Efficient Structural Joins on Indexed XML Documents

Shu-Yao Chien
CS Dept. UCLA
csy@cs.ucla.edu

Zografoula Vagena
CSE Dept., UC Riverside
foula@cs.ucr.edu

Donghui Zhang
CSE Dept., UC Riverside
donghui@cs.ucr.edu

Vassilis J. Tsotras*
CSE Dept., UC Riverside
tsotras@cs.ucr.edu

Carlo Zaniolo*
CS Dept. UCLA
zaniolo@cs.ucla.edu

Abstract

Queries on XML documents typically combine selections on element contents, and, via path expressions, the structural relationships between tagged elements. Structural joins are used to find all pairs of elements satisfying the primitive structural relationships specified in the query, namely, *parent–child* and *ancestor–descendant* relationships. Efficient support for structural joins is thus the key to efficient implementations of XML queries. Recently proposed node numbering schemes enable the capturing of the XML document structure using traditional indices (such as B+-trees or R-trees). This paper proposes efficient structural join algorithms in the presence of tag indices. We first concentrate on using B+-trees and show how to expedite a structural join by avoiding collections of elements that do not participate in the join. We then introduce an enhancement (based on *sibling pointers*) that further improves performance. Such sibling pointers are easily implemented and dynamically maintainable. We also present a structural join algorithm that utilizes R-trees. An extensive experimental comparison shows that the B+-tree structural joins are more robust. Furthermore, they provide drastic improvement gains over the current state of the art.

* This research was supported by NSF grants IIS-9907477, EIA-9983445, IIS-0070135, and the Dept. of Defense.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

The problem of managing and querying XML documents efficiently poses interesting challenges for database researchers. XML documents can have a rather complex internal structure; in fact, an XML document can be viewed as an ordered tree. Tree nodes correspond to document elements (or attributes) while edges represent direct element-subelement relationships. This tree-centric representation is apparent in XML languages like Quilt [7], XQuery [38] and XPath [37], which qualify documents for retrieval both by their structure and the values in their elements. For instance, a value-based selection for a document can be specified by conditions on its elements' names (tags), their attributes, and the text strings (i.e., PCDATA) contained in such elements. A structure-based selection instead relies on the structural relationships, namely: parent-child and ancestor-descendant relationships. For example, the query:

section[title="Overview"]//figure[caption="R-tree"]

finds all figures with caption 'R-tree' under sections whose title is "Overview". This complex query consists of three conditions:

- the conditions **section[title="Overview"]** and **figure[caption="R-tree"]** are value-based since they select document elements using their values or contents, while
- the double slash in `section//figure` corresponds to a structural condition, in particular, an ancestor-descendant relationship. It is a shorthand for a *path expression* specifying that there must be a path leading from the first element to the second one (i.e., the second element must be a descendant of the first in the document tree).

Using path expressions, users are allowed to navigate through arbitrary long paths in the tree. A single slash in the query (`section/figure`), would find only those figures that are children of section elements (i.e., a parent-child relationship).

Traditional indexing schemes, such as B+-trees, can be easily extended to support value based queries on XML documents. Path expression queries pose a much harder problem, requiring the computation of *structural joins* [1, 23]. Previously proposed structural join algorithms assume that the ancestor and the descendant elements are provided before the join in two ordered lists [1, 23]. However, that implies that all ancestor and descendant elements are accessed either from indices or even from physical data pages before the join. That may cause unnecessary I/O and slow down the join process. Instead we assume that an index exists in each of the joined lists. Typically, list elements belong to the same document tag (sections, figures, chapters, etc.) and tag indices are easily maintained [10]. Moreover, such indices can resolve element relationships by utilizing recently proposed *durable* numbering schemes that capture the document structure [23, 10].

This paper proposes efficient structural join algorithms in the presence of tag indices. First we concentrate on using B+-trees (with durable numbers) and show how to take advantage of the available structural information *before* joining, to filter out and minimize unnecessary data reads. We then introduce a simple enhancement that further improves performance by adding *sibling* pointers based on the notion of "containment". Such pointers are easily implemented and can be maintained dynamically. We also present a structural join algorithm that utilizes R-trees. An extensive experimental comparison shows that the B+-tree structural join enhanced with few sibling pointers provides the most robust solution. Furthermore, this approach can provide drastic improvement gains over the current state of the art.

The main contributions of this paper are summarized below:

- An efficient B+-tree based structural join algorithm is presented;
- The utilization of sibling pointers is introduced for further performance gains;
- An extensive experimental section is provided, comparing the above algorithms with an R-tree based structural join algorithm as well as the current state of the art.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 proposes the B+-tree based structural join algorithms, while section 4 discusses the R-tree based structural join. Problem variations are examined in Section 5 while Section 6 presents performance results and Section 7 concludes the paper.

2 Background and Previous Work

To efficiently address complex XML queries, it is important to: (i) quickly determine the structural relationship among any pair of tree nodes, (ii) efficiently find all occurrences of a structural relationship. We first provide background information on the importance of numbering schemes in determining structural relationships. We then discuss previous work on structural joins and indexing for XML documents.

Numbering Schemes. To locate all figures under a given section, the obvious but inefficient way is to traverse the whole subtree of the section (i.e., a top-down approach). Structural relationships can determined faster if a numbering scheme is embedded on the document's tree. One approach is to assign to each tree node three numbers, namely: the node's *preorder* and *postorder* ranks as well as its level in the XML tree [12, 39, 23, 1, 18]. In a preorder (postorder) traversal, a tree node is visited and assigned its preorder (postorder) rank before (after) its children are recursively traversed from left to right. The preorder traversal of a document's tree representation is equivalent to its *textual order*. If the document's textual representation is read sequentially, elements will be accessed in their preorder rank. Conversely, the document text can be recreated following a preorder traversal on its tree representation.

A node v is an ancestor of a node u iff $preorder(v) < preorder(u)$ and $postorder(v) > postorder(u)$. If the $(preorder, postorder)$ ranks are seen as an interval (clearly: $preorder(v) < postorder(v)$), then the ancestor contains the interval of the descendant. In the textual representation, the opening tag $\langle v \rangle$ is before $\langle u \rangle$ while the closing tag $\langle /v \rangle$ is after $\langle /u \rangle$. Furthermore, node v is a parent of node u if in addition $level(v) + 1 = level(u)$.

