but we would still want to compute a higher similarity between these documents. We account for this by introducing Nesting reduction and repeating reduction method in all sections of the document. Using our resulting Weight Edge-set similarity comparison (WESC) measure, we show that standard clustering algorithms do very well at pulling together documents derived from the same DTD.

Outline that the remaining paper shows the preliminaries about research problem and processing modal. Section (3) gives the set if theories about matching cross and Section (4) give the extended XML clustering pattern algorithm called tree match. Section (5) presents experimental studies b/w the novel algorithm and existing method. Finally, section (6) gives existing work related on the XML clustering pattern algorithm.

2. RELATED WORKS

A. Clustering XML PATTERN

We have to be deal with the problem of clustering XML documents using such as namely (1) the XML structural is ordered labeled trees, (2) similarity calculated from these tree and (3) clustering algorithms. Clustering methods are divided into mainly two types. Hierarchical and Non-hierarchical methods. A non-hierarchical method which consists of data set into a number of clusters. Hierarchical methods provide a nested sets of data in which pairs of elements or clusters are connected successively until every element in the data set becomes linked. Nonhierarchical methods are low computational requirements because the

value is (O (kn) for example n records need to be grouped into k clusters, the parameters like the number of formed clusters are known as priori. Hierarchical methods are computationally expensive because the value is O (n2) and n documents need to be clustered. The hierarchical methods are used to increasing the effectiveness and efficiency of recovery [8–10]. The wide ranging of clustering methods you can refer to [11, 12]. In this paper, we have to select the Hierarchical methods.

B. Clustering experimental

We have to test the performance of overall quality of the clustering results using Artificial and original data. In which original data are used records from the ACM SIGMOD database, which depends on three DTDS. Artificial XML records are generated by using IBM's Alpha Works XML generator, which is depends on six DTDS. All the experiments are performed on a PC, Pentium(R) D 2.80GHZ, 1.24GB RAM, using the HTML programming language. Figure 1 presents the structure of the programme. We have to show the result on Table 1 based upon on the nine DTDS.
In general, at each node in the query tree pattern, that specifies the node predicate on the attributes e.g., tag.

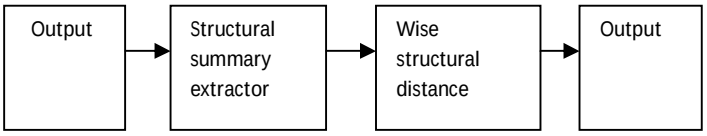

Figure.1.Programme structure

Using WESC and the type of Selkow. From the result we can see WESC development accuracy on the clustering. The overall performance on the time cost, mainly for the large XML documents.

Table 1
Experimental result

Number of docs	1150	1150	930	930
Number of DTDS	9	9	9	9
Average size	0.83kb	0.83kb	4.2kb	4.2kb
algorithm	WESC	selkow	WESC	selkow
Number of clustering	9	9	9	9
Accuracy rate	0.975	0.873	0.949	0.812
Recall rate	0.978	0.881	0.937	0.911
Time consuming	5.3s	9.8s	8.2s	14.6s

C. Data Model and Query Patterns

Node labels are a set of attribute and value pairs, which suffices to form tags, PCDATA content, etc. The XML database consisting of the ordered, labeled trees and rooted forest, each node representing the element and the edges corresponding element-Subelement relationships.

From the sample XML document of Figure 2 and its tree representation is shown in Figure 3. Queries in XML query languages like XQuery, Quilt [7], and XML-QL make fundamental use of node labeled tree patterns for matching related portions of data in the XML database. The query pattern labels which consists of element tags, attribute-value comparisons and string values, and the query pattern edges which include the parent-child edges "using sole line" or ancestor-descendant edges "using a dual line". For example, the XQuery path looks in the represented of the embedded tree pattern in Figure 3(a). This query pattern would match the document in Figure 2. In general, at each node in the query tree pattern, that specifies the node predicate on the attributes e.g., tag, content of the node .In this paper, exactly what is permitted in this predicate is not material. The well-organized access of mechanism that constructing the suffices such as index structures to find the XML database nodes which satisfies the predicate nodes.

D. Matching Basic Structural Relationships

The query pattern can be matched by (i) binary structural relationship should be matched against to the XML database, and (ii) "stitching" collectively these basic matches. A difficult query tree pattern decomposed into a basic binary structural relationship such as parent-child and ancestor-descendant between relationships of nodes. For example, the basic structural relationships matching to the query tree pattern of Figure 3(a) are shown in Figure 3(b).

A structural relationships should match the straightforward approach against an XML database is to use traversal-style algorithms by using child-pointers or parent-pointers. Such "tuple-at-a-time" processing strategies are known to inefficient compared to the set-at-a-time strategies used in database systems. Pointer-based joins have been optional solution to this problem in object-oriented databases and shown as quite well-organized.

In the framework of XML databases which may have a large number of children nodes and the query pattern are often to the matching ancestor-descendant structural relationships (for example, the (book, author) edge in the query pattern of Figure 4, in addition to parent-child structural relationships. In this case, there are two options: (i) explicitly maintaining only (parent, child) node pairs and identifying (ancestor, descendant) node pairs through repeated joins; or (ii) explicitly maintaining (ancestor,

descendant) node pairs. A large amount of query processing time would have to use the previous approach, although the later approach would take to use a

```
<book>
<title> XML < =title>
<all authors>
        <author> jane < =author>
        <author> john < =author>
< =all authors>
<year> 2000 < =year>
<Chapter>
        <head> Origins < =head>
        <Section>
                <head> ...< =head>
            <section> ...< =section>
             < =section>
            <section> ...< =section>
          < =chapter>
        <chapter> ...< =chapter>
                        < =book>
```

