XR-Tree: Indexing XML Data for Efficient Structural Joins[*]

Haifeng Jiang Hongjun Lu[‡] Wei Wang
Department of Computer Science
Hong Kong University of Science and Technology
Hong Kong SAR, China
{*jianghf, luhj, fervvac*}*@cs.ust.hk*

Beng Chin Ooi
School of Computing
National University of Singapore
Singapore
ooibc@comp.nus.edu.sg

Abstract

XML documents are typically queried with a combination of value search and structure search. While querying by values can leverage traditional database technologies, evaluating structural relationship, specifically parent-child or ancestor-descendant relationship, between XML element sets has imposed a great challenge on efficient XML query processing.

This paper proposes XR-tree, namely, XML Region Tree, which is a dynamic external memory index structure specially designed for strictly nested XML data. The unique feature of XR-tree is that, for a given element, all its ancestors (or descendants) in an element set indexed by an XR-tree can be identified with optimal worst case I/O cost. We then propose a new structural join algorithm that can evaluate the structural relationship between two XR-tree indexed element sets by effectively skipping ancestors and descendants that do not participate in the join. Our extensive performance study shows that the XR-tree based join algorithm significantly outperforms previous algorithms.

1. Introduction

As XML is gaining unqualified success in being adopted as a universal data exchange format, particularly in the World Wide Web, the problem of managing and querying XML documents poses interesting challenges to database researchers. Although XML documents could have rather complex internal structures, they share the same data type underlying the XML paradigm: *ordered trees*. Tree nodes represent document elements, attributes or text data, while edges represent the element-subelement (or parent-child) relationship.

To retrieve such tree-shaped data, several XML query languages have been proposed in the literature. Examples include XPath [9] and XQuery [5]. XQuery is being standardized as a major XML query language. The main building block of XQuery is XPath, which addresses part of XML documents for retrieval, both by value search and structure search. For example, "`paragraph//section`" is to find all sections that are contained in each paragraph. Here, the double slash '//' represents the ancestor-descendant relationship. A single slash '/' in an XPath represents a parent-child relationship, for example "`section/figure`". To evaluate the query "`paragraph//section`", a naïve tree traversal strategy could cause a scan of the whole XML data tree even when there are few results. Alternatively, the "set-at-a-time" strategy would first retrieve all `paragraph` and `section` elements, possibly with some *tag index*, and then find all occurrences of the ancestor-descendant relationship between the element sets. This is termed as *structural join* in [22].

Since *structural join* takes the tiger share of time for evaluating path expression queries, it has attracted a lot of interest from the research community [25, 18, 22]. The proposed algorithms take advantage of a numbering scheme that encodes each element with a $(start, end)$ pair, or the *region* of each element in a data tree [25].

The state-of-the-art structural join algorithm was proposed in [22], which takes as input two ordered lists, one for ancestors and the other for descendants. By maintaining an in-memory stack, the algorithm requires to scan the two input lists only once. Such an approach, however, implies that every ancestor or descendant is accessed once no matter it has matches or not. The question is: can we skip ancestors and descendants that have no matches in a join?

Chien et al recently proposed a new structural join algorithm, namely *B+* algorithm, which can utilize B^+-tree indexes built on the $start$ attribute of the joining element sets [8]. Although the *B+* algorithm can effectively skip descendants without matches (by B^+-tree range queries), it is not effective in skipping ancestors. As such, using B^+-trees

[*]This work was conducted at the School of Computing, National University of Singapore, Singapore.

[†]The project was partly supported by the Research Grant Council of the Hong Kong SAR, China (grants AoE/E-01/99, HKUST6060/00E).

[‡]This author's work was also supported by a grant from the National 973 project of China (No. G1998030414).

only solves half of the problem.