While easily computed, the above numbering scheme is affected by document updates; node ranks change when inserting or deleting a node. A *durable* numbering scheme was recently proposed in [23]. In this scheme, each node is assigned a $(start, end)$ interval (where $end > start$) such that: (i) for any node u and its parent node v, the interval $(start(u), end(u))$ is contained in interval $(start(v), end(v))$; (ii) for two sibling nodes u, v, if u is the predecessor of v in preorder traversal, then $end(u) < start(v)$. The *start* numbers follow the pre-order traversal, but ranges of unused numbers are left between subsequent nodes to make room for future insertions. Node v is an ancestor of a node u iff $start(v) < start(u) < end(v)$.

Figure 1 shows a sample XML document with its durable intervals (for simplicity, the tree level numbers are not shown). The root node is assigned interval [1,2100], while its three children nodes have intervals [10,600], [710,1200], and [1400,2000]. Ranges [2,9], [601, 709], [1201,1399], and [2001,2099] remain unused to handle future insertions.

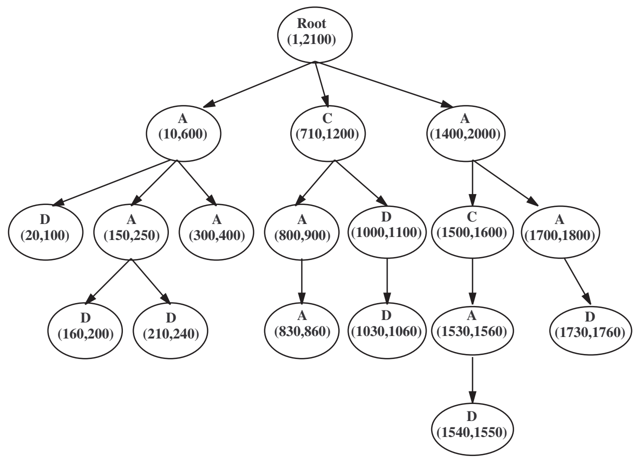


Figure 1: A sample XML document.

Note that under both numbering schemes, for any two distinct nodes u, v, the following hold: (i) the interval of u is completely before or completely after the interval of v, or, (ii) the interval of u contains or is contained by the interval of v. Hence intervals can never intersect partly.

Structural Joins. Structural join algorithms can take advantage of the numbering schemes previously discussed to expedite execution. Consider again our previous path query and assume that all sections with title **Overview** have been identified and stored in list A, while list D contains all figures with caption **R-tree**. To answer the query, we must find all occurrences of section/figure pairs that satisfy the ancestor-descendant relationship. This corresponds to computing a *structural join* [1] between the *ancestor* list A and the *descendant* list B. (Structural joins are termed "containment queries" in [39] while [23] uses the terms *EE-Join* and *EA-Join*.) More formally, given two lists of elements A and D, where each element is of the format: $\langle DocID,\ start,\ end,\ level\rangle$ the (ancestor-descendant) structural join reports all pairs $(a_i,\ d_j)$ such that (i) $a_i.DocID = d_j.DocID$, and (ii) $a_i.start < d_j.start$ and $a_i.end > d_j.end$. A parent-child structural join also requires that $a_i.level + 1 = d_j.level$.

Structural joins are considered core operations in optimizing XML queries [15, 25, 39, 23, 1]. Various techniques have been proposed using traditional relational DBMS [15, 30, 39] or native XML query engines [25]. For example, [39] proposes a variation of the merge join algorithm using multi-predicates. Nevertheless, this algorithm can perform a lot of unnecessary steps especially when computing parent-child relationships [1]. Similarly, the sort-merge join proposed in [23] may scan an element set many times during the join computation.

The Stack-Tree-Desc algorithm proposed by [1] represents the state-of-the-art in structural joins; it assumes that each element list is stored ordered on *start* and a stack mechanism is introduced to maintain elements that will be used later in the join. This leads to optimal join performance: only one sequential scan is performed on each list. Nevertheless, as it will be demonstrated in the next section, if an index is available on each ordered list, various elements that do not participate in the join will not be scanned. This can offer drastic improvement in the structural join performance.

XML Indexing. Various techniques have been recently proposed for indexing XML data. We first discuss proposals that do not facilitate a numbering scheme [17, 16, 21, 26, 27, 11]. These works create a *structural summary* of the XML document, in the form of a labelled directed graph, similar to the one used to model the XML document. The idea is to preserve all paths, while having fewer nodes and edges. Structural summaries extract structural information directly from the data. However, unlike a schema, they are not static, and thus may change with any update. They are approximate and they need to encode information about long, seldom-queried paths, leading to increased complexity. An index typically consists of a structural summary along with a stored mapping from summary nodes to data nodes. It can be used to evaluate path expressions directly (in a top-down approach) by pruning the search space. Nevertheless, since a structural summary does not contain all data nodes, many paths need to still be examined.

The solution proposed in [11] uses prefix-encoding. Paths are encoded as strings and inserted into a special index called Index Fabric which is based on Patricia tries. The index structure is tailored for path-queries that originate in the document root. Other paths require multiple index lookups or a post-processing phase. To remedy this drawback, the notion of *refined path* was proposed. However, the refined paths needs to be preselected before index loading time.

Durable node numbers provide an important advantage since they can serve as stable references for external indices [23, 10, 18]. In [23] B+-trees are used to index the document elements and attributes; parent-child relationships are supported by the structure index. In [18] node intervals are represented as points in a higher dimensional space and an R-tree is used to store these points. In [10] we presented an indexing scheme based on multiversion indices (B+-trees and R-trees) and durable numbers to efficiently answer queries in multiversion documents. The scheme includes one index for the full document and a separate index for the elements of each tag (e.g. sections, chapters, figures). All indices are built on the element *start* numbers.

3 Structural Joins using B+-trees

Existing structural join algorithms [1, 23] do not utilize index structures but sequentially scan the input lists. I/O's can be wasted for scanning elements that do not participate in the join. Recent durable numbering schemes have enabled indexing of XML files with mainstream indices, like B+-trees and R-trees. Maintaining such indices in the presence of document updates is trivial due to the reference permanence provided by the durable numbering. In this section we propose structural join algorithms using B+-trees; R-tree based structural joins are examined in section 4.