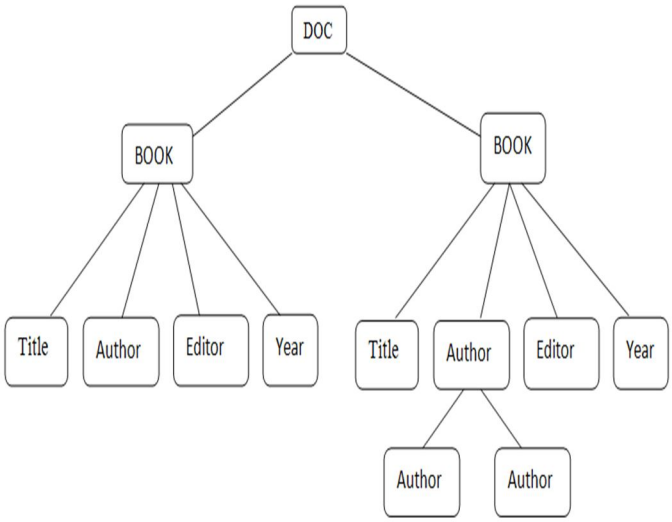

Figure. 2. Sample XML document

Figure .3.Tree representation

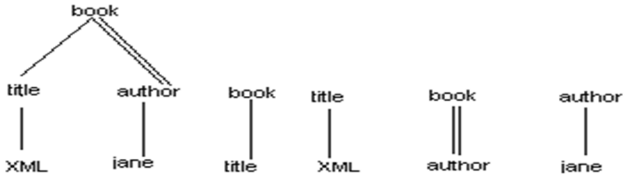

Figure .4. Structural relationships

much quadratic space. In also using pointer-based joins is likely to be infeasible.

E. Representing Positions of Elements and String Values in an XML Database:

The well-organized key to an uniform mechanism for set-at-a-time (join-based) matching of structural relationship is a positional representation of occurrences of XML elements and string values in the XML database e.g., [9, 29], which extend the classic reversed index data structure in information recovery .

The element of an XML database is occurrence by the represented as the 3-tuple DocId, SPos : EPos, LevelNo, and the location of a string occurance in the XML database can be represented as the 3-tuple DocId, SPos, LevelNo, where (i) DocId is the identifier of the document; (ii) SPos and EPos can be created by counting word numbers from the beginning of the document with identifier DocId awaiting the start of the element and end of the element respectively and (iii) LevelNo is the nesting depth of the element in the text. Figure 3 depict a 3- tuple with each tree node, based on this representation of location. (A node from these Docld is chosen by 1).

Structural relationships between tree nodes both elements or string values whose position are recorded in this method can be determined easily: (i) *ancestor-descendant*: a tree node n2 whose location in the XML database is programmed as (D2; S2 : E2;L2) is a descendant of a tree node n1 whose location is programmed as (D1; S1 : E1;L1) iff D1 = D2; S1 < S2 and E2 < E1;1 (ii) *parent-child*: a tree node n2 whose location in the XML database is programmed as (D2; S2 : E2;L2) is a child of a tree node n1 whose location is programmed as (D1; S1 : E1; L1) iff D1 = D2; S1 < S2;E2 < E1,and L1 +1 = L2.

Figure 2 is indicated by example of the author node position is (1;6 : 8; 3) is a descendant of the book node with position is (1;1 : 70; 1), and the string “jane” with location (1; 7; 4) is a author node child with location (1;6 : 8; 3). A solution point value note that representation of node position in the XML data tree is that inspection an ancestordescendant structural relationship is as simple as inspection a parentchild structural relationship.

The motive is that one can test for an ancestor-descendant structural relationship without knowledge of the intermediate nodes on the path. Also significance noting but this representation of position of elements and string values permit for inspection order and proximity relationships between elements and/or string values, the main issue of these paper is not explored further.

3. CONSTRUCTION

In the rest of this paper, we take advantage of the (DocId, SPos: EPos, LevelNo) representation of positions of XML elements and string values to (i) devise novel, I/O and CPU optimal (in an asymptotic sense) join algorithms for matching basic structural relationships (or, containment queries) against an XML database; (ii) present an analysis of these algorithms; and (iii) show their behavior in practice using a variety of experiments. The task of matching a complex XML clustering query pattern then reduces to that of evaluating a join expression with one join operator for each binary structural relationship in the query pattern. Different join orderings may result in different evaluation costs, as usual. Finding the optimal join ordering is outside the scope of this paper, and is the subject of future work in this area.
1For leaf strings, EPos is the same as SPos.

- Algorithm Tree-Merge-Anc (AList, DList)
- All nodes in AList and DList have the same DocId AList is the list of potential ancestors, in sorted order of SPos
- DList is the list of potential descendants in sorted order of SPos
- Desc-init = DList->FNode; OutputList = NULL;

```
for (a = AList->FNode; a ! = NULL; a = a->NxtNode) {
  for (d = init-desc; (d ! = NULL and d.SPos < a.SPos); d = d->NxtNode) {
        /*  unmatchable d's */}
   Desc-init = d;
   for (d = init-desc; (d ! = NULL and and d.EPos < a.EPos); d = d->NxtNode){
        if ((a.SPos < d.SPos) and (d.EPos < a.EPos)
            [ and (d.Levelno = a.LevelNo + 1)])
}
}
```

- Condition for parent-child relationships

```
        {
        append (a,d) to OutputList;
        }
```

Figure.5. Algorithm Tree-Merge-Anc with output in sorted ancestor/parent order

A. P-C and A-D Algorithms

In this section, we develop two families of join algorithms for matching parent-child and ancestor-descendant structural relationships efficiently: *tree-relation* and *stack-tree*, and present an analysis of these algorithms.

Consider an ancestor-descendant (or, parent-child) structural relationship (e1; e2), for example, (book, author) (or, (author, Jane)) in our running example. Let AList = [a1; a2; : : :] and DList = [d1; d2; : : :] be the lists of tree nodes that match the node predicates e1 and e2, respectively, each list sorted by the (DocId, SPos) values of its elements. There are a number of ways in which the AList and the DList could be generated from the database that stores the XML data. For example, a native XML database system could store each element node in the XML data tree as an object with the attributes: ElementTag, DocId, SPos, EPos, and LevelNo.

