Structural Joins: A Primitive for Efficient XML Query Pattern Matching

Shurug Al-Khalifa
Univ of Michigan
shurug@eecs.umich.edu

H. V. Jagadish
Univ of Michigan
jag@eecs.umich.edu

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

Jignesh M. Patel
Univ of Michigan
jignesh@eecs.umich.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Yuqing Wu
Univ of Michigan
yuwu@eecs.umich.edu

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.*

1 Introduction

XML employs a tree-structured model for representing data. Quite naturally, queries in XML query languages (see, e.g., [10, 7, 6]) 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 (see, e.g., [14, 27, 26]), as well as using native XML query engines (see, e.g., [21, 23, 22]). 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. [29] 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, LevelNum)` 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 (see Section 2.3 for details about this representation). Checking that structural relationships in the XML tree, like ancestor-descendant and parent-child (corresponding to containment and direct containment relationships, respectively, in the XML document representation), are present between elements amounts to checking that certain *inequality conditions* hold between the components of the positions of these elements.

While the MPMGJN algorithm outperforms standard RDBMS join algorithms, we show in this paper that it can perform a lot of *unnecessary* computation and I/O for matching basic structural relationships, especially in the case of parent-child relationships (or, direct containment queries). *In this paper, we take advantage of the* `(DocId, StartPos : EndPos, LevelNum)` *representation of positions of XML elements and string values to devise novel I/O and CPU optimal join algorithms for matching structural relationships against an XML database*.

Since a great deal of XML data is expected to be stored in relational database systems (all the major DBMS vendors including Oracle, IBM and Microsoft are providing system support for XML data), our study provides evidence that RDBMS systems need to augment their suite of physical join algorithms to include structural joins to be competitive on XML query processing. Our study is equally relevant for native XML query engines, since structural joins provide for an efficient set-at-a-time strategy for matching XML query patterns, in contrast to the node-at-a-time approach of using tree traversals.

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* (Section 3). 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 [29], 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 (Section 3). 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.
- We show experimental results on a range of data and queries using the TIMBER native XML query engine built on top of SHORE (Section 4). We show that while, in some cases, the performance of tree-merge algorithms can be comparable to that of stack-tree algorithms, in many cases they are considerably worse. This is consistent with the analysis presented in Section 3.

We describe related work in Section 5, and discuss ongoing and future work in Section 6.

2 Background and Overview

2.1 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-subelement relationships. Node labels consist of a set of (attribute, value) pairs, which suffices to model tags, `PCDATA` content, etc. For the sample XML document of Figure 1(a), its tree representation is shown in Figure 1(b). (The utility of the numbers associated with the tree nodes will be explained in Section 2.3.)

Queries in XML query languages like XQuery [6], Quilt [7], and XML-QL [10] 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 in Figure 2(a). This query pattern would match the document in Figure 1.

In general, at each node in the query tree pattern, there is a *node predicate* that specifies some predicate on the attributes (e.g., tag, content) 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 (such as index structures) to identify the nodes in the XML database that satisfy any given node predicate.

2.2 Matching Basic Structural Relationships

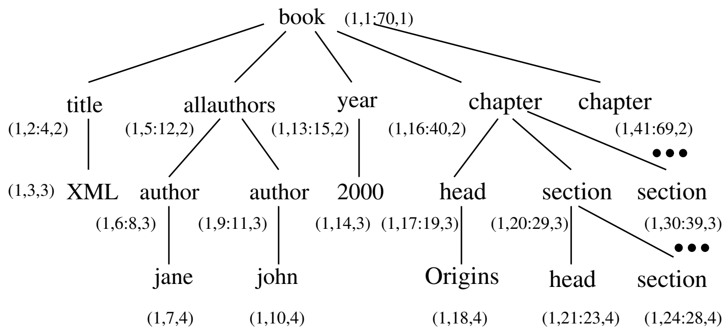
A complex query tree pattern can be decomposed into a set of basic binary structural relationships such as parent-child and ancestor-descendant between pairs of nodes. The query pattern can then be matched by (i) matching each of the binary structural relationships against the XML database, and (ii) "stitching" together these basic matches. For example, the basic structural relationships corresponding to the query tree pattern of Figure 2(a) are shown in Figure 2(b).

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 [28] have been suggested as a solution to this problem in object-oriented databases, and shown to be quite efficient.