In this paper, we propose XR-tree (or XML Region Tree), a dynamic external memory index structure specially designed for XML data. Different from traditional B$^+$-trees, XR-trees index element nodes on their region codes, specifically $(start, end)$ pairs. The novel feature of XR-tree is that, for any element E, all its ancestors (or descendants) in a given element set $\mathcal{E}$ indexed by an XR-tree can be retrieved with optimal $O(\log N + R)$ worst case I/O cost, where N is the size of $\mathcal{E}$ and R is the number of elements retrieved. Such a unique feature of XR-tree makes it possible to most effectively skip both ancestors and descendants during a structural join if XR-trees are built on two joining element sets. The idea of XR-tree is motivated by an internal memory data structure: interval trees [4]. There are also works on interval management in external memory [1] and indexing time intervals [13]. XR-tree stands out among those proposed approaches in that it deals specifically with *regions* of XML elements while existing approaches manage arbitrary one-dimensional intervals. By fully exploiting the strictly nested property of XML, we are able to deploy more efficient data structures for managing regions of XML elements.

We summarize the contributions of this paper as follows:

1. We propose a novel external memory index structure, XR-tree, which supports efficient retrieval of elements by structural relationship. We further show that XR-tree takes linear storage consumption and can be dynamically maintained in a very efficient manner.
2. We present a new structural join algorithm, namely *XR-stack*, which utilizes XR-tree indexes on two joining element sets to effectively skip both ancestors and descendants that do not participate in a join.
3. An extensive performance study was conducted on the *XR-stack* algorithm, in comparison with previous state-of-the-art algorithms. Our experimental results show that the *XR-stack* algorithm performs significantly better than existing approaches.

The rest of the paper proceeds as follows. Section 2 is dedicated to some background knowledge and previous work on XML. We describe the data structure of XR-tree in section 3 and then show how XR-trees can be dynamically maintained in section 4. In section 5, we present two types of structural queries supported by XR-tree with I/O cost analysis. Afterwards, we present an efficient structural join algorithm, *XR-stack*, which utilizes XR-trees built on joining element sets. Section 6 reports experimental results. Section 7 concludes the paper.

2. Background and related work

XML data is commonly modelled by a tree structure, where nodes represent elements, attributes and text data, and parent-child pairs represent nesting between XML elements. To efficiently evaluate XML queries, it is important to: (a) efficiently determine structural relationship, specifically parent-child or ancestor-descendant relationship, between any pair of element nodes; (b) find all occurrences of a structural relationship between two element sets.

In this section, we first give some background information on numbering schemes for XML. Then we discuss existing structural join algorithms.

2.1. XML numbering scheme

The structural relationship between two element nodes can be quickly determined by a region encoding scheme, where each element is assigned with a pair of numbers $(start, end)$, based on its position in the data tree [25, 22, 8], with the following held: for any two distinct elements u and v, (1) the region of u is completely before or after v, or (2) the region of u completely contains v or is contained by the region of v. Formally, element u is an ancestor of element v iff $u.start < v.start$ and $v.end < u.end$. Since regions of two distinct elements never intersect partially, the formula can be simplified as $u.start < v.start < u.end$.

Region codes for element nodes can be effectively generated by a depth-first traversal of the tree and sequentially assigning a number at each visit [25, 18]. Figure 1 depicts an example XML data tree where elements are encoded in this manner. The root is a $dept$ element, which spans from position 1 to 100. The first employee element, emp, spans from 2 to 15, and so on.

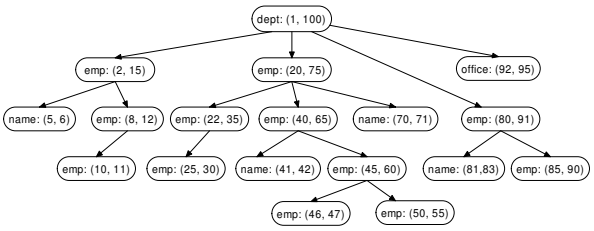


Figure 1. An example XML document

There are other approaches to numbering XML element nodes. One is the *durable* numbering scheme, where each element is numbered with a pair $(order, size)$ [18, 7]. For any two distinct elements u and v, u is an ancestor of v iff $u.order < v.order < u.order + u.size$. Dietz's numbering scheme uses tree traversal orders [12]. A tree node is assigned a pair of $(preorder, postorder)$ tree traversal orders. Element u is an ancestor of element v iff $u.preorder < v.preorder$ and $v.postorder < u.postorder$.