An index could be built across all the element tags, which could then be used to find the set of nodes that match a given element tag. The set of nodes could then be sorted by (DocId, SPos) to produce the lists that serve as input to our join algorithms. Given these two input lists, AList of potential ancestors (or parents) and DList of potential descendants (resp, children), the algorithms in each family can output a list OutputList = [(ai; dj)] of join results, sorted either by (DocId, ai.SPos, dj.SPos) or by (DocId, dj.SPos, ai.SPos). Both variants are useful, and the variant chosen may depend on the order in which an optimizer chooses to compose the structural joins to match the complex XML query pattern.

B. Tree Relationship Join Algorithms

The algorithms in the *tree-relation* family are a natural extension of traditional relational merge joins (which use an equality join condition) to deal with the multiple inequality conditions that characterize the ancestor-descendant or the parent-child structural relationships, based on the (DocId, SPos: EPos, LevelNo) representation. The recently proposed multi-predicate merge join (MPMGJN) algorithm [29] is also a member of this family. The basic idea here is to perform a modified merge-join, possibly performing multiple scans through the "inner" join operand to the extent necessary. Either AList or DList can be used as the inner (resp., outer) operand for the join: the results are produced sorted (primarily) by the outer operand. In Figure 5, we present the tree-merge algorithm for the case when the outer join operand is the ancestor; this is similar to the MPMGJN algorithm. Similarly, Figure 6 deals with the case when the outer join operand is the descendant. For ease of understanding, both algorithms assume that all nodes in the two lists have the same value of DocId, their primary sort attribute. Dealing with nodes from multiple documents is straightforward, requiring the comparison of DocId values and the advancement of node pointers as in the traditional merge join.

C. An Analysis of the Tree-Merge Algorithms

Traditional merge joins that use a single equality condition between two attributes as the join predicate can be shown to have time and space complexities O(jinputj + joutputj), on sorted inputs, while producing a sorted output. In general, one cannot establish the same time complexity when the join predicate involves multiple equality and/or inequality conditions. In this section, we identify the criteria under which tree-merge algorithms have asymptotically optimal time complexity.

Algorithm Tree-Merge-Anc for ancestor-descendant Structural Relationship:

Theorem .1 *The space and time complexities of Algorithm* Tree-Merge-Anc *are* O (jAListj+jDListj Output Listj),*for the ancestor-descendant structural relationship.*

The intuition is as follows. Consider first the case where no two nodes in AList are themselves related by an ancestordescendant relationship. In this case, the size of OutputList is O(jAListj + jDListj). Algorithm Tree-Merge-Anc makes a

Algorithm Tree-Merge-Desc (AList, DList)

- Assume that all nodes in AList and DList have the same DocId
- AList is the list of potential ancestors, in sorted order of SPos
- DList is the list of potential descendants in sorted order of SPos

```
init-anc = AList->FNode; OutputList = NULL;
for (des = DList->FNode; des ! = NULL; des = des->NNode)
{
    for (anc = init-anc; (anc ! = NULL and anc.EPos < des.SPos); anc = anc->NxtNode) {
            /* unmatchable a's */}
    init-anc = anc;
    for (anc = init-anc; (anc ! = NULL and anc.SPos < anc.SPos); anc = anc->NxtNode)
{
            if ((anc.SPos < des.SPos) and (des.EPos < anc.EPos)
                    [and (des.LevelNo = anc.LevelNo + 1)]) {
                the condition is for parent-child relationships
                append (anc,des) to OutputList; }
}
}
```

Figure.6. Algorithm Tree-Merge-Desc with output in sorted descendant/child order

Single pass over the input AList and at most two passes over the input DList.2 Thus, the above theorem are satisfied in this case.

Consider next the case where multiple nodes in AList are themselves related by ancestor-descendant relationship. This can happen, for example, in the (section, head) structural relationship for the XML data in Figure 4. In this case, multiple passes may be made over the same set of descendant nodes in DList, and the size of OutputList may be O (jAListj _ jDListj), which is quadratic in the size of the input lists. However, we can show that the algorithm still has optimal time complexity, i.e., O (jAListj + jDListj + jOutputListj). One cannot establish the I/O optimality of Algorithm Tree-Merge-Anc. In fact, repeated paging can cause its I/O behavior to degrade in practice, as we shall see in Section 4.

Algorithm Tree-Merge-Anc for parent-child Structural Relationship:

When evaluating a parentchild structural relationship, the time complexity of Algorithm Tree-Merge-Anc is the same as if one were performing an ancestor-descendant structural relationship match between the same two input lists. However, the size of OutputList for the parent-child structural relationship can be much smaller than the size of the OutputList for the ancestor-descendant structural relationship. In particular, consider the case when all the nodes in AList form a (long) chain of length n, and each node in AList has two children in DList, one on either side of its child in AList; this is shown in Figure 7(a). In this case, it is easy to verify that the size of OutputList is O(jAListj + jDListj), but the time complexity of Algorithm Tree-Merge-Anc is O((jAListj + jDListj)2); the evaluation is pictorially depicted in Figure 6(b), where each node in AList is associated with the sublist of DList that needs to be scanned. The I/O complexity is also quadratic in the input size in this case.

Algorithm Tree-Merge-Desc: There is no analog to Theorem 1 for Algorithm Tree-Merge-Desc, since the time 2A clever implementation that uses a one node look ahead in AList can reduce the number of passes over DList to just one. Complexity of the algorithm can be ((jAListj + jDListj + jOutputListj) 2) in the worst case. This happens, for example, in the case shown in Figure 7(c), when the first node in AList is an ancestor of each node in DList. In this case, each node in DList has only two ancestors in AList, so the size of OutputList is O(jAListj + jDListj), but AList is repeatedly scanned, resulting in a time complexity of O(jAListj_jDListj); the valuation is depicted in Figure 7(d), where each node in DList is associated with the sublist of AList that needs to be scanned. While the worst case behavior of many members of the treemerge family is quite bad, on some data sets and queries they perform quite well in practice. We shall investigate the behavior of Algorithms Tree-Merge-Anc and Tree-Merge-Desc experimentally in Section 4.