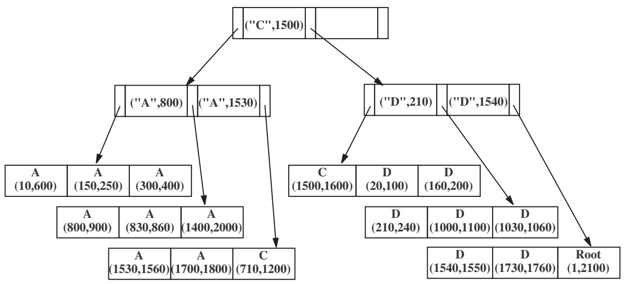
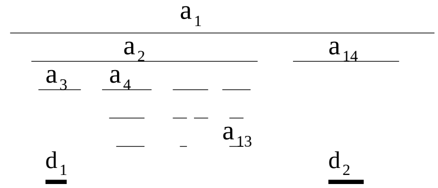


Figure 2: The B+-tree corresponding to the XML document of figure 1.

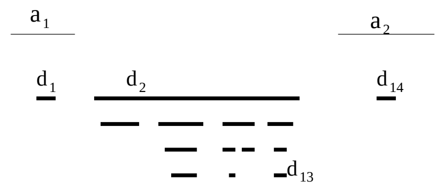
For simplicity we consider a single, typically large, document. Extension to multi-document databases is trivial. Similarly, we concentrate on the ancestor-descendant join; the parent-child join its a simple extension using the *level* numbers. (Hence in the rest, the *DocID, level* attributes are not shown but are implicitly assumed.) We assume that a separate index is used to cluster elements from the same tag (chapters, sections, figures, etc.) This index organization has been shown to be very efficient for simple path queries (i.e., when the subtree of an element does not contain elements from the same tag) [10]. In practice these multiple indices can be combined into a single index, simply by adding the tag name in the search key. An example appears in Figure 2; the index is built on the *(tag, start)* combination. (In a multi-document database, the search key would be: *(DocID, tag, start)*).

To motivate the use of indexing in structural joins, consider the examples illustrated in Figure 3. Thin line segments represent the *(start,end)* intervals of elements in the A (ancestor) list, while thick line segments correspond to element intervals in the D (descendant) list.

Figure 3a depicts a scenario where ancestor elements should be skipped. For this example, the sequential scan Stack-Tree-Desc algorithm from [1] will proceed as follows: (1) push a_1, a_2 and a_3 into the stack and join them with d_1; (2) pop a_3 and a_2 from the stack; (3) examine (push into and pop from the stack) elements a_4 through a_{13} from the A list; (4)



(a) Skip ancestor elements



(b) Skip descendant elements

Figure 3: Motivation for using the B+-tree index.

push a_{14} into the stack and then join a_{14} and a_1 with d_2. Clearly, the third step is wasteful. We can utilize the B+-tree index to omit the examination of elements a_4 through a_{13} as follows: after a_2 is popped from the stack, directly go to a_{14}. Here a_{14} is the A element having the smallest *start* which is larger than $a_2.end$.

Figure 3b illustrates the case where descendant elements should be avoided. Here, after a_1 is joined with d_1, the Stack-Tree-Desc algorithm will sequentially scan the descendant elements d_2 through d_{13}. Instead, the B+-tree index can be used to jump directly to d_{14}, which is the element from list D having the smallest *start* that is larger than $a_2.start$.

The complete B+-tree based structural-join algorithm *(Anc_Des_B+)* follows. Since the leaf pages in each B+-tree are linked together, we can view the leaf elements in that tree as a sorted list. Initially, variables a and d denote the first elements (having the smallest *start*) of the two sorted lists. Then the algorithm systematically moves a and d forward (to be the next element in their lists, etc.) and performs the join until one of the list becomes empty. During the execution, a *stack* of elements from the A list is maintained. That is, if all elements $a_1, a_2, \ldots, a_k$ where a_i is before a_{i+1}, are ancestors of d, we push a_1 through a_k into the stack before we join d with them. This is similar to the *Stack_Anc_Dec* algorithm processing. However, in algorithm *Anc_Des_B+*, steps 11 and 15 utilize the B+-trees to skip elements from the A and D lists, respectively. In practice, to avoid unnecessary B+-tree accesses, we first check whether the next ancestor element is in the same page p as the previous ancestor element (by checking the last ancestor element in p).

Algorithm *Anc_Des_B+*(List A, List D)

1. Let a, d be the first elements of A and D;
2. while (not at the end of A or D) do
3.   if (a is an ancestor of d) then
4.   Locate all elements in A that are ancestors of d and push them into *stack*;

5. Let a be the last element pushed;
6. Output d as a descendant of all elements in *stack*;
7. Let d be the next element in D;
8. else if ($a.end < d.start$) then
9. Pop all *stack* elements which are before d;
10. Let l be the last element popped;
11. Let a be the element in A (locate using B+-tree) having the smallest *start* that is larger than $l.end$;
12. else /* a is after d, or a is a descendant of d*/
13. Output d as a descendant of all elements in *stack*;
14. if (ancestor stack is empty) then
15. Let d be the element in D (locate using B+-tree) having the smallest *start* that is larger than $a.start$;
16. else
17. Let d be the next element in D;
18. endif
19. endif
20. endwhile

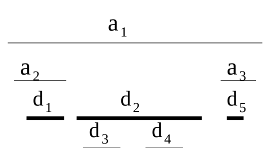


Figure 4: Skip descendants when the stack is not empty leads to error.

Moreover, step 14 shows that in order for the algorithm to correctly skip elements in the descendant list D, the stack must be currently empty. Otherwise, we may skip erroneously some D elements that need to be joined with the in-stack ancestor elements. Figure 4 shows an example of this situation. Here we assume the algorithm is currently checking ancestor a_3 against descendant d_2, while element a_1 is in *stack*. Since a_3 is after d_2, the algorithm goes to step 13. Without step 14, we would then have continued to step 15 and skip descendants d_3 and d_4. However, this would fail to join d_3 and d_4 with a_1.

3.1 Embedding the Containment Forest

We present an easily maintainable enhancement that can further improve join performance. We first introduce the concept of the *containment forest (C-forest)*. A C-forest is a structure linking the elements from the same tag. Each element corresponds to a node in the structure and is linked to other elements from the same tag via *parent*, *first-child* and *right-sibling* pointers. In the rest, the terms parent, child and sibling refer to the C-forest structure.