In the context of XML databases, nodes may have a large number of children, and the query pattern often requires matching ancestor-descendant structural relationships (for example, the `(book, author)` edge in the query pattern of Figure 2(a)), 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. The former approach would take too much query processing time, while the latter approach would use too

```
<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

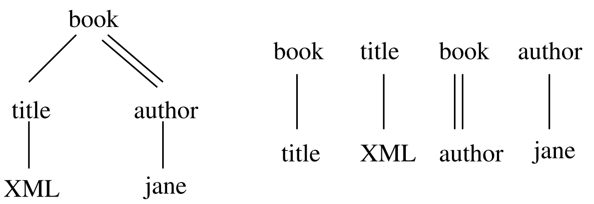


Figure 2. (a) Tree pattern, (b) Structural relationships

much (quadratic) space. In either case, using pointer-based joins is likely to be infeasible.

2.3 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 (see, e.g., [8, 9, 29]), which extends the classic inverted index data structure in information retrieval [25].

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. Figure 1(b) 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 n_2 whose position in the XML database is encoded as $(D_2, S_2 : E_2, L_2)$ is a descendant of a tree node n_1 whose position is encoded as $(D_1, S_1 : E_1, L_1)$ iff $D_1 = D_2, S_1 < S_2$ and $E_2 < E_1$;[1] (ii) *parent-child*: a tree node n_2 whose position in the XML database is encoded as $(D_2, S_2 : E_2, L_2)$ is a child of a tree node n_1 whose position is encoded as $(D_1, S_1 : E_1, L_1)$ iff $D_1 = D_2, S_1 < S_2, E_2 < E_1$,and $L_1 + 1 = L_2$.

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.

2.4 An Overview

In the rest of this paper, we take advantage of the `(DocId, StartPos : EndPos, LevelNum)` 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 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.

[1] For leaf strings, `EndPos` is the same as `StartPos`.

```
Algorithm Tree-Merge-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 StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

begin-desc = DList->firstNode; OutputList = NULL;
for (a = AList->firstNode; a != NULL; a = a->nextNode) {
   for (d = begin-desc; (d != NULL && d.StartPos < a.StartPos); d = d->nextNode) {
      /* skipping over unmatchable d's */ }
   begin-desc = d;
   for (d = begin-desc; (d != NULL && d.EndPos < a.EndPos); d = d->nextNode) {
      if ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos)
            [&& (d.LevelNum = a.LevelNum + 1)]) {
         /* the optional condition is for parent-child relationships */
         append (a,d) to OutputList; }
   }
}
```

Figure 3. Algorithm Tree-Merge-Anc with output in sorted ancestor/parent order

3 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 (e_1, e_2), for example, (`book, author`) (or, (`author, jane`)) in our running example. Let `AList` $=$ $[a_1, a_2, \ldots]$ and `DList` $= [d_1, d_2, \ldots]$ be the lists of tree nodes that match the node predicates e_1 and e_2, 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` $= [(a_i, d_j)]$ of join results, sorted either by (`DocId`, a_i`.StartPos`, d_j`.StartPos`) or by (`DocId`, d_j`.StartPos`, a_i`.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. In Figure 3, 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 4 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.

3.1.1 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 $O(|input| + |output|)$, 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 3.1 *The space and time complexities of Algorithm* `Tree-Merge-Anc` *are* $O(|\texttt{AList}| + |\texttt{DList}| + |\texttt{OutputList}|)$*, 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 ancestor-descendant relationship. In this case, the size of `OutputList` is $O(|\texttt{AList}| + |\texttt{DList}|)$. 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 StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

begin-anc = AList->firstNode; OutputList = NULL;
for (d = DList->firstNode; d != NULL; d = d->nextNode) {
   for (a = begin-anc; (a != NULL && a.EndPos < d.StartPos); a = a->nextNode) {
      /* skipping over unmatchable a's */ }
   begin-anc = a;
   for (a = begin-anc; (a != NULL && a.StartPos < a.StartPos); a = a->nextNode) {
      if ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos)
            [&& (d.LevelNum = a.LevelNum + 1)]) {
         /* the optional condition is for parent-child relationships */
         append (a,d) to OutputList; }
   }
}
```

Figure 4. 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 is satisfied in this case.