2.2. Structural joins

A structural join is to find all occurrences of structural relationship between two element sets. More for-

mally, given two input lists, $AList$ of potential ancestors (or parents) and $DList$ of potential descendants (or children), where each element in the lists is of the format: $(DocId, start, end, level)$, a structural join is to report all pairs (a_i, d_j), $a_i \in AList$ and $d_j \in DList$, such that (1) $a_i.DocId = d_j.DocId$; and (2) $a_i.start < d_j.start < a_i.end$. To retrieve only parent-child pairs, the condition $a_i.level = d_j.level - 1$ is also required.

XML query processing is dedicated to tree pattern matching while structural joins are considered as a core operation in optimizing XML queries [25, 18, 8]. Various techniques were proposed to leverage the power of widely available RDBMS [21, 14, 25, 24, 15] or to use native XML query engines [19]. In particular, [25] proposed a variant of the traditional merge join algorithm, called multi-predicate merge join (MPMGJN). However, it may perform a lot of unnecessary computation and I/O for matching structural relationship. Similarly, $\mathcal{EE}$-Join and $\mathcal{EA}$-Join in [18] may scan an element set multiple times.

The Stack-Tree-Desc join algorithm proposed in [22] improved the merge based structural join algorithms with stack mechanism. The basic idea is to take the two input lists, $AList$ and $DList$, both sorted on their $start$ values, and conceptually merge them. A stack is introduced to maintain ancestor elements that will be used later in the join. As such, only one sequential scan is performed on $AList$ and $DList$.

Chien et al, proposed a stack-based structural join algorithm that can utilize the B$^+$-tree indexes built on the $start$ attribute of the participating element sets [8]. An enhancement to the basic B$^+$-tree approach is to add *sibling* pointers based on the notion of "containment". They also presented a structural join algorithm that utilizes R-trees with *synchronized tree traversal* [6, 17].

3. XR-tree index: The structure

In this section, we present the structure of XR-tree, an index specially designed to index XML data for efficient structural joins. We first give definitions of some concepts, followed by the definition of XR-tree and some discussions.

3.1. Preliminary definitions

Definition 1 *Given a key k and an element with region $E_i(s_i, e_i)$* [1]*, E_i is said to be* stabbed *by k, or k* stabs *E_i, if $s_i \leq k \leq e_i$. Given a set of ordered keys, $k_j (0 \leq j < n)$, where $k_x < k_y$ if $x < y$, and an element $E_i(s_i, e_i)$, E_i is said to be* primarily stabbed *by k_j, or k_j* primarily stabs *E_i, if (1) $s_i \leq k_j \leq e_i$, and (2) for all $l, l < j$, $k_l < s_i$, that is, k_j is the smallest key that stabs E_i.*

[1] Element and region will be used interchangeably hereafter.

Definition 2 *Given a set of ordered keys, $k_j (0 \leq j < n)$, where $k_x < k_y$ if $x < y$, and a set of elements $\mathcal{E} = \bigcup_i (s_i, e_i)$, the* stab list *of a key k_j is the list of elements in $\mathcal{E}$ that are stabbed by k_j, denoted as SL_j or $SL_{(k_j)}$. The* primary stab list *of a key k_j is the list of elements in $\mathcal{E}$ that are primarily stabbed by k_j, denoted as PSL_j or $PSL_{(k_j)}$.*

The stabbing and primarily stabbing relationships can be illustrated with Figure 2. There are five keys, $k_0 < k_1 < k_2 < k_3 < k_4$, and seven regions (s_i, e_i), $0 \leq i < 7$. The lists of regions stabbed by k_j are $SL_0 = \{(s_0, e_0), (s_1, e_1), (s_2, e_2)\}$, $SL_1 = \{(s_0, e_0), (s_3, e_3)\}$, $SL_2 = \{(s_0, e_0), (s_4, e_4), (s_5, e_5)\}$, $SL_3 = \{(s_0, e_0), (s_4, e_4)\}$, and $SL_4 = \{(s_6, e_6)\}$, respectively. The primary stab lists of the keys are $PSL_0 = \{(s_0, e_0), (s_1, e_1), (s_2, e_2)\}$, $PSL_1 = \{(s_3, e_3)\}$, $PSL_2 = \{(s_4, e_4), (s_5, e_5)\}$, $PSL_3 = \emptyset$ and $PSL_4 = \{(s_6, e_6)\}$ respectively.