D. Stack-Tree Join Algorithms

We observe that a depth-first traversal of a tree can be performed in linear time using a stack of size as large as the height of the tree. In the course of this traversal, every ancestor-descendant relationship in the tree is manifested by the descendant node appearing somewhere higher on the stack than the ancestor node. We use this observation to motivate our search for a family of stack based structural join algorithms, with better worst-case I/O and CPU complexity than the tree-merge family, for both parent-child and ancestor-descendant structural relationships.

Unfortunately, the depth-first traversal idea, even though appealing at first glance, cannot be used directly since it requires traversal of the whole database. We would like to traverse only the candidate nodes provided to us as part of the input lists. We now describe our *stack-tree* family of structural join algorithms; these algorithms have no counterpart in traditional join processing.

E. Stack-Tree-Desc

Consider an ancestor-descendant structural relationship (e 1; e2). Let AList = [a1; a2; : : :] and DList = [d1; d2; : : :] be the lists of tree nodes that match node predicates e 1 and e2, respectively, sorted by the (DocId, SPos) values of its elements.

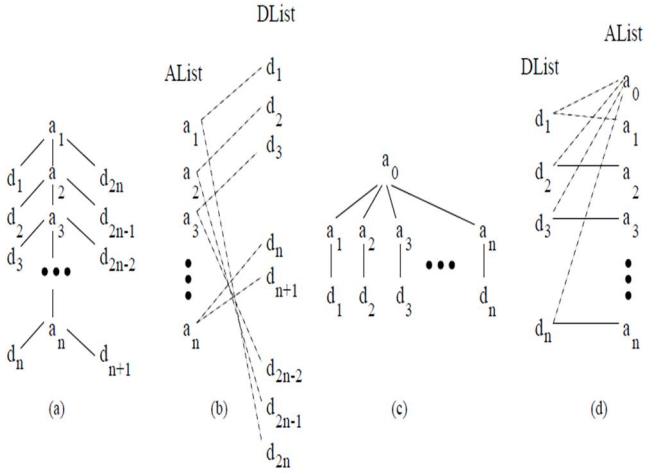

Figure.7. (a), (b) Worst case for Tree-Merge-Anc and (c), (d) Worst case for Tree-Merge-Desc

We first discuss the stack-tree algorithm for the case when the output list [(ai; dj)] is sorted by (DocId, dj.SPos, ai.SPos). This is both simpler to understand and extremely efficient in practice. The algorithm is presented in Figure 5 and 6 for the ancestor-descendant case.

The basic idea is to take the two input operand lists, AList and DList, both sorted on their (DocId, SPos) values and conceptually merge (interleave) them. As the merge proceeds, we determine the ancestor-descendant relationship, if any, between the current top of stack and the next node in the merge, i.e., the node with the smallest value of SPos. Based on this comparison, we manipulate the stack, and produce output.

The stack at all times has a sequence of ancestor nodes, each node in the stack being a descendant of the node below it. When a new node from the AList is found to be a descendant of the current top of stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of stack, we know that it is a descendant of all the nodes in the stack. Also, it is guaranteed that it won't be a descendant of any other node in AList. Hence, the join results involving this DList node with each of the AList nodes in the stack are output. If the new node in the merge list is not a descendant of the current top of stack, then we are guaranteed that no future node in the merge list is a descendant of the current top of stack, so we may pop stack, and repeat our test with the new top of stack. No output is generated when any element in the stack is popped.

The parent-child case of Algorithm Stack-Tree-Desc is even simpler since a DList node can join only (if at all) with the top node on the stack. In this case, the "for loop" inside the "else" case of Figure 8 needs to be replaced with:

if (d.LevelNo = stack->top.LevelNo + 1) append (stack->top,d) to OutputList

Example 3.1 [Algorithm Stack-Tree-Desc]

Some steps during an example evaluation of Algorithm Stack-Tree-Desc, for a parent-child structural relationship, on the dataset of Figure 9(a), are shown in Figures 9(b)–(e). The ai's are the nodes in AList and the dj's are the nodes in DList. Initially, the stack is empty, and the conceptual merge of AList and DList is shown in Figure 9(b). In Figure 9(c), a1 has been put on the stack, and the first new element of the merged list, d1, is compared with the stack top (at this point (a1; d1) is output).

Figure 9(d) illustrates the state of the execution several steps later, when a1; a2; : : : ; an are all on the stack, and dn is being compared with the stack top (after this point, the OutputList includes (a1; d1); (a2; d2); : : : ; (an; dn)). Finally, Figure 9(e) shows the state of the execution when the entire input has almost been processed. Only a 1 remains on the stack (all the other ai's have been popped from the stack), and d2n is compared with a1. Note that all the desired matches have been produced while making only a single pass through the *entire* input. Recall that this is the same dataset of Figure 7(a), which illustrated the sub-optimality of Algorithm Tree-Merge-Anc, for the case of parent-child structural relationships.

F. Stack-Tree-Anc

We next discuss the stack-tree algorithm for the case when the output list [(ai; dj)] needs to be sorted by (DocId, ai.SPos, dj.SPos).

It is not straightforward to modify Algorithm Stack-Tree-Desc to produce results sorted by ancestor because of the following: if node a from AList on the stack is found to be an ancestor of some node d in the DList, then every node a0 from AList that is an ancestor of a (and hence below a on the stack) is also an ancestor of d. Since the SPos of a0 precedes the start position of a, we must delay output of the

join pair (a; d) until after (a0; d) has been output. But there remains the possibility of a new element d0 after d in the DList joining with a0 as long a0 is on stack, so we cannot output the pair (a; d) until the ancestor node a0 is popped from stack. Mean while, we can build up large join results that cannot yet be Algorithm Stack-Tree-Desc (AList, DList)
/* Assume that all nodes in AList and DList have the same DocId */

- AList is the list of potential ancestors, in sorted order of SPos
- DList is the list of potential descendants in sorted order of SPos

```
anc = AList->FNode; des = DList->FNode; OutputList = NULL;
While (the input lists are not empty or the stack is not empty)
{
        if ((anc.SPos > stack->top.EPos) and (des.SPos > stack->top.EPos)) {
                /* time to pop the top element in the stack */
                tuple = stack->pop (); }
         else if (anc.SPos < des.SPos) {
                stack->push(a)
                anc = anc->NxtNode }
        else {
                for (anc1 = stack->bottom; anc1 ! = NULL;
        anc1 = anc1->up) {
                        append (anc1,des) to OutputList
}
                des = des->NNode

        }
}
```

Figure.8. Algorithm Stack-Tree-Desc with output in sorted descendant order output. Our solution to this problem is described in Figure 9 for the ancestor-descendant case.

