Volume 2, Issue 4, April 2012 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Processing and Evaluation of Broad Fragments of XPath

B.Kiran Kumar
Department of CSE
Sree Vidyanikethan Engineering college
bkiran1351@gmail.com

Mr. Shaik Salam [*]
Department of CSE
Sree Vidyanikethan Engineering College
Shaiksalam17@yahoo.com

Abstract*— XML queries typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. The primitive tree structured relationships are parent-child and ancestor-descendant, and finding all occurrences of these relationships in an XML database is a core operation for XML query processing. In this paper, we develop two families of structural join algorithms for this task: tree-merge and stack-tree. The tree-merge algorithms are a natural extension of traditional merge joins and the recently proposed multi-predicate merge joins, while the stack-tree algorithms have no counterpart in traditional relational join processing. We present experimental results on a range of data and queries using the TIMBER native XML query engine built on top of SHORE. We show that while, in some cases, tree-merge algorithms can have performance comparable to stack-tree algorithms, in many cases they are considerably worse. This behavior is explained by analytical results that demonstrate that, on sorted inputs, the stack-tree algorithms have worst-case I/O and CPU complexities linear in the sum of the sizes of inputs and output, while the tree-merge algorithms do not have the same guarantee

***Keywords*— XML query processing, XPath query evaluation, tree-pattern query, partial tree-pattern query.**

I. INTRODUCTION

XML employs a tree-structured model for representing data. Quite naturally, queries in XML query languages typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. For example, the XQuery path expression: book[title 'XML']//author[. = 'jane'] matches author elements that (i) have as content the string value "jane", and (ii) are descendants of book elements that have a child title element whose content is the string value "XML". This XQuery path expression can be represented as a node-labeled tree pattern with elements and string values as node labels. Such a complex query tree pattern can be naturally decomposed into a set of basic parent-child and ancestor-descendant relationships between pairs of nodes. For example, the basic structural relationships corresponding to the above query are the ancestor descendant relationship (book, author) and the parent-child relationships (book, title), (title, XML) and (author, jane). The query pattern can then be matched by (i) matching each of the binary structural relationships against the XML database, and (ii) "stitching" together these basic matches. Finding all occurrences of these basic structural relationships in an XML database is clearly a core operation in XML query processing, both in relational implementations of XML databases, and in native XML databases. There has been a great deal of work done on how to find occurrences of such structural relationships (as well as the query tree patterns in which they are embedded) using relational database systems, as well as using native XML query engines. These works typically use some combination of indexes on elements and string values, tree traversal algorithms, and join algorithms on the edge relationships between nodes in the XML data tree. More recently, Zhang et al. proposed a variation of the traditional merge join algorithm, called the multi-predicate merge join (MPMGJN) algorithm, for finding all occurrences of the basic structural relationships (they refer to them as containment queries). They compared the implementation of containment queries using native support in two commercial database systems, and a special purpose inverted list engine based on the MPMGJN algorithm. Their results showed that the MPMGJN algorithm could outperform standard RDBMS join algorithms by more than an order of magnitude on containment queries. The key to the efficiency of the MPMGJN algorithm is the (DocId, StartPos : EndPos, Leve lNum) representation of positions of XML elements, and the (DocId, StartPos, LevelNum) representation of positions of string values, that succinctly capture the *structural relationships* between elements (and string values) in the XML database. Checking that structural relationships in the XML tree, like ancestor-descendant and parent-child (corresponding to containment and direct containment relationships, respectively.

1.1 Outline and Contributions

We begin by presenting background material in Section 2. Our main contributions are as follows: _ We develop two families of join algorithms to perform matching of the parent-child and ancestor-descendant structural relationships efficiently: *tree-merge* and *stack-tree* Given two input lists of tree nodes, each sorted by (DocId, StartPos), the algorithms compute an output list of sorted results joined according to the desired structural relationship. The tree-merge algorithms are a natural extension of merge joins and the recently proposed MPMGJN algorithm, while the stack-tree algorithms have no counterpart in traditional relational join processing. _ We present an analysis of the tree-merge and the stack-tree algorithms. The stack-tree algorithms are I/O and CPU optimal (in an asymptotic sense), and have worst case I/O and CPU complexities linear in the sum of sizes of the two input lists and the output list for both ancestor descendant (or, containment) and parent-child (or, direct containment) structural relationships. The tree-merge algorithms have worst-case quadratic I/O and CPU complexities, but on some natural classes of structural relationships and XML data, they have linear complexity as well.