Next, we discuss how these three pointers are determined for each node. Given nodes n and n_p from the same tag, node n_p is called the *parent* of node n iff: (a) n_p is n's ancestor in the document tree (i.e. $n_p.start < n.start < n.end < n_p.end$); and (b) there is no other ancestor node n_a of n from the same tag, such that n_p is an ancestor of n_a. Symmetrically, we call n a child of n_p. Given same-tag nodes n and n_c, node n_c is called the *first-child* of n iff: (a) n_c is a child of n; and (b) there does not exist another same-tag node that is a child of n which is before n_c (node n_1 is before n_2 iff $n_1.end < n_2.start$). Finally, given same-tag nodes n and n_s, n_s is the *right-sibling* of n iff: (a) n and n_s have the same parent; and (b) there is no same-tag node between them which has the same parent (n_2 is between n_1 and n_3 iff $n_1.end < n_2.start$ and $n_2.end < n_3.start$). Figure 5 depicts two C-forests, for the elements with tag A and D respectively, taken from the sample document in Figure 1.

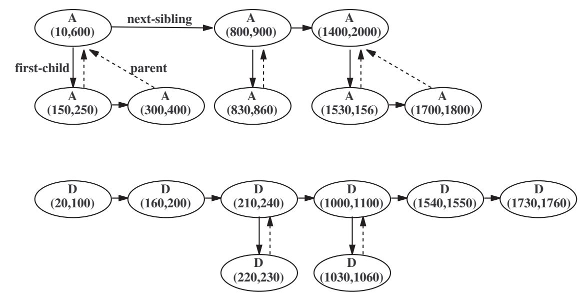


Figure 5: C-forests of tag A and D for the XML document of Figure 1.

A C-forest has the following properties:

- The (*start*, *end*) interval of each node contains all intervals in its subtree (hence the name containment forest).
- The *start* numbers in the forest follow a preorder traversal.
- The *start* (*end*) numbers of sibling nodes are in increasing order.

A C-forest for a given tag can be easily embedded within the B+-tree that indexes the tag's elements. This is accomplished by adding C-forest parent and next-sibling pointers among the leaf records of the B+-tree. First-child pointers are implicit because an element and its first child are always stored subsequently in the B+-tree. Details are discussed in the next subsection.

The *Anc_Des_B+* join algorithm also applies when the B+-tree is enhanced with the C-forest pointers. One difference appears in step 11 that finds the element a_{new} which has the smallest *start* larger than *a.end*. Consider the embedded C-forest. The relationship between a_{new} and a is as follows: if a has a right-sibling then $a_{new} = a.sibling$. If a does not have a right-sibling, but a's parent has a sibling, then $a_{new} = a.parent.sibling$, and so on. If neither a nor any of a's ancestors have a right-sibling, then the join algorithm completes since no other A elements need to be examined.

Interestingly, at step 11, all of a's ancestors are in the running stack. Since each element has now a right-sibling pointer, the address of a_{new} is identified directly, without any extra I/O. This improves algorithm Anc_Des_B+, since the B+-tree traversal is avoided in this case. Moreover, CPU time is gained as well. In the plain B+-tree, even if all related pages are in memory, we still need to search (e.g. binary search) for the new ancestor element in these pages. This is avoided with the sibling pointers, since the readily available pointer to the new ancestor has both page ID and position within the page. In the rest we refer to the enhanced index and the improved join algorithm as *B+sp-tree* and *Anc_Des_B+sp*, respectively.

3.2 Dynamic Index Maintenance

Maintaining the plain B+-tree in the presence of document updates is straightforward: each element update translates to updating the element's durable *start* number. However, when the C-forest structure is embedded, the (C-forest) parent and right-sibling pointers need be maintained effectively at each element insertion/deletion. We first present the B+sp-tree insertion algorithm; deletion is examined afterwards.

Algorithm *Insert_B+sp*(int *start*, Attribute *attr*)

1. Use the B+-tree insertion algorithm to insert a new element $(start, attr)$;
2. if (leaf page overflow occurs) then
3.   for (each element i that is moved to a new page)
4.   Adjust pointers pointing to i;
5.   endfor
6. endif
7. Find the *parent*, *left sibling*, *right sibling*, and *first child* of the new element and link them with it.

Basically, we first insert a new element into the B+-tree using *start* as key, and then we adjust the right-sibling and parent pointers among leaf elements. The core step is step 7, which locates among the existing elements which will be the parent, left sibling, right sibling, and first child of the newly inserted element. Once the related elements are found, linking the new element into the C-forest is easy. (Identifying the possibly many children under the new element is performed starting from its first child and following right-sibling pointers. The right sibling pointer of the last child is set to NULL.)

We now focus on how to locate all related elements. To locate the left sibling of the new element a, we first use the B+-tree to locate the element e with the largest *start* which is smaller than *a.start*. Clearly, either e is the parent of a, or e is before a. If e is a's parent, then a does not have any left sibling. Otherwise, consider the parent of e. Again, either *e.parent* is a's parent, or it is before a. In the first case, e is the left sibling of a. In the second case, we recursively consider the parent of *e.parent*, and so on. Eventually, if the highest ancestor of e in the C-forest is before a, then this becomes the left sibling of a.

The procedure of finding the left sibling also identifies a's parent. To locate the right sibling of a, we use the B+-tree to find the element e with the smallest *start* that is greater than *a.end*. If *e.parent* is the same as the parent of a, e is the right-sibling of a; otherwise a does not have any right-sibling.

To locate the first child of a is simpler. Since the leaf elements in the B+-tree are sorted by *start* number, we simply examine the element stored right after a. This element is either the first child of a, or after a. We can decide whether it is a first child simply by checking whether the *end* of the element is larger than *a.end*.

Finally, if a page overflow takes place after we insert the new element (steps 2-6), the B+-tree insertion algorithm will move some leaf records into another page. The pointers pointing to these moved records need to be adjusted, but this procedure is similar to the above.