As with Algorithm Stack-Tree-Desc, the stack at all times has a sequence of ancestor nodes, each node in the stack being a descendant of the node below it. Now, we associate two lists with each node on the stack: the first, called *self-list* is a list of result elements from the join of this node with appropriate DList elements; the second, called *inherit-list* is a list of join results involving AList elements that were descendants of the current node on the stack. As

before, when a new node from the AList is found to be a descendant of the current top of stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of stack, it is simply added to the self-lists of the nodes in the stack.

Again, as before, if no new node (from either list) is a descendant of the current top of stack, then we are guaranteed that no future node in the merge list is a descendant of the current top of stack, so we may pop stack, and repeat our test with the new top of stack. When the bottom element in stack is popped, we output its self-list first and then its inherit-list. When any other element in stack is popped, no output is generated. Instead, we append its inherit-list to its self-list, and append the result to the inherit-list of the new top of stack.

An optimization to the algorithm (incorporated in Figure 9) is as follows: no self-list is maintained for the bottom node in the stack. Instead, join results with the bottom of the stack are output immediately. This results in a small space savings, and renders the stack-tree algorithm partially non-blocking.

G. An Analysis of Algorithm Stack-Tree-Desc

Algorithm Stack-Tree-Desc is easy to analyze. Each AList element in the input may be examined multiple times, but these can be amortized to the element on DList, or the element at the top of stack, against which it is examined. Each element on the stack is popped at most once, and when popped, causes examination of the new top of stack with the current new element. Finally, when a DList element is compared against the top element in stack, then it either joins with all elements on stack or none of them; all join results are immediately output. In other words, the time required for this part is directly proportional to the output size. Thus, the time required for this algorithm is O(jinputj + joutputj) in the worst case. Putting all this together, we get the following result:

Theorem.2 *The space and time complexities of Algorithm* Stack-Tree-Desc*are*O (jAListj+ jDListj + j outputListj)*, for both ancestor-descendant and parent-child structural relationships.*

Further, Algorithm* Stack-Tree-Desc *is a non-blocking algorithm.

Clearly, no competing join algorithm that has the same input lists, and is required to compute the same output list, could have better asymptotic complexity. The I/O complexity analysis is straightforward as well. Each page of the input lists is read once, and the result is output as soon as it is computed. Since the maximum size of stack is proportional to the height of the XML database tree, it is quite reasonable to assume that all of stack fits in memory at all time. Hence, we have the following result:

Theorem 3 *The I/O complexity of Algorithm* Stack-Tree-Desc *is* O(jAListj B + jDListj B + jOutputListj B)*, for ancestor-descendant and parent-child structural relationships, where* B *is the blocking factor.*

H. An Analysis of Algorithm Stack-Tree-Anc

The key difference between the analyses of Algorithms Stack-Tree-Anc and Stack-Tree-Desc is that join results are associated with nodes in the stack in Algorithm Stack-Tree-Anc. Obviously, the list of join results at any node in the stack is linear in the output size. What remains to be analyzed is the appending of lists each time the stack is popped.

If the lists are implemented as linked lists (with start and end pointers), these append operations can be carried out in unit time, and require no copying. Thus one comparison per AList input and one per output are all that are performed to manipulate stack. Combined with the analysis of Algorithm Stack-Tree-Desc, we can see that the time required for this algorithm is still O(jinputj + joutputj) in the worst case. The I/O complexity analysis is a little more involved. Certainly, one cannot assume that all the lists of results not yet output fit in memory. Careful buffer management is required. It turns out that the only operation we ever perform on a list is to append to it (except for the final read out).

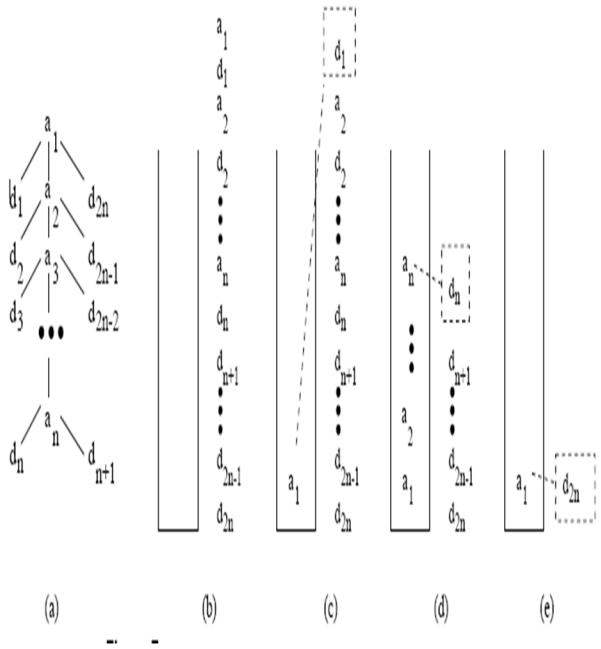

Figure.9. (a) Dataset (b)–(e) Steps during evaluation of Stack-Tree-Desc

As such, we only need to have access to the tail of each list in memory as computation proceeds. The rest of the list can be paged out. When list x is appended to list y, it is not necessary that the head of list x be in memory, the append operation only establishes a link to this head in the tail of y. So all we need is to know the pointer for the head of each list, even if it is paged out. Each list page is thus paged out at most once, and paged back in again only when the list is ready for output. Since the total number of entries in the lists is exactly equal to the number of entries in the output, we thus have that the I/O required on account of maintaining lists of results is proportional to the size of output (provided that there is enough memory to hold in buffer the tail of each list: requiring two pages of memory per stack entry — still a requirement within reason). All other I/O activity is for the input and output. This leads to the desired linearity result.