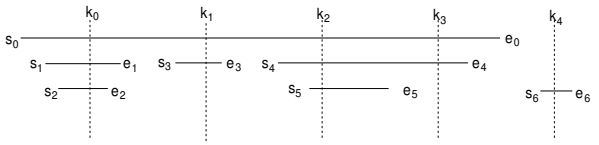


Figure 2. Keys and stabbed regions

It is easy to see that, strict ancestor-descendant relationship holds between each pair of neighboring elements in a (primary) stab list. Take k_0 and PSL_0 as an example. (s_0, e_0) is an ancestor of (s_1, e_1), which is an ancestor of (s_2, e_2). In other words, the first element in a $PSL_{(k)}$ is the ancestor of all other elements in $PSL_{(k)}$ and its region *covers* all other regions in $PSL_{(k)}$.

Definition 3 *The start and end positions, ps_j, pe_j, of key k_j with primary stab list PSL_j are defined the start and end positions of the first element of PSL_j when $PSL_j \neq \emptyset$, and (nil, nil) if $PSL_j = \emptyset$.*

In Figure 2, we have $(ps_0, pe_0) = (s_0, e_0)$, $(ps_1, pe_1) = (s_3, e_3)$, $(ps_2, pe_2) = (s_4, e_4)$, $(ps_3, pe_3) = (nil, nil)$, and $(ps_4, pe_4) = (s_6, e_6)$.

3.2. XR-tree: Its structure

With the above introduction, we proceed to define the structure of an XR-tree index.

Definition 4 *An XR-tree for a set of region-encoded XML elements is a tree with the following properties:*

1. *An XR-tree is a balanced tree.*
2. *An internal node contains m key entries in the form of (k_i, ps_i, pe_i), with $k_0 < k_1 < \cdots < k_{m-1}$, and $d \leq m \leq 2d$, where d is the degree of the XR-tree.*

3. *An internal node with m keys also contains $m + 1$ pointers $p_j, (0 \leq j \leq m)$, pointing to the nodes in the next level of the tree, such that all keys in the node pointed by p_i are less than k_i, and all keys in the node pointed by p_{i+1} are greater than or equal to k_i, respectively.*

4. *An internal node n is associated with a stab list, $SL(n)$, which holds all elements E_i, such that E_i is stabbed by at least one key in n but not stabbed by any key of any ancestor of n. Each element in $SL(n)$ is in the form of $(s, e, pointer)$, where (s, e) is the region of the element and $pointer$ points to the data entry of the element.*

5. *SL_j, PSL_j for the set of all keys k_j in an internal node n are defined on the list $SL(n)$ by Definition 2. Each pair of (ps_j, pe_j) of k_j is defined by Definition 3.*

6. *Leaf nodes contain element entries, in the form of $(s, e, InStabList, pointer)$, where (s, e) is the region of the element, and s is the index key. $InStabList$ is a flag indicating whether the element is included in any stab list of internal nodes, and $pointer$ points to the data entry of the element.*

7. *Leaf nodes are linked from left to right.*

By definition, an XR-tree is essentially a B$^+$-tree with a complex index key entry and extra stab lists associated with its internal nodes. We will see later that it is such a complex key and the stab lists that enable XR-tree indexes to support efficient structural joins.

In relational systems, indexes are usually built on an attribute or a set of attributes in a table. For XML documents, we can build indexes on sets of elements/attributes defined by certain predicates. Figure 3 shows the XR-tree for the set of emp elements in the example document in Figure 1. The $pointer$ field is omitted for clarity.