The following algorithm discusses how to delete an element from the B+sp-tree:

Algorithm *Delete_B+sp*(Pointer a)

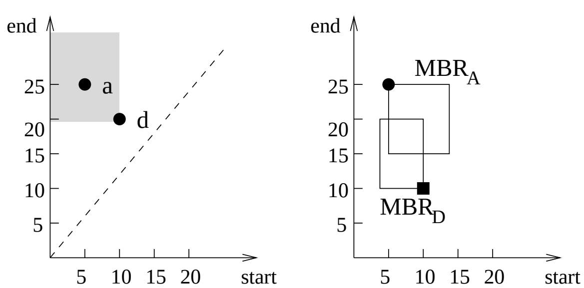
1. Locate the left sibling and the first child of a;
2. Adjust the C-forest pointers of the related elements;
3. Use the B+-tree deletion algorithm to delete element a;
4. if (a leaf page underflow occurs) then
5.   for (each element i that is moved to a new page)
6.   Adjust pointers pointing to i;
7.   endfor
8. endif

We first need to unlink the element a from the C-forest. Again, the core step is to locate the related elements (parent, first child, left sibling, right sibling). Different from the insertion algorithm, there is no need to locate the parent and right sibling elements of a, since the pointers to these elements are already stored along with a. Once the related elements are identified, it is straightforward to adjust the C-forest pointers: for the children of a's children, we should set the parent pointers of them to point to *a.parent*; for the left sibling of a, we should set its right sibling pointer to point to the first child of a; for the last child of a, we should set its right sibling pointer to point to $a.right$. Next, we use the normal B+-tree deletion algorithm to delete element a. If a page underflow occurs, records will be redistributed to a sibling page. In this case, we need to adjust the pointers which point to the elements that are redistributed.

To analyze the efficiency of the insertion/deletion algorithms of the B+sp-tree, we first define some the terms max-depth and max-span. Given a C-forest, the *depth* of a node is the number of ancestors the node has. Then **max-depth** denotes the maximum depth of any node in the forest, while **max-span** corresponds to the maximum number of children under any node. These parameters depend on the document characteristics. Elements with longer intervals tend to attain many children elements and create deeper subtrees (and thus increase the max-span and max-depth). The efficiency of the B+sp-tree insertion/deletion algorithm is given below. Due to space limitations the proof of the theorem is omitted.

Theorem 1 *The amortized insertion/deletion cost of a B+sp-tree is $O(h + s + d)$, where h is the height of the B+-tree and s, d are the max-span and max-depth of the embedded C-forest.*

4 Structural Join using R-trees



(a) two points (b) two pages

Figure 6: The ancestor-descendant relationship in the transformed space.

Multidimensional indices (like the R-tree [19, 5]) offer an alternative in indexing XML documents [10, 18]. Consider the *(start,end)* interval of element e. This interval is mapped to a point ($e.start$, $e.end$) in the two-dimensional space which is then indexed by an R-tree. Figure 6a depicts two points corresponding to elements $a : (5, 25)$ and $d : (10, 20)$. Since $end > start$ for any element, all points in this representation lie above the diagonal. Moreover, the following properties hold:

- An element a is an ancestor of element d iff the corresponding point of a is located on the upper left side of point d (Figure 6a).
- Let A, D be two R-tree pages that store points from the ancestor and descendant lists, respectively. Let MBR_A and MBR_D be the minimum bounding rectangles of these pages. Page A may contain an ancestor of some point in page D iff the upper left corner of MBR_A (the black dot in figure 6b) is located on the upper left side of the lower right corner of MBR_D (the black square in figure 6b).

Typically, spatial join algorithms join two sets of rectangular objects and report pairs of objects that intersect. If each set is indexed by an R-tree, a *synchronized tree traversal (STT)* is followed [4, 20]. Initially, the two root pages are joined. To join a pair of index pages, every element in the first page is joined with every element in the second page, if they intersect. Eventually, at the leaf level, two objects are joined if they intersect. The R-tree based structural join algorithm that we propose also uses STT. The difference is that instead of using the intersection condition, the condition to join two pages A and D is that A's upper left corner is located to the upper left of D's lower right corner; and the condition to join two objects a and d is that a is located to the upper left of d.

5 Problem Variations

5.1 Parent-Child Join

All the three ancestor-descendant structural join algorithms can be directly used to answer parent-child joins. An additional condition is performed on the levels of each (ancestor,descendant) pair found.

5.2 Self Joins

If both ancestor and descendant elements have the same tag (self join), a simpler algorithm applies. For example, in the self join case, if we maintain two pointers a and d as the previous algorithms do, the two pointers may point to the same element, a case which (for simplicity) is omitted from the previous algorithms. Moreover, the B+-tree index algorithms were used to skip sub-trees when possible. In the self-join case, however, as long as a sub-tree has more than one element, every element of the sub-tree will appear in the join result (e.g. every element is a descendant of the sub-tree's root). We could use the R-tree join algorithm. However, there is no guarantee that each disk page will be examined at most once.

We hereby propose an non-indexed self join algorithm, assuming that the elements are sorted in ascending order of *start*:

Algorithm *Self_Anc_Des*(List A)

Let a be the first element in A;
while (not at the end of A)
    Pop all *stack* elements which are before a;

4. Output a as a descendant of the remaining elements in *stack*;
5. Push a into *stack*;
6. Let a be the next element in A.
7. endwhile

The algorithm traverses the list exactly once, while maintaining a stack in which any adjacent pair of elements has parent-child relationship. For each element a being examined, we first pop from the stack those elements that are before a since they will not join with any later elements. We then join a with the in-stack elements and finally push a into the stack.

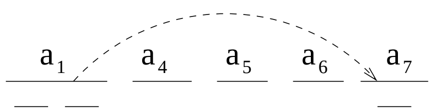


Figure 7: The next-useful-tree pointer helps to speedup self join.

Discussion: If there are many elements which do not have any ancestor or descendant at all (e.g. elements a_4, a_5, a_6 in Figure 7), algorithm Self_Anc_Des is not efficient since it has to go through the whole list. One possible optimization is to link *useful* trees (i.e. trees having at least two elements): in figure 7 a link connects a_1 to a_7. Algorithm Self_Anc_Des will then skip elements which do not participate in the join.

5.3 Structural Join in a Pipelining Environment

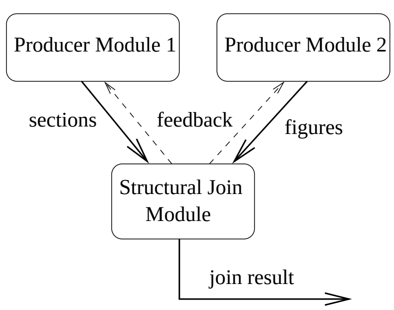


Figure 8: Structural join engine.