II. Data Model and Query Patterns

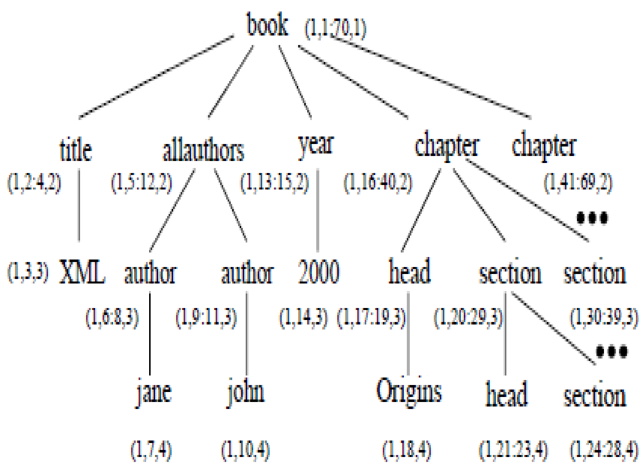
An XML database is a forest of rooted, ordered, labeled trees, each node corresponding to an element and the edges representing (direct) element-sub element relationships. Node labels consist of a set of (attribute, value) pairs, which suffices to model tags, PCDATA content, etc. its tree representation. Queries in XML query languages like XQuery , Quilt and XML-QL make fundamental use of (node labeled) tree patterns for matching relevant portions of data in the XML database. The query pattern node labels include element tags, attribute-value comparisons, and string values, and the query pattern edges are either parent-child edges (depicted using single line) or ancestor-descendant edges (depicted using a double line). For example, the XQuery path expression in the introduction can be represented as the rooted tree pattern . This query pattern would match the document In general, at each node in the query tree pattern, there is a *node predicate* that specifies some predicate on the attributes of the node in question. For the purposes of this paper, exactly what is permitted in this predicate is not material. It suffices for our purposes that there be the possibility of constructing efficient access mechanisms to identify the nodes in the XML database that satisfy any given node predicate.

2.1 Matching Basic Structural Relationships

A complex query tree pattern can be decomposed into a set of basic binary structural relationships such as parent-child and ancestor-descendant between pairs of nodes. The query pattern can then be matched by (i) matching each of the binary structural relationships against the XML database, and (ii) "stitching" together these basic matches. For example, the basic structural relationships corresponding to the query tree pattern . A straightforward approach to matching structural relationships 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 be inefficient compared to the set-at-a-time strategies used in database systems. Pointer-based joins have been suggested as a solution to this problem in object-oriented databases, and shown to be quite efficient.

```
<book>
   <title> XML </title>
   <allauthors>
      <author> jane </author>
      <author> john </author>
   </allauthors>
   <year> 2000 </year>
   <chapter>
      <head> Origins </head>
      <section>
         <head> ...</head>
         <section> ...</section>
      </section>
      <section> ...</section>
   </chapter>
   <chapter> ...</chapter>
</book>
```

(a)



(b)