Consider next the case where multiple nodes in `AList` are themselves related by an ancestor-descendant relationship. This can happen, for example, in the (`section`, `head`) structural relationship for the XML data in Figure 1. In this case, multiple passes may be made over the same set of descendant nodes in `DList`, and the size of `OutputList` may be $O(|\texttt{AList}| * |\texttt{DList}|)$, which is quadratic in the size of the input lists. However, we can show that the algorithm still has optimal time complexity, i.e., $O(|\texttt{AList}| + |\texttt{DList}| + |\texttt{OutputList}|)$.

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 parent-child 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 5(a). In this case, it is easy to verify that the size of `OutputList` is $O(|\texttt{AList}| + |\texttt{DList}|)$, but the time complexity of Algorithm `Tree-Merge-Anc` is $O((|\texttt{AList}| + |\texttt{DList}|)^2)$; the evaluation is pictorially depicted in Figure 5(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 3.1 for Algorithm `Tree-Merge-Desc`, since the time complexity of the algorithm can be $O((|\texttt{AList}| + |\texttt{DList}| + |\texttt{OutputList}|)^2)$ in the worst case. This happens, for example, in the case shown in Figure 5(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 $O(|\texttt{AList}| + |\texttt{DList}|)$, but `AList` is repeatedly scanned, resulting in a time complexity of $O(|\texttt{AList}| * |\texttt{DList}|)$; the evaluation is depicted in Figure 5(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 tree-merge 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.

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.

3.2.1 Stack-Tree-Desc

Consider an ancestor-descendant structural relationship (e_1, e_2). Let $\texttt{AList} = [a_1, a_2, \ldots]$ and $\texttt{DList} = [d_1, d_2, \ldots]$ be the lists of tree nodes that match node predicates e_1 and e_2, respectively, sorted by the (`DocId`, `StartPos`) values of its elements.

[2] A clever implementation that uses a one node lookahead in `AList` can reduce the number of passes over `DList` to just one.

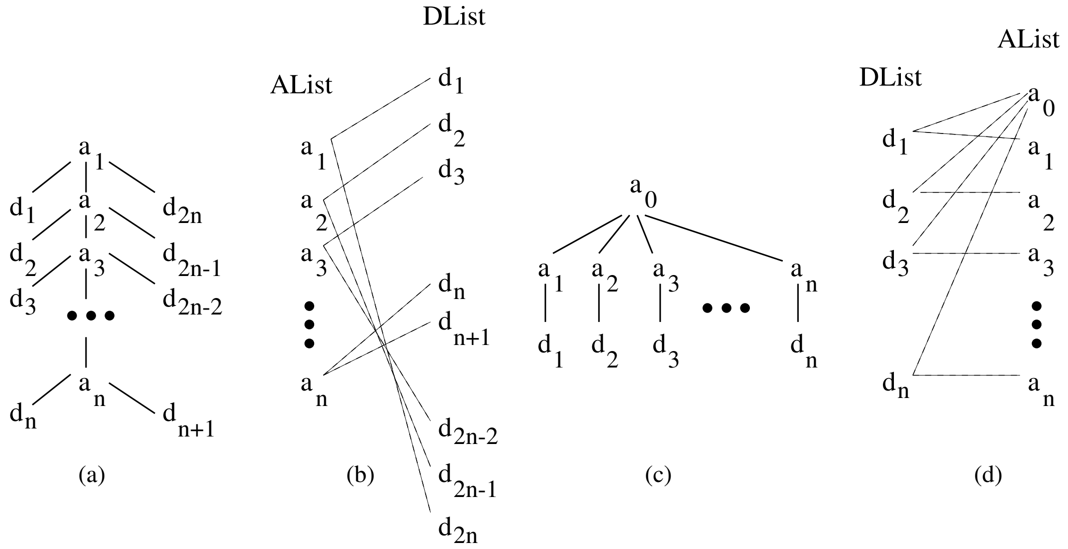


Figure 5. (a), (b) Worst case for `Tree-Merge-Anc` and (c), (d) Worst case for `Tree-Merge-Desc`