We consider the structural join problem in a different scenario: the input lists are not known beforehand, but are generated by previous-stage execution modules in a pipelined environment. For example, in figure 8, the inputs to the join arrive from producer modules 1 and 2. In this scenario, the structural join module can not benefit from pre-built indices. Thus, if the structural join module cannot send any feedback to the producer modules, the best approach is to use the non-indexed sequential algorithm proposed in [1].

However, if feedbacks are allowed (i.e., the join module is "active"), we can still use the Anc_Des_B+ algorithm to improve the performance, even though the B+-tree indices are not available to the structural join module. The idea is as follows. In steps 11 of algorithm Anc_Des_B+, we skip ancestors by locating (using the B+-tree) the element which the smallest *start* larger than *a.end*, where a is the ancestor being examined currently. In the pipelining environment, the join module should send a feedback asking producer module 1 to skip ancestors accordingly. Similarly, in step 15, the join module should ask producer module 2 to skip descendants. Simply not sending the skipped data across execution modules would reduce the size of intermediate results. Moreover, a clever producer module might be able to use this feedback to further optimize its data generation plan.

6 Performance Analysis

We compare the performance of the non-indexed join algorithm (Stack-Tree-Desc [1]), the B+-tree join algorithm (Anc_Des_B+), the B+-tree join with sibling pointers (Anc_Des_B+sp), as well as two R*-tree join algorithms (using breath-first and depth-first traversals). Moreover, as will be explained later in this section, we also implemented an optimization of Anc_Des_B+sp, using a partial list (i.e., the most important) of the sibling pointers. The notation used in the performance figures is described in table 1.

6.1 Experimental Setup

```
<!ELEMENT department (name, email?, employee+)>
<!ATTLIST department nodeID ID #REQUIRED startPos CDATA
 #REQUIRED endPos CDATA #REQUIRED level CDATA #REQUIRED
 docID CDATA #FIXED "1">
<!ELEMENT employee (employe*, name+, email?)>
<!ATTLIST employee nodeID ID #REQUIRED startPos CDATA
 #REQUIRED endPos CDATA #REQUIRED level CDATA #REQUIRED
 docID CDATA #FIXED "1">
<!ELEMENT name (#PCDATA)>
<!ATTLIST name nodeID ID #REQUIRED startPos CDATA
 #REQUIRED endPos CDATA #REQUIRED level CDATA #REQUIRED
 docID CDATA #FIXED "1">
<!ELEMENT email (#PCDATA)>
<!ATTLIST email nodeID ID #REQUIRED startPos CDATA
 #REQUIRED endPos CDATA #REQUIRED level CDATA #REQUIRED
 docID CDATA #FIXED "1">
```

Figure 9: The DTD for the experimental data.

For the purposes of our experiments we have implemented a storage manager with an LRU replacement policy as well as the B+ tree index structure, and its enhancement using the C-forest. Both access methods provide bulk-loading facilities. All the algorithms were implemented in C++ and compiled using the GNU C++ compiler, under Linux Mandrake 8.0. The experiments were contacted on a Pentium III 1.2GHz processor.

Notation:	Meaning:	Section:
no_index	structural join using sequential scan (Stack-Tree-Desc [1])	2
B+	structural join using B+ tree indices (Anc_Des_B+)	3
B+sp	structural join using B+ trees with sibling pointers (Anc_Des_B+sp)	3.1
B+psp	structural join using B+ trees and partial list of sibling pointers	3.1
*R*_df*	depth first structural join using R*-trees	4
*R*_bf*	breadth first structural join using R*-trees	4

Table 1: Implemented Algorithms

We used synthetic data for all our experiments in order to control the structural and consequently join characteristics of the XML documents. We generated several XML files using the IBM XML data generator [36] providing it with the DTD presented in figure 9. The derived documents were further processed to obtain the desired data workloads.

We parsed the generated XML files in order to add the *(start,end)* numbers to each element node using an event-based XML parser [35] that we implemented in Java. The parser uses a stack that grows to the maximum nesting-depth of the XML nodes and assigns the durable numbers in two passes over the XML file. One pass allocates the durable numbers while the second pass integrates them into the XML text file as attributes to each node. Subsequently, we parsed the generated file to create the XML node lists. The elements of each list include the attributes of the associated node, as well as its *(start,end)* interval. All lists are stored in binary format. The data are then bulkloaded into the B+trees. For the experiments we used a page size of 8K. The size of the lists varies from 20M to 100M. To measure the join performance we count the CPU time of each algorithm as well as the number of I/O's. Note that because of the merge join characteristics of the structural join, most I/O's are non-sequential. We actually tested all algorithms for their I/O characteristics and found that between 75% and 80% of their I/O activity was random.

6.2 Join Performance Comparison

Determining the Buffer Pool Size: We started the experiments by varying the buffer pool size. Precisely, we tried sizes of 20, 40, 80, 100, 150 and 200 pages. As far as the structural no_index join algorithm is concerned, performance is not greatly affected by the buffer size. We observed the same behavior for the B+-tree algorithms, when the buffer pool contains 80 or more pages. The reasons are:

1. The no_index join algorithm scans the lists only once. As a result, after a page is visited for the first time and then flushed out, it will not be requested again in the future.

2. The same holds for the B+-tree algorithms. The difference in performance for smaller sized buffer pools is attributed to the fact that some index pages might be flushed out and then requested again. However, with larger buffer pool sizes and with LRU policy, chances are that the index pages will remain in the buffer pool.

Hence in the following experiments the buffer pool size is fixed to 80 pages.

Join Performance

We performed experiments varying the relative join characteristics of the ancestor and descendant lists. We also varied the result sizes. To identify the performance characteristics of each algorithm, we first examined their behavior by varying the percentage of elements from the ancestor list that join with descendants. In this scenario, skipping (mainly) ancestors becomes gradually more important. We then considered the alternative case of varying the percentage of descendant elements that participate in the join result. Here, the ability to (mainly) skip descendants will improve join performance. Finally, we examined scenarios where skipping in both lists is possible.

Skipping Ancestors: In the first group of experiments we kept the percentage of the descendants that are joined with at least one ancestor high (90%), and we varied the percentage of the joined ancestors. For that purpose, we started with two lists where both ancestors and descendants are joined equally high (90%) and gradually reduced the percentage of the joined ancestors. This reduction is performed by effectively removing elements from the descendant file (while at the same time maintaining the descendant join percentage around 90%). The join performance is shown in table 2.