Theorem4 *The space and time complexities of*Algorithm Stack-Tree-Anc are O (jAListj+jDListj +jOutputListj), *for both ancestor-descendant and parent-child structural relationships.*

The I/O complexity of Algorithm Stack-Tree-Anc *is* O(jAListj B + jDListj B + jOutputListj B), *for both ancestor-descendant and parent-child structural relationships, where is the blocking Factor.*

4. EXPERIMENTAL EVALUATIONS

In this section, we present the results of an actual implementation of the various join algorithms for XML data sets. Due to space limitations, we evaluate only the structural join algorithms we introduce in this paper, namely, TREE-MERGE JOIN(TMJ) and STACK-TREE JOIN (STJ). Once more, the output can be sorted in two ways, based on the "ancestor" node or the descendant" node in the join. Correspondingly, we consider two flavors of these algorithms, and use the suffix "-A" and "-D" to differentiate between these. The four algorithms are thus labeled: TMJ-A, TMJ-D, STJA and STJ-D.

For reasons of space, we omit detailed comparison of our structural join algorithms with traversal-style algorithms, and with traditional relational join algorithms in a commercial database. As expected, the performance of the traversal-style algorithms degrades considerably with the size of the dataset, and yields very poor performance compared with our structural join algorithms. Also, consistent with the results of [29], structural join algorithms (implemented outside the database) perform significantly better than native relational DBMS join algorithms, even in the presence of indexes.

We implemented the join algorithms in the TIMBER XML query engine. TIMBER is an native XML query engine that is built on top of SHORE [5]. Since the goal of TIMBER is to efficiently handle complex XML queries on large data sets, we implemented our algorithms so that they could participate in complex query evaluation plans with pipelining. All experiments using TIMBER were run on a 500MHz Intel Pentium III processor running WindowsNT Workstation v4.0. SHORE was compiled for a 8KB page size. SHORE buffer pool size was set to 32MB, and the container size in our implementation was 8000 bytes.

All numbers presented here are produced by running the experiments multiple times and averaging all the execution times except for the first run (i.e., these are warm cache numbers).

Algorithm Stack-Tree-Anc (AList, DList)

- Assume that all nodes in AList and DList have the same DocId
- AList is the list of potential ancestors, in sorted order of SPos
- DList is the list of potential descendants in sorted order of SPos

```
anc = AList->FNode; des = DList->FNode; OutputList =
NULL;
while (the input lists are not empty or the stack is not empty)
{
if ((anc.SPos > stack->top.EPos) andand (des.SPos > stack-
>top.EPos)) {
/* time to pop the top element in the stack */
        tuple = stack->pop();
        if (stack->size == 0) {
                append tuple.inherit-list to OutputList }
        else {
                append tuple.inherit-list to tuple.self-list
                append the resulting tuple.self-list to stack-
        >top.inherit-list
}
}
        else if (anc.SPos < des.SPos) {
                stack->push(anc)
                anc = anc->NxtNode }
        else {
                for (anc1 = stack->bottom; anc1 ! = NULL;
        anc1 = anc1->up)
{
                    if (anc1 == stack->bottom) append
        (anc1,des) to OutputList
                     else append (anc1,des) to the self-list of
        anc1
        }
       des = des->NxtNode
    }
}
```

Figure.10. Algorithm Stack-Tree-Anc with output in sorted ancestor order

5. WORKLOAD

For our workload, we used the IBM XML data generator to generate a number of data sets, of varying sizes and other data characteristics, such as the fan out (MaxRepeats) and the maximum depth, using the Organization DTD presented in Figure 11. We also used the XMach-1 [1] and XMark [2] benchmarks, and some real XML data. The results obtained were very similar in all cases, and in the interest of space we present results only for the largest organization data set that we generated. This data set consists of 6.3 million element nodes, corresponding to proximately 800MB of XML documents in text format. The characteristics of this data set in terms of the number of occurrences of element tags are summarized in Table 1.

We evaluated the various join algorithms using the set of queries shown in Table 1. The queries are broken up into two classes. QS1 to QS6 are simple structural relationship queries, and have an equal mix of parent-child queries and ancestor descendant queries. QC1 and QC2 are complex chain queri*es*, and are used to demonstrate the performance of the algorithms when evaluating complex queries with multiple joins in a pipeline.

A. Detailed Implementation:

The focus in the experiments is to characterize the performance of the four structural join algorithms, and understand their differences. Before doing so in the following subsections, we present here some additional detail regarding the manner in which these were implemented for the experiments reported. Our choice of implementation, on top of SHORE and TIMBER, was driven purely by the need for sufficient control the algorithms themselves could just as well have been implemented on many other platforms, including (as new join methods) on relational databases.

All join algorithms were implemented using the operator iterator model [15]. In this model, each operator provides an *open*, *next* and *close* interface to other operators, and allows the database engine to construct an operator tree with an arbitrary mix of query operations (different join algorithms or algorithms for other operations such as aggregation) and naturally allows for a pipelined operator evaluation. To support this iterated model, we pay careful attention to the manner in which results are passed from one operator to another.

Algorithms such as the TMJ algorithms may need to repeatedly scan over one of the inputs. Such repeated scans are feasible if the input to a TMJ operator is a stream from a disk file, but is not feasible if the input stream originates from another join operator (in the pipeline below it). We implemented the TMJ algorithms so that the nodes in a current sweep are stored in a temporary SHORE file. On the next sweep, this temporary SHORE file is scanned. This solution allows us to limit the memory used by TMJ implementation, as the only memory used is managed by the SHORE buffer manager, which takes care of evicting pages of the temporary file from the buffer pool if required. Similarly for the STJ-A algorithm, the inherit and self-lists

are stored in a temporary SHORE file, again limiting the memory used by the algorithm.

In both cases, our implementation turns logging and locking off for the temporary SHORE files. Note that STJ-D can join the two inputs in a single pass over both inputs and never has to spool any nodes to a temporary file.