In Figure 3, the left internal node has an empty stab list, while the root and the right internal node have 2 and 3 regions in their stab lists respectively. Note that, although the region $(20, 75)$ is stabbed by key 46, it is not put in the stab list of the right internal node because $(20, 75)$ is already stabbed by key 24 in the root. By definition, we only include a region in the stab list of the top-most node. For key 19, $ps_{(19)}$ and $pe_{(19)}$ are set to nil because $PSL_{(19)} = \emptyset$. For key 46, $PSL_{(46)} = \{(40, 65), (45, 60), (46, 47)\}$, and $ps_{(46)} = 40, pe_{(46)} = 65$. Since $PSL_{(79)} = \emptyset$, $ps_{(79)} = pe_{(79)} = nil$.

Astute readers may find that some keys in internal nodes do not appear as $start$ positions (or keys) of elements in the leaf nodes. In fact, index keys are not necessarily $start$ positions of certain elements. Ideally, keys should be chosen to minimize the size of stab lists. For example, if we choose 80, the $start$ of element $(80, 91)$, instead of 79, as the key in the right internal node, we will have one more stabbed

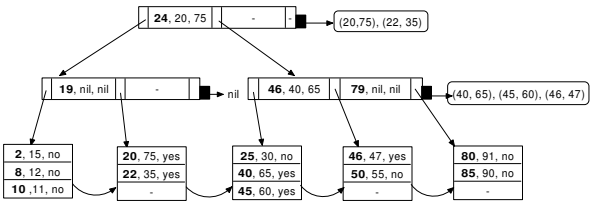


Figure 3. The XR-tree for the element set *emp* shown in Figure 1

element. Such optimization can simply be done by using a value (as the internal key) that is smaller than the keys at the right branch, if possible. We have to use key 46, which is equal to the smallest key at the right branch of the key, since 45 is the $start$ position of another region.

3.3. More on stab lists

An XR-tree is basically a B$^+$-tree augmented with stab lists of internal nodes. One expected concern is the space overhead of stab lists, hence the manipulation overhead caused by them. In this section, we discuss these issues.

Sizes of stab lists

According to the property of XR-tree, each element region can only be included, if any, in one stab list of all internal nodes. Therefore, the total elements in all stab lists will not exceed the total elements indexed. Such an upper bound, however, is hardly reached. It is obvious that each index key can only stab at most h_d elements, where h_d is the maximum number of nestings of element nodes indexed. Therefore, the maximum number of pages for a stab list is $S_{max} = \frac{h_d B_I f_{max}}{B_S f_{min}}$, where B_I is the maximum number of entries in an internal node, B_S is the maximum number of tuples a stab list page can hold and f_{min}, f_{max} are the minimum and maximum page fill ratios, respectively.

If we assume the size of a key entry plus a pointer is the same as the size of an entry in the stab list, f_{min} and f_{max} be 0.5 and 1.0, respectively, we have $S_{max} = 2h_d$ (pages). Therefore, with reasonable levels of nesting, we expect that the number of pages for the stab list attached to an internal node is small, ranging from zero to a few pages.

Since it is rather difficult to estimate the size of stab lists more precisely, we conducted some experiments using data from two benchmarks, XMach [2] and XMark [20]. We selected various sets of elements by changing the selection predicates, built indexes on those element sets, and counted the number of stab list pages. The results show that, for XR-trees of real-world data, the average size as well as the maximum size of stab lists is about several disk pages, and the total size of stab lists is much smaller than the whole set of elements indexed (less than 10% of leaf pages for highly nested data sets with the number of nestings larger than 10).

Stab list access cost

The formula, $S_{max} = 2h_d$, implies that the stab list of an internal node could span a few pages or even tens of pages in cases where an element set is extremely highly nested. The cost of access to a stab list becomes a concern for such extreme cases. For example, we often need to locate the PSL_j for a given key k_j when searching in or updating an XR-tree. It is undesirable if we need to scan a lot of pages before the PSL_j is located in the stab list. The problem, however, can be tackled with a *ps directory page* that maps each ps_j to the location of PSL_j. Figure 4 shows an example of a ps directory page that maps ps fields of five index keys to the beginning of their PSL's.