Figure 1. (a) A sample XML document fragment, (b) Tree representation

2.2 Representing Positions of Elements and String Values in an XML Database

The key to an efficient, uniform mechanism for set-at-a-time (join-based) matching of structural relationships is a positional representation of occurrences of XML elements and string values in the XML database, which extends the classic inverted index data structure in information retrieval. The position of an element occurrence in the XML database can be represented as the 3-tuple (DocId, StartPos : EndPos, LevelNum), and the position of a string occurrence in the XML database can be represented as the 3-tuple (DocId, StartPos, LevelNum), where (i) DocId is the identifier of the document; (ii) StartPos and EndPos can be generated by counting word numbers from the beginning of the document with identifier DocId until the start of the element and end of the element, respectively; and (iii) LevelNum is the nesting depth of the element (or string value) in the document. depicts a 3- tuple with each tree node, based on this representation of position. (The DocId for each of these nodes is chosen to be 1.) Structural relationships between tree nodes (elements or string values) whose positions are recorded in this fashion can be determined easily: (i) *ancestor-descendant*: a tree node n2 whose position in the XML database is encoded as (D2; S2 : E2;L2) is a descendant of a tree node n1 whose position is encoded as (D1; S1 : E1;L1) iff D1 = D2; S1 < S2 and E2 < E1;1 (ii) *parent-child*: a tree node n2 whose position in the XML database is encoded as (D2; S2 : E2;L2) is a child of a tree node n1 whose position is encoded as (D1; S1 : E1; L1) iff D1 = D2; S1 < S2;E2 < E1,and L1 +1 = L2. For example, in Figure 1(b), the author node with position (1;6 : 8; 3) is a descendant of the book node with position (1;1 : 70; 1), and the string "jane" with position (1; 7; 4) is a child of the author node with position (1;6 : 8; 3). A key point worth noting about this representation of node positions in the XML data tree is that checking an ancestor descendant structural relationship is as easy as checking a parent child structural relationship. The reason is that one can check for an ancestor-descendant structural relationship without knowledge of the intermediate nodes on the path. Also worth noting is that this representation of positions of elements and string values allow for checking order and proximity relationships between elements and/or string values; this issue is not explored further in our paper.

III. Structural Join Algorithms

In this section, we develop two families of join algorithms for matching parent-child and ancestor-descendant structural relationships efficiently: *tree-merge* 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, StartPos) 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, StartPos, EndPos, and LevelNum. 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, StartPos) 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.StartPos, dj.StartPos) or by (DocId, dj.StartPos, ai.StartPos). 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.

3.1 Tree-Merge Join Algorithms

The algorithms in the *tree-merge* 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, StartPos : EndPos, LevelNum) 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. 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,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.

3.2 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.

IV. CONCLUSION

In this paper, our focus has been the development of new join algorithms for dealing with a core operation central to much of XML query processing, both for native XML query processor implementations as well for relational XML query processors. In particular, the Stack-Tree family of structural join algorithms was shown to be both I/O and CPU optimal, and practically efficient. There is a great deal more to efficient XML query processing than is within the scope of this paper. For example, XML permits links across documents, and such "pointer-based joins" are frequently useful in querying. We do not consider such joins in this paper, since we believe that they can be adequately addressed using traditional value-based join methods. There are many issues yet to be explored, and we currently have initiated efforts to address some of these, including the piecing together of structural joins and value-based joins to build effective query plans.

V. FUTURE WORK

PartialTreeStack outperforms a state of- the art XQuery engine on PTPQs.Indexing techniques were shown to speed up substantially holistic stack-based algorithms on TPQs. An interesting research direction involves extending the algorithms presented in this paper for PTPQs so that they can take advantage of these optimization techniques. Using materialized views to optimize our PTPQ evaluation algorithm is another useful extension of the present work.

REFERENCES

[1] M. P. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of PODS*, 1995.
[2] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to theWorldWideWeb Consortium 19-August-1998.Available from http://www.w3.org/TR/NOTE-xml-ql., 1998.
[3] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non equijoin algorithms. *Proceedings of SIGMOD*, 1991.
[4] M. Fernandez and D. Suciu. SilkRoute: Trading between relations and XML. *WWW9*, 2000.
[5] T. Fiebig and G. Moerkotte. Evaluating queries on structure with access support relations. *Proceedings of WebDB*, 2000.
[6] D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
[7] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.
[8] G. Jacobson, B. Krishnamurthy, D. Srivastava, and D. Suciu. Focusing search in hierarchical structures with directory sets. In *Proceedings of CIKM*, 1998.
[9] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of SIGMOD*, 1999. *Proceedings of SIGMOD*, 1997.
[10] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. *Proceedings of SIGMOD*, 1996.
[11] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.
[12] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, 1999.
[13] U. of Washington. The Tukwila system. Available from http://data.cs.washington.edu/integration/tukwila/.
[14] U. of Wisconsin. The Niagara system. Available from http://www.cs.wisc.edu/niagara/.
[15] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. *Proceedings of SIGMOD*, 1996.
[16] G. Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, New York, 1983.
[17] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of VLDB*, 2000.
[18] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. De- Witt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of VLDB*, 1999.
[19] E. Shekita and M. Carey. A performance evaluation of pointer based joins. *Proceedings of SIGMOD*, 1990.
[20] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.