<!ELEMENT manager (name,(manager|department|employee)+)>
<! ATTLIST manager id CDATA #FIXED "1">
<!ELEMENT department (name, email?, employee+, department*)>
<!ATTLIST department id CDATA #FIXED "2">
<!ELEMENT employee (name+,email?)>
<!ATTLIST employee id CDATA #FIXED "3">
<!ELEMENT name (#PCDATA)>
<!ATTLIST name id CDATA #FIXED "4">
<!ELEMENT email (#PCDATA)>
<!ATTLIST email id CDATA #FIXED "5">

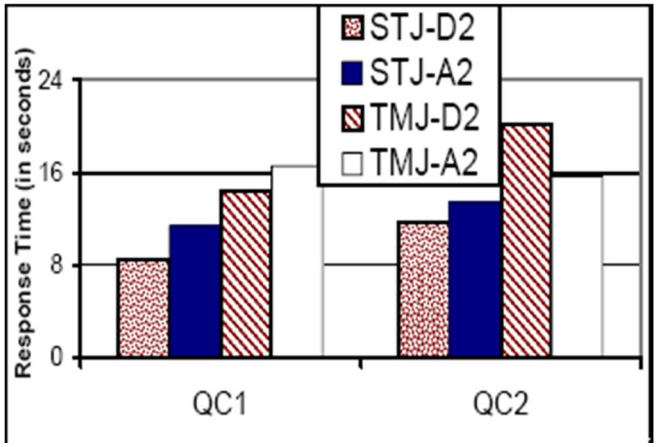

Figure.11. DTD used in our experiments

Table 2
Description of queries and characteristics of the data set

Node	Count
Manager	25,880
Department	342,450
Employee	574,530
Email	250,530

Query	XQuery Path Expression	Result
QS1	employee/email	140,700
QS2	employee//email	1 42,958
QS3	manager/department	6,855
QS4	manager/department	57,137
QS5	manager/employee	7259
QS6	manager/employee	0,774
QC1	manager/employee/email	7990
QC2	manager/employee/email	232,406

To amortize the storage and access overhead associated with each SHORE object, in our implementation we group nodes into a large *container* object, and create a SHORE object for each container. The join algorithms write nodes to containers and when a container is full it is written to the temporary SHORE file as a SHORE record. The performance benefits of this approach are substantial; we do not go into details for lack of space.

B. STJ and TMJ, Simple Structural Join Queries

Here, we compare the performance of the STJ and the TMJ algorithms using all the six simple queries, QS1–QS6, shown in Table 1. Figure 12 plots the performance of the four algorithms. As shown in the Figure, STJ-D outperforms the remaining algorithms in all cases.

The reason for the superior performance of STJ-D is because of its ability to join the two data sets in a single pass over the input nodes, and it never has to write any nodes to intermediate files on disk. From Figure 12(a), we can also see that STJ-A usually has better performance than both TMJ-A and TMJ-D. For queries QS4 and QS6, the STJ-A algorithms and the two TMJ algorithms have comparable performance.

These queries have large result sizes (approximately 600K and 1M tuples respectively as shown in Table 1). Since STJ-A keeps the results in the lists associated with the stack, and can output the results only when the bottommost element of the stack is popped, it has to perform many writes and transfers of the lists associated with the stack elements (in our implementation, these lists are maintained in temporary SHORE files).

Furthermore, the effect of buffer pool size is likely to be critical when one of the inputs has nodes that are deeply nested amongst themselves, *and* the node that is higher up in the XML tree has many nodes that it joins with. For example, consider the TMJ-A algorithms, and the query "manager/employee". If many manager nodes are nested below a manager node that is higher up in the XML tree, then after the join of the manager node at the top is done, repeated scans of the descendant nodes will be required for the manager nodes that are descendants of the manager node at the top. Such scenarios are rare in our data set, and, consequently, the buffer pool size has only a marginal impact on the performance of the algorithms.

C. Complex Queries

Here, we evaluate the performance of the algorithms using the two complex chain queries, QC1 and QC2, from Table 1. Each query has two joins and for this experiment,

both join operations are evaluated in a pipeline. For each complex query one can evaluate the query by using only ancestor-based join algorithms or using only descendant-based join algorithms. These two approaches are labeled with suffixes "-A2" and "-D2" for the ancestor-based and descendant-based approaches respectively.

The performance comparison of the STJ and TMJ algorithms for both query evaluation approaches (A2 and D2) is shown in Figure 11. From the figure we see that STJ-D2 has the highest performance once again, since it is never has to spool nodes to intermediate files.

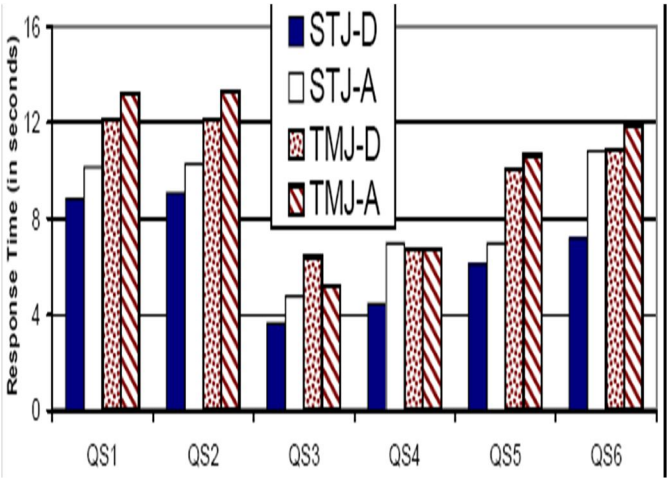

Figure.12(a). STJ and TMJ, simple queries: QS1, QS6

Matchings between pairs of trees in memory has been a topic of study in the algorithm community for a long time (e.g., see [22] and references therein).

The algorithms developed deal with many variations of the problem but unfortunately are of high complexity and always assume that trees are entirely memory resident. The problem also has been considered in the programming language community, as it arises in various type checking scenarios but once again solutions developed are geared towards small data collections processed entirely in main memory.

Figure.12(b). STJ and TMJ, complex queries: QC1, QC2

Jacobson et al. [16] present linear time merging-style algorithms for computing the elements of a list that are descendants/ancestors of some elements in a second list, in the context of focusing keyword-based searches on the Web and in UNIX-style file systems. Jagadish et al, [17] present linear time stack-based algorithms for computing elements of a list that satisfy a hierarchical aggregate selection condition wrt elements in a second list, for the directory data model. However, none of these algorithms compute *joins results*, which is the focus of our work.