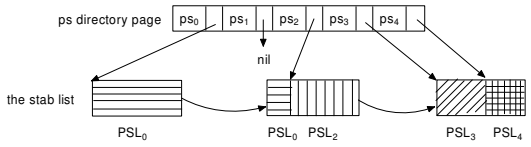


Figure 4. Mapping ps_j to PSL_j in a stab list

As shown in Figure 4, a ps directory page contains one entry for each index key. The entry is of the format: $(ps_j, pointer)$, where ps_j is the ps field of key k_j and $pointer$ points to the head of PSL_j in the stab list. The $pointer$ is set to nil if $PSL_j = \emptyset$, as for ps_1. Since the size of an entry is smaller than the size of a key entry in an internal node and the numbers of entries in an internal node and its ps directory page are the same, one ps directory page is always sufficient to map all ps fields to their PSL's. Note that the ps directory page is necessary only when a stab list spans more than one disk page. It is easy to see that locating PSL_j of key k_j takes only 1 or 2 disk I/O's.

4 Updating

In this section, we present algorithms for insertion and deletion in XR-trees. Update in XR-trees should not be confused with update in source XML documents for which XR-tree indexes are built. We need to update an XR-tree only when the related part of the source data is updated. The problem of updating XML is still an open issue and detailed discussion is out of the scope of this paper. Interested readers may refer to [23].

4.1. Insertion

Inserting new elements is similar to insertion in a B$^+$-tree in that new elements are added to leaf pages, overflowing pages are split, and splits propagate up the tree [10]. Algorithm 1 lists the routine for insertion.

We now go through Algorithm 1 and review some features related to stab list maintenance during insertion.

- In step I1, when navigating down to a leaf page, E should be inserted into the stab list of the highest internal node that stabs it, if any, with its $InStabList$ flag set to yes.

- In case of overflow (step I22 and I32), when splitting an internal node, we also need to split its stab list. An illustrative example is shown in Figure 5(a). In fact, the split cost is independent of the size of the stab list because we only need to access the page which holds the splitting point. No other pages of the stab list need to be touched.

- After split, we propose not only a new key k' to the upper level, but also the set of elements stabbed by k', denoted as $StabSet'$, as demonstrated in Figure 5(b), where k' is inserted after k_2, which is selected as the key to be given up. Note that $StabSet'$ contains all elements from $SL(I)$ and $SL(I_{new})$ that are stabbed by k' and they are removed from $SL(I)$ and $SL(I_{new})$ at the same time. If the split page is a leaf page, we retrieve elements in L and L_{new} that are newly stabbed (i.e. $InStabList$ = no) and turn their flags to yes.

Algorithm 1 Insertion

Input: A new element, $E = (s, e, pointer)$, to be inserted.

I1 **[Find a leaf page for insertion]** Navigate down to a proper leaf page for insertion. Insert E into the stab list of the highest internal node that stabs it, if any.

I2 **[Insert E into the leaf page L]**

I21 If L has room for another element, insert E and return.

I22 Otherwise, split L by moving second half entries to a new leaf page L_{new}. Insert E into the correct leaf page. Give up a new key entry $(k', pointer')$, together with its $StabSet'$.

I3 **[Insert $(k', pointer', StabSet')$ into internal node I]**

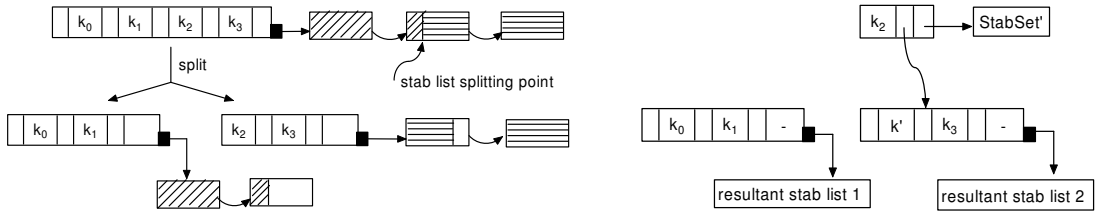
I31 If I has enough room, insert the new entry and its $StabSet'$ into I.