We first discuss the stack-tree algorithm for the case when the output list $[(a_i, d_j)]$ is sorted by `(DocId,` d_j`.StartPos,` a_i`.StartPos)`. This is both simpler to understand and extremely efficient in practice. The algorithm is presented in Figure 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, StartPos)` 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 `StartPos`. 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 6 needs to be replaced with:

```
if (d.LevelNum = stack->top.LevelNum + 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 7(a), are shown in Figures 7(b)–(e). The a_i's are the nodes in `AList` and the d_j's are the nodes in `DList`. Initially, the stack is empty, and the conceptual merge of `AList` and `DList` is shown in Figure 7(b). In Figure 7(c), a_1 has been put on the stack, and the first new element of the merged list, d_1, is compared with the stack top (at this point (a_1, d_1) is output). Figure 7(d) illustrates the state of the execution several steps later, when $a_1, a_2, \ldots, a_n$ are all on the stack, and d_n is being compared with the stack top (after this point, the `OutputList` includes $(a_1, d_1), (a_2, d_2), \ldots, (a_n, d_n)$). Finally, Figure 7(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 a_i's have been popped from the stack), and d_{2n} is compared with a_1. 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 5(a), which illustrated the sub-optimality of Algorithm `Tree-Merge-Anc`, for the case of parent-child structural relationships. ∎

3.2.2 Stack-Tree-Anc

We next discuss the stack-tree algorithm for the case when the output list $[(a_i, d_j)]$ needs to be sorted by `(DocId,` a_i`.StartPos,` d_j`.StartPos)`.

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 a' from `AList` that is an ancestor of a (and hence below a on the stack) is also an ancestor of d. Since the `StartPos` of a' precedes the start position of a, we must delay output of the join pair (a, d) until after (a', d) has been output. But there remains the possibility of a new element d' after d in the `DList` joining with a' as long a' is on stack, so we cannot output the pair (a, d) until the ancestor node a' is popped from stack. Meanwhile, 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 StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

a = AList->firstNode; d = DList->firstNode; OutputList = NULL;
while (the input lists are not empty or the stack is not empty) {
   if ((a.StartPos > stack->top.EndPos) && (d.StartPos > stack->top.EndPos)) {
      /* time to pop the top element in the stack */
      tuple = stack->pop(); }
   else if (a.StartPos < d.StartPos) {
      stack->push(a)
      a = a->nextNode }
   else {
      for (a1 = stack->bottom; a1 != NULL; a1 = a1->up) {
         append (a1,d) to OutputList
      }
      d = d->nextNode
   }
}
```

Figure 6. Algorithm Stack-Tree-Desc with output in sorted descendant order

output. Our solution to this problem is described in Figure 8 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 8) 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.

3.2.3 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 $O(|input| + |output|)$ in the worst case. Putting all this together, we get the following result:

Theorem 3.2 *The space and time complexities of Algorithm* Stack-Tree-Desc *are* $O(|\texttt{AList}|+|\texttt{DList}|+|\texttt{OutputList}|)$*, 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 3.3 *The I/O complexity of Algorithm* Stack-Tree-Desc *is* $O(\frac{|\texttt{AList}|}{B} + \frac{|\texttt{DList}|}{B} + \frac{|\texttt{OutputList}|}{B})$*, for ancestor-descendant and parent-child structural relationships, where B is the blocking factor.* ∎

3.2.4 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.