Join processing is central to database implementation and there is a vast amount of work in this area [15]. For inequality join conditions, band join [11] algorithms are applicable when there exists a fixed arithmetic difference between the values of join attributes. Such algorithms are not applicable in our domain as there is no notion of fixed arithmetic difference. In the context of spatial and multimedia databases, the problem of computing joins between pairs of spatial entities has been considered, where commonly the predicate of interest is overlap between spatial entities [18, 24, 19] in multiple dimensions.

The techniques developed in this paper are related to such join operations. However, the predicates considered as well as the techniques we develop are special to the nature of our structural join problem. In the context of semistructured and XML databases, the issue of query evaluation and optimization has attracted a lot of research attention. In particular, work done in the context of the Lore database management system [20, 21], and the Niagara system [23], has considered various aspects of query processing on such data. XML data and various issues in their storage as well as query processing using relational database systems have recently been considered in [14, 27, 26, 4, 13]. In [14, 27, 13], the mapping of XML data to a number of relations was considered along with translation of a select subset of XML queries to relational queries. In subsequent work [26, 4, 12], the authors considered the problem of publishing XML documents from relational databases.

Our work is complementary to all of these since our focus is on the join algorithms for the primitive (ancestor-descendant and parentchild) structural relationships. Our join algorithms can be used by these previous works to advantage. The representation of positions of XML elements used by us, (DocId, StPos : EPos, LevelNo), is essentially that of Consens and Milo, who considered a fragment of the PAT text searching operators for indexing text databases [9], This representation was used to compute containment relationships between "text regions" in the text databases. The focus of that work was solely on theoretical issues, without elaborating on efficient algorithms for computing these relationships.

Finally, the recent work of Zhang et al. [29] is closely related to ours. They proposed the multi predicate merge join (MPMGJN) algorithm for evaluating containment queries, using the (DocId, SPos: EPos, LevelNo) representation. The MPMGJN algorithm is a member of our Tree-Merge family. Our analytical and experimental results demonstrate that the Stack-Tree family is considerably superior to the Tree-Merge family for evaluating containment queries.

6. CONCLUSIONS

Now, We have identified a large optimal query classes namely that is Z\,\\,α; Z\,\\,α,β and Z\,\\,α,β,γ respectively and We have introduced a notion of matching cross to address the problem of the suboptimality in holistic XML clustering tree pattern matching algorithms. Based on these results, we have planned a new holistic algorithm called TreeMatch to achieve such abstract optimal query classes. Finally, general experiments demonstrate the advantage of our algorithms and verify the accuracy of abstract results.

REFERENCES

1. Boukottaya, C. Vanoirbeek, Schema matching for transforming structured documents,The 2005 ACM Symposium on Document engineering, Bristol, United Kingdom.
2. G.Koloniari, E. Pitoura, Peer-to-peer management of XML data, issues and research challenges, SIGMOD Record 34 (2) (2005) 6–17.
3. R. Nayak, R. Witt, A. Tonev, Data mining and XML documents, The 2002 International Workshop on the Web and Database (WebDB 2002), June 24–27, 2002.
4. R. Nayak, M. Zaki (Eds.), Knowledge discovery from XML documents, PAKDD 2006 workshop proceedings,Lecture Notes in Computer Science, vol.
5. D.D. Chamberlin, J. Robie, and D. Florescu. Quilt:An XML query language for heterogeneous data sources. In *Proceedings of WebDB*, 2000.
6. N. Jardine, C.J. van Rijsbergen, The use of hierarchical clustering in information retrieval, Information storage and retrieval 7(1971) 217–240.
7. M.P. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of PODS*, 1995.
8. M. Hearst, J.O. Pedersen, Reexamining the cluster hypothesis: Scatter/gather on retrieval results, Proceedings of the ACM SIGIR Conference, Zurich, Switzerland, 1996, pp. 76–84.
9. E.Rasmussen, Clustering algorithms, W. Frakes, R. Baeza-Yates (Eds.), Information Retrieval: Data Structures and Algorithms, Prentice Hall, 1992.
10. M. Halkidi, Y. Batistakis, M. Vazirgiannis, Clustering algorithms and validity measures, SSDBM Conference, Virginia, USA, 2001.
11. T. Fiebig and G. Moerkotte. Evaluating queries on structure with access support relations. *Proceedings of WebDB*, 2000.
12. D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
13. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.
14. G. Jacobson, B. Krishnamurthy ,D. Srivastava, and D. Suciu. Focusing search in hierarchical structures with directory sets. In *Proceedings of CIKM*, 1998.

[1]N.Kannaiya Raja received degree MCA from Alagappa University and ME from Anna University Chennai in 2007

joined assistant professor various engineering colleges in Tamil Nadu affiliated to Anna University and has eight years teaching experience and his research works in deep packet inspection. He has been session chair in major conference and workshops in computer vision on algorithm papers, network, mobile communication, image processing papers and pattern reorganization. His current primary areas of research are packet inspection and network. He is interested to conduct guest lecturer in various engineering in Tamil Nadu.

[2]Dr.K.Arulanandam received PhD doctorate degree in 2010 from Vinayaka mission university Salem. He has twelve year teaching experience in various engineering colleges in Tamil Nadu which are affiliated to Anna University and his research experience network, mobile communication networks, image processing papers and algorithm papers

[3]P.Umadevi received degree BE from Madras in 2001 and M.E from Anna University Chennai in 2008 joined assistant

professor in various engineering colleges in Tamil Nadu affiliated to Anna University and has six years teaching experience and her research works in Cryptography. His current primary areas of research are packet inspection and network. She is interested to conduct conference and guest lecturer in various engineering in Tamil Nadu.

[4] A.Balakrishnan received degree B.E Computer Science and Engineering from Anna University Chennai. Now pursuing

ME Computer Science and Engineering in Arulmigu Meenakshi Amman College of Engineering affiliated to Anna University Chennai.