I32 Otherwise, split I by moving the second half entries, together with their primary stab lists, to the new node I_{new}. Insert the new entry and its $StabSet'$ into the proper internal node. Give up a new key entry $(k', pointer')$, together with its $StabSet'$. Repeat step I3, or go to I4 if I is the root node.

I4 **[Grow the XR-tree taller]** If the node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

I/O cost analysis The insertion I/O cost for XR-trees is the same as for B$^+$-trees, i.e. $O(\log_F N)$, except for additional cost for maintaining stab lists of internal nodes.

We now give amortized I/O cost for stab list maintenance for each insertion. The basic idea is to consider the worst case total I/O cost for inserting N elements into an empty XR-tree and the amortized cost is the total cost divided by



(a) Splitting a node

(b) Giving up a new key with StabSet'

Figure 5. Splitting an internal node and giving up a new key and its StabSet'

N. Here, the *worst case* is that all elements in leaf pages are stabbed by internal nodes.

As we know, each stabbed element at height h is either given up from the lower level node at height $h-1$ (step I22 and I32) or directly inserted during insertion(step I1). Let us assume that all elements in stab lists are given up from lower level nodes, which will give a looser upper bound of I/O cost. It is obvious that a stabbed element at height h is displaced (or given up) for h times all the way from the leaf page to its current position during node split. Let the cost of one displacement (i.e from height $h-1$ to h) of an element be C_{DP} (see section 4.3). A stabbed element at height h incurs at most $C_{DP} \cdot h$ displacement cost. The amortized displacement I/O cost $\overline{C_{DP}}$ for each insertion is the total cost of displacements divided by the total number of insertions, i.e. $\overline{C_{DP}} = \sum_{h=1}^{h=H} N_h C_{DP} \cdot h/N$, where N_h is the total number of stabbed elements at height h. Since each internal node can stab at most $h_d B_I f$ elements, the maximum total number of elements in the stab lists of internal nodes with height > 1 can be approximated by $\frac{h_d}{B_L B_I f^2} N$, where f is the average page fill ratio and h_d, B_I, B_L were introduced in section 3.3. Since $\frac{h_d}{B_L B_I f^2} \ll 1$, it implies that most stabbed elements are at height 1 in the worst case where *all* elements are stabbed. Thus, we have $\overline{C_{DP}} \approx N_1 C_{DP}/N \leq C_{DP}$.

Theorem 1 *The amortized I/O cost for inserting a new element into an XR-tree is $O(\log_F N + C_{DP})$, where N is the number of elements indexed, F is the fanout of the XR-tree, C_{DP} is the cost for one displacement of a stabbed element, or the cost for deleting an element from a stab list and then inserting it into another stab list.*

4.2. Deletion

Deleting elements from an XR-tree is similar to deletion in a B$^+$-tree in that elements are deleted from leaf pages, redistribution or merging occurs if leaf pages underflow. Algorithm 2 shows a sketch of the XR-tree deletion algorithm. Similarly, we need to take special care of stab list maintenance for internal nodes. We summarize the new features below:

- When navigating down, we need to remove E from the internal node if its stab list contains E (step D1).

Algorithm 2 Deletion

Input: An element, $E = (s, e, pointer)$, to be deleted.

D1 **[Locate E in leaf page]** Locate the leaf page containing E. Delete E from the internal node I if its stab list contains E.

D2 **[Delete E from leaf page L]**

- D21 Delete E from L. If L remains at least half full, return.
- D22 Otherwise, let S be a sibling of L. If S has extra entries, redistribute entries between L and S and update their parent entry.
- D23 Otherwise, merge S and L. Propagate the deletion up the tree with the key of their parent entry.