As can be seen from the table, all the algorithms, except the R-tree based ones, perform similarly. The performance loss that the B+sp algorithm faces in comparison with the B+-tree and the no_index joins is attributed to the increase of the file size due to the addition of sibling pointers. For the B+-tree, the only overhead that occurs is the initial loading of the index pages. However, this is quite small. The R-tree algorithms very often need to examine a page multiple times since their data pages are not completely clustered by the element *start*numbers.

B+psp Optimization: We thus offer an optimization that can combine the advantages of the plain B+-tree as well as the sibling pointers. Typically, the sibling pointer for element e, is beneficial when it points to an element that is in a different page than e. Hence

Join Ancestors	no_index	B+	B+psp	B+sp	R*_df	R*_bf
90%	182	180	180	190	228	230
70%	150	149	150	155	196	196
55%	132	130	130	140	178	178
40%	109	108	108	114	156	156
25%	86	84	84	90	130	130
15%	74	67	67	70	119	119

Table 2: Join Performance (#sec) when 90% of ancestors join with varying proportion of descendants.

we can reduce the additional space overhead by maintaining only the useful sibling pointers. Experimental measurements that we contacted on our data, showed that on average a very small number of sibling pointers (usually 1% to 2% of the total number of sibling pointers) actually points to an element on a different page. We experimented with two different policies on how to decide which sibling pointers to maintain. One policy keeps the sibling pointer of an element whose subtree contains more than a fixed number, say 100, elements in the C-forest. The other policy kept the pointers that lead to a different page. The latter was more efficient and was implemented in the rest of our experiments (B+psp). In the results depicted in table 2 the B+psp algorithm performs equally well as the B+-tree join. However, as it will be shown later it can get even better performance. In the following experiments we discuss only the optimized version of the sibling-pointer join (B+psp).

Skipping Descendants: Next, we kept the percentage of ancestors that are joined with at least one descendant high (85%), and varied the percentage of the joined descendants. In this case, it becomes beneficial to be able to skip descendant nodes. The join performance is shown in figure 10.

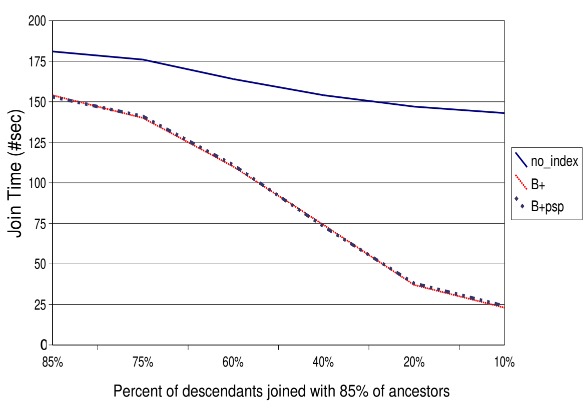


Figure 10: Comparing the join performance.

As the percentage of joined descendants decreases the B+ and B+psp algorithms outperform the no_index join. This is because the latter is constrained to sequentially scan the two lists. The B+-tree algorithms avoid descendants by skipping elements. The R-tree algorithms performed much worse than the no_index join (around 80% more time) and are not depicted.

Actually, the gain in performance when using the indexed joins is greater in the case of excluding descendants than in the case of excluding ancestors. This is attributed to the fact that, when in need to exclude ancestors, the algorithms cannot take full advantage of the place of the current descendant to decide where the next candidate ancestor resides. Instead they can only jump past the current ancestor node. That may lead to visiting ancestor nodes that turn out not to join with any descendant nodes.

Skipping both Ancestors and Descendants: In the last group of experiments we varied the percentage of both ancestors and descendants that can be joined. Among all experiments we only show the results where 1% of ancestors elements participate in the join, while varying the number of descendants. The results are shown in figure 11. Clearly, the B+-tree and B+psp perform better than the no_index and R-tree joins. Actually, here, the R-tree joins are similar to the no_index case, since only few ancestors join and the R-trees can locate them quickly. When the number of descendants decreases, the B+psp tree performs better than the B+tree. This is because the algorithm takes advantage of the sibling pointers at the ancestor nodes when it needs to access pages where the plain B+tree would need to traverse full index paths to find them.

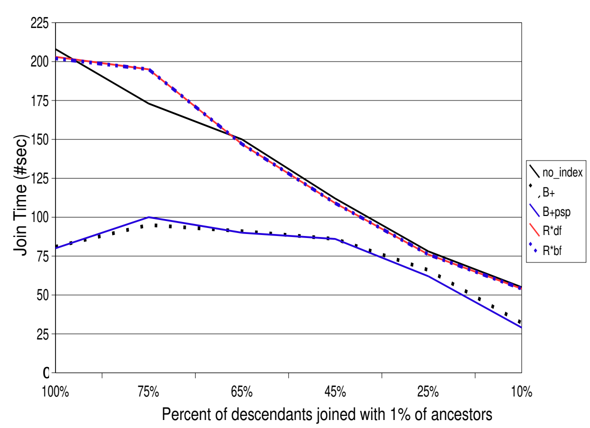


Figure 11: Comparing the join performance.

7 Conclusions & Future Work

In this paper, we proposed various indexed structural join algorithms over XML data. Performance results prove that the indexed algorithms are more robust than the state-of-art algorithms, which do not utilize index structures. Among the indexed approaches, the B+-tree with sibling pointer performs the best.

We have also shown that traditional indexing schemes are easily enhanced to support our structural join algorithm—besides the more conventional value-based searches. Finally, we have shown how these indexing schemes can be made *durable*, and thus support updates on XML documents, or the preservation of multiple versions of these documents [10]. These results are applicable to both the situation where native storage managers are used to store and query XML documents, and the situation where more traditional database systems are used for that purpose.

As future work to this paper, we plan to examine the indexed structural join over multi-versioned documents.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching", *Proc. of ICDE*, 2002.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. L. Wiener, "The Lorel Query Language for Semistructured Data", *Journal on Digital Libraries* 1(1), 1997.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multiversion B-Tree", *VLDB Journal* 5(4), 1996.

[4] T. Brinkhoff, H. Kriegel and B. Seeger, "Efficient Processing of Spatial Joins using R-trees", *Proc. of SIGMOD*, 1993.

[5] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, 1990.