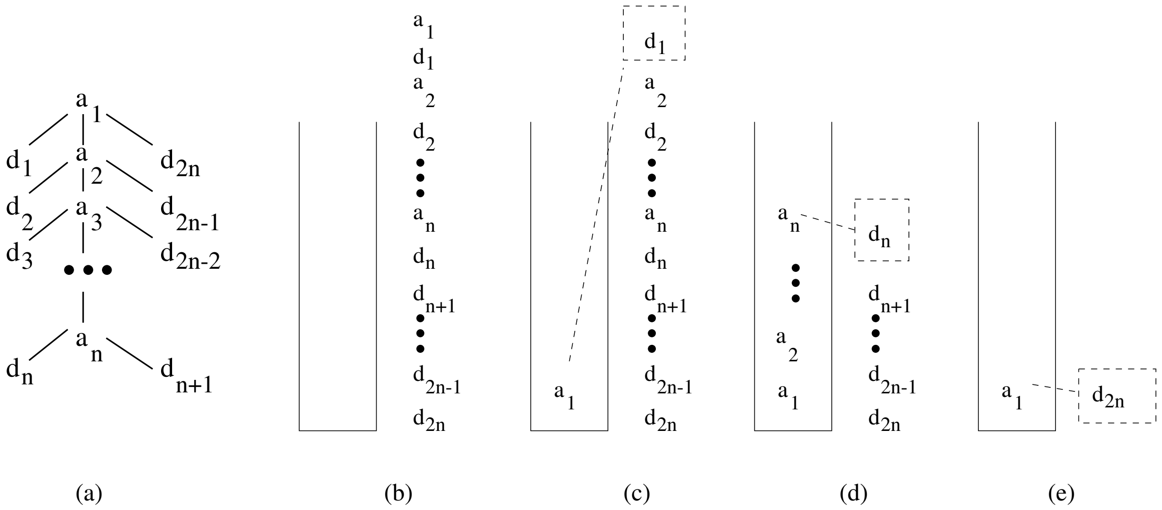


Figure 7. (a) Dataset (b)–(e) Steps during evaluation of `Stack-Tree-Desc`

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 $O(|input| + |output|)$ 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). 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.

Theorem 3.4 *The space and time complexities of Algorithm* `Stack-Tree-Anc` *are* $O(|\texttt{AList}| + |\texttt{DList}| + |\texttt{OutputList}|)$, *for both ancestor-descendant and parent-child structural relationships.*

The I/O complexity of Algorithm `Stack-Tree-Anc` *is* $O(\frac{|\texttt{AList}|}{B} + \frac{|\texttt{DList}|}{B} + \frac{|\texttt{OutputList}|}{B})$, *for both ancestor-descendant and parent-child structural relationships, where B is the blocking factor.* ∎

4 Experimental Evaluation

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, STJ-A 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.

4.1 Experimental Testbed

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 StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */

a = AList->firstNode; d = DList->firstNode; OutputList = NULL;
while (the input lists are not empty or the stack is not empty) {
   if ((a.StartPos > stack->top.EndPos) && (d.StartPos > stack->top.EndPos)) {
      /* time to pop the top element in the stack */
      tuple = stack->pop();
      if (stack->size == 0) { /* we just popped the bottom element */
         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 (a.StartPos < d.StartPos) {
      stack->push(a)
      a = a->nextNode }
   else {
      for (a1 = stack->bottom; a1 != NULL; a1 = a1->up) {
         if (a1 == stack->bottom) append (a1,d) to OutputList
         else append (a1,d) to the self-list of a1
      }
      d = d->nextNode
   }
}
```

Figure 8. Algorithm Stack-Tree-Anc with output in sorted ancestor order

4.2 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 fanout (MaxRepeats) and the maximum depth, using the Organization DTD presented in Figure 9. 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 approximately 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 queries*, and are used to demonstrate the performance of the algorithms when evaluating complex queries with multiple joins in a pipeline.

4.3 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 iterator 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 9. DTD used in our experiments

Node	Count
manager	25,880
department	342,450
employee	574,530
email	250,530

Query	XQuery Path Expression	Result Cardinality
QS1	employee/email	140,700
QS2	employee//email	142,958
QS3	manager/department	16,855
QS4	manager//department	587,137
QS5	manager/employee	17,259
QS6	manager//employee	990,774
QC1	manager/employee/email	7,990
QC2	manager//employee/email	232,406

Table 1. Description of queries and characteristics of the data set

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.

4.4 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 10 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 10, 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). With larger result sizes this list management slows down the performance of STJ-A in practice. Figure 10 also shows that the two TMJ algorithms have comparable performance.

We also ran these experiments with reduced buffer sizes, but found that for this data set the execution time of all the algorithms remained fairly constant. Even though the XML data sets that we use are large, after applying the predicates, the candidate lists that we join are not very large. 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.

4.5 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.

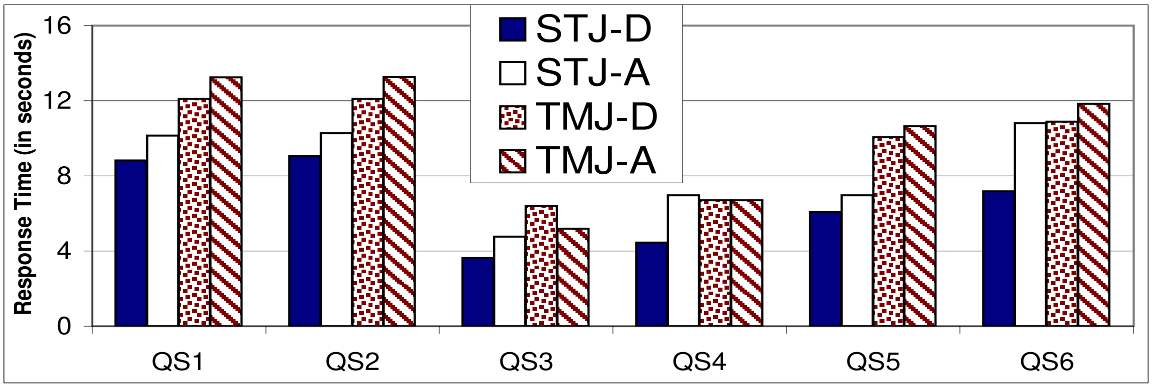


Figure 10. STJ and TMJ, simple queries: QS1–QS6

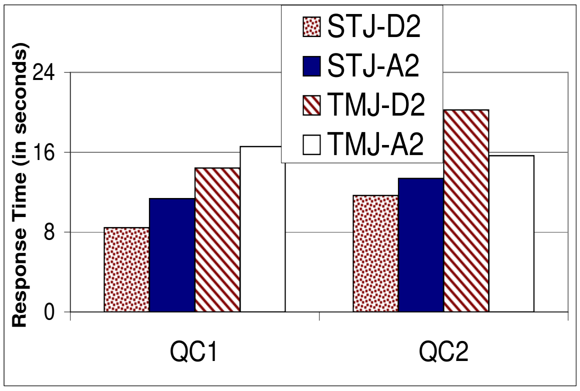


Figure 11. STJ and TMJ, complex queries: QC1, QC2

5 Related Work

Matchings between pairs of trees in memory has been a topic of study in the algorithms community for a long time (e.g., see [3] 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.

Many algorithms are known to be very efficient over tree structures. Most relevant to us in this literature are algorithms for checking the presence of sets of edges and paths. 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 *join 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, 12, 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 parent-child) 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, StartPos : EndPos, LevelNum)`, is essentially that of Consens and Milo, who considered a fragment of the PAT text searching operators for indexing text databases [8, 9], and discussed optimization techniques for the algebra. 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, StartPos : EndPos, LevelNum)` 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

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.

Acknowledgements

We would like to thank Chun Zhang for her helpful comments on an early version of this paper.

References

[1] XMach-1. Available from http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html.

[2] The XML benchmark project. Available from http://www.xml-benchmark.org.

[3] A. Apostolico and Z. Galil. Pattern matching algorithms. *Oxford University Press*, 1997.

[4] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Middleware for publishing object relational data as XML documents. *Proceedings of VLDB*, 2000.

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of SIGMOD*, 1994.

[6] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A query language for XML. W3C Working Draft. Available from http://www.w3.org/TR/xquery, Feb. 2001.

[7] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proceedings of WebDB*, 2000.

[8] M. P. Consens and T. Milo. Optimizing queries on files. In *Proceedings of SIGMOD*, 1994.

[9] M. P. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of PODS*, 1995.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to the World Wide Web Consortium 19-August-1998. Available from http://www.w3.org/TR/NOTE-xml-ql., 1998.

[11] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non equijoin algorithms. *Proceedings of SIGMOD*, 1991.

[12] M. Fernandez and D. Suciu. SilkRoute: Trading between relations and XML. *WWW9*, 2000.

[13] T. Fiebig and G. Moerkotte. Evaluating queries on structure with access support relations. *Proceedings of WebDB*, 2000.

[14] D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[15] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[16] G. Jacobson, B. Krishnamurthy, D. Srivastava, and D. Suciu. Focusing search in hierarchical structures with directory sets. In *Proceedings of CIKM*, 1998.

[17] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of SIGMOD*, 1999.

[18] N. Koudas and K. C. Sevcik. Size separation spatial join. *Proceedings of SIGMOD*, 1997.

[19] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. *Proceedings of SIGMOD*, 1996.

[20] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[21] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, 1999.

[22] U. of Washington. The Tukwila system. Available from http://data.cs.washington.edu/integration/tukwila/.

[23] U. of Wisconsin. The Niagara system. Available from http://www.cs.wisc.edu/niagara/.

[24] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. *Proceedings of SIGMOD*, 1996.

[25] G. Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, New York, 1983.

[26] 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.

[27] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of VLDB*, 1999.

[28] E. Shekita and M. Carey. A performance evaluation of pointer based joins. *Proceedings of SIGMOD*, 1990.

[29] 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.