[6] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi and L. Tanca, "XML-GL: A Graphical Language for Querying and Restructuring XML", *Proc. of WWW Conf.*, 1999.

[7] D. Chamberlin, J. Robie, D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", *Proc. of WebDB*, 2000.

[8] S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom, "Change Detection in Hierarchically Structured Information", *Proc. of SIGMOD*, 1996.

[9] S.-Y. Chien, V.J. Tsotras and C. Zaniolo, "Efficient Management of Multiversion Documents by Object Referencing", *Proc. of VLDB*, 2001.

[10] S.-Y. Chien, V.J. Tsotras, C. Zaniolo, and D. Zhang, "Efficient Complex Query Support for Multiversion XML Documents", *Proc. of EDBT*, 2002.

[11] B.F.Cooper, N. Sample, M.J.Franklin, G.R.Hjatlason, and M. Shadmon, "A fast index for semistructured data", *Proc. VLDB*, 2001.

[12] P.F. Dietz and D. Sleator, "Two Algorithms for Maintaining Order in a List", *Proc. STOC*, 1987, pp: 365-372.

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, "A Query Language for XML" , *Proc. of WWW Conf.*, 1999.

[14] M. Fernandez and D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", *Proc. of ICDE*, 1998.

[15] D. Florescu and D. Kossman, "Storing and Querying XML Data Using an RDBMS", *IEEE Data Engineering Bulletin*, 22(3), pp: 27-34, 1999.

[16] A. Gionis, M. Garofalakis, R. Rastogi, S. Seshadri and K. Shim, "XTRACT: A system for extracting document type descriptors from XML documents", *In Proc. SIGMOD*, 2000.

[17] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructural databases", *Proc. of 23rd VLDB Conf.*, 1997.

[18] T. Grust, "Accelerating XPath Location Steps", *Proc. of SIGMOD*, (to appear) 2002.

[19] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proc. of SIGMOD*, 1984.

[20] Y. Huang, N. Jing and E. Rundensteiner, "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations", *Proc. of VLDB*, 1997.

[21] R. Kaushik, P. Shenoy, P. Bohannon and E. Gudes. "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data" *Proc. of ICDE*, 2002.

[22] A. Kumar, V. J. Tsotras and C. Faloutsos, "Designing Access Methods for bitemporal Databases", *IEEE TKDE* 10(1), 1998.

[23] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions", *Proc. of VLDB*, 2001.

[24] D. Lomet and B. Salzberg, "Access Methods for Multiversion Data", *Proc. of SIGMOD*, 1989.

[25] J. McHugh and J. Widom, "Query optimization for XML", *Proc. of VLDB*, 1999.

[26] T. Milo and D. Suciu, "Index structures for path expressions", *Proc. ICDT*, 1999.

[27] S. Nestorov, J. Ullman, J. Weiner and J. Chawathe. "Representative Objects: Concise representations of semistructured hierarchical data", *Proc. ICDE*, 1997.

[28] B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time-Evolving Data", *ACM Computing Surveys* 31(2), 1999.

[29] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. J. DeWitt and J. F. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities" *Proc. of VLDB*, 1999.

[30] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt and J.F. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proc. VLDB*, 1999.

[31] F. Tian, D. J. DeWitt, J. Chen and C. Zhang, "The Design and Performance Evaluation of Various XML Storage Strategies", http://www.cs.wisc.edu/niagara/ Publications.html

[32] V.J. Tsotras and N. Kangelaris, "The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries", *Information Systems* 20(3), 1995.

[33] Y. Tao and D. Papadias, "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries", *Proc. of VLDB*, 2001.

[34] P. Varman and R. Verma, "An Efficient Multiversion Access Structure", *IEEE TKDE* 9(3), 1997.

[35] SAX (Simple API for XML), http://sax. sourceforge.net

[36] IBM XML Generator, http://www.alphaworks. ibm.com/tech/xmlgenerator

[37] World Wide Web Consortium, "XML Path Language (XPath)", version 1.0, Nov 16, 1999. http:// www.w3.org/TR/xpath.html

[38] World Wide Web Consortium, "XQuery 1.0: An XML Query Language", W3C Working Draft Jun 7, 2001 (work in progress). http://www.w3.org/TR/xquery/

[39] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo and G.M. Lohman, "On Supporting Containment Queries in Relational Database Management Systems", *Proc. SIGMOD*, 2001.

Appendix

Disclaimer: the appendix is provided for convenience of the reviewers and should not be counted towards the 12-page limit.

A Non-Indexed Structural Join Algorithm of [1]

We review the *Stack-Tree-Des* structural join algorithm of [1]. The algorithm joins two sorted lists A and D, where A is the list of anticipated ancestors and D is the list of anticipated descendants, both sorted by increasing order of *start*.

Algorithm *Stack-Tree-Des*(List A, List D)

```
Let a, d be the first elements of A and D;
while ( the input lists are not empty or the stack is not empty ) do
    if ( a.start > stack.top.end and d.start > stack.top.end) then
        stack.pop();
    else if ( a.start < d.start ) then
        stack.push(a);
        Let a be the next element in A;
    else
        Output d as a descendant of all elements in stack;
        Let d be the next element in D;
    endif
endwhile
```

The basic idea is to merge the two sorted lists A with D. As the merge proceeds, a stack is maintained to keep track of all ancestors $a_{i_1}, a_{i_2}, \cdots$ for current ancestor element a_k. As the merge proceeds, the algorithm determines the ancestor-descendant relationship, if any, between the current top of stack and the next element to examine, i.e. having a smaller *start*.

When the next node is a descendant of the current top of stack, it is simply pushed onto the stack. When a new node from D is found to be a descendant of current top of stack, we know that it is a descendant of all the elements in the stack. Also, it is quaranteed that it won't be a descendant of any other node in A. If the new node in the merge list is not a descendant of the current top of stack, then it is guaranteed that no future node in D is a descendant of the current top of stack, so the top element is poped and repeat the test with the new top of stack.

The above algorithm will output results in ascending order of descendant elements. [1] also proposed a variation of the algorithm to output in ascending order of ancestor elements. The main idea is that along with each in-stack ancestor element, a buffer of produced results are maintained; whenever an element is popped from the stack, the result pairs stored in the buffer of it are kept in the buffer of the new stack head; if the last element of the stack is popped, the result pairs stored in the buffer of it are actually reported.