D3 **[Delete the entry from an internal node I]**

- D31 Suppose entry j is to be deleted. Delete entry j from I and "reinsert" elements in $SL(I)$ that are no longer stabbed by I. If I remains at least half full, return.
- D32 Otherwise, let S be a sibling of I. If S has extra entries, redistribute entries between L and S. Update their parent entry properly and return.
- D33 Otherwise, merge L and S, and their stab lists. Propagate the deletion up the tree with the key of their parent entry.

D4 **[Shorten the tree]** If the root has only one child after the deletion, make the child as the new root.

- Deleting an index entry from I will possibly cause some elements in $SL(I)$ no longer stabbed by I. For each such element E', we need to "reinsert" it into the highest internal node that stabs it. If no such internal node exists, we set the $InStabList$ flag of E' to no.

- After redistribution between two internal nodes, we need to update the entry with key k (that separates them) in the parent node P with a new key k'. $SL_{(k')}$ should be removed from the two internal nodes and inserted into $SL(P)$. At the same time, some elements in $SL(P)$ might be no longer stabbed by P after k is replaced with k'. This can be handled similarly as the entry deletion case. Redistribution between two leaf pages follows similar procedure.

- When merging two internal nodes, say, from I to its left sibling S, we also need to merge $SL(I)$ to $SL(S)$. This can simply be done by linking $SL(I)$ to $SL(S)$.

I/O cost analysis As the insertion operation, deletion in an XR-tree incurs additional cost for stab list maintenance due to the displacement of stabbed elements. In the interest of space, we omit the detailed analysis.

Theorem 2 *The amortized I/O cost for deleting an element from an XR-tree is $O(\log_F N + 3 \cdot C_{DP})$, where N is the number of elements indexed, F is the fanout of the XR-tree, C_{DP} is the cost for the displacement of an element from one stab list to another.*

4.3. Cost for manipulating stab lists

As has been shown, updating an XR-tree incurs a little additional I/O for stab list maintenance. We are particularly interested in the I/O cost for deleting an element from a stab list, C_{SD}, and inserting an element into a stab list, C_{SI}, because they are factors of amortized update cost. Note that $C_{DP} = C_{SD} + C_{SI}$.

As a matter of fact, both C_{SD} and C_{SI} will be no more than 2 or 3 disk I/O's, based on the analysis in section 3.3, where we proposed the usage of a ps directory page for stab lists that span more than one page. (ps, pe) fields of an internal node can be updated without additional I/O cost after we insert an element into its stab list or delete an element from its stab list.

5 Processing structural joins with XR-trees

In this section, we describe how XR-trees can be used to support efficient structural joins in XML documents. We first describe two basic operations, followed by a structural join algorithm for XR-tree indexed data. In our discussion, we assume that the XML element sets are indexed using XR-trees.

5.1. Basic operations: Searching for descendants and ancestors

A structural join finds matching ancestor-descendant, or parent-child pairs between two sets of elements. To process such joins, we identify two basic operations:

1. FindDescendants: Given an element E_a, find all its descendants in an element set indexed by an XR-tree.
2. FindAncestors: Given an element E_d, find all its ancestors in an element set indexed by an XR-tree.

5.1.1 Searching for descendants

Since XML element nodes are strictly nested, for a given element (s_a, e_a), finding all its descendants is to find all elements E_i such that $s_a < E_i.start < e_a$. It is just a simple range query over the $start$ position of elements, on which they are indexed in the backbone of XR-trees. The algorithm is rather straightforward as outlined in Algorithm 3. Note that there is no need to access stab lists when searching for descendants. It is obvious that Algorithm 3 correctly retrieves all descendants of element (s_a, e_a) in T. The following theorem gives the I/O cost for the algorithm FindDescendants (proof omitted in the interest of space):

Algorithm 3 FindDescendants

Description: find all descendant elements of $E_A = (s_a, e_a)$ in XR-tree T.

```
1: node := T.root;
2: while node is not a leaf page do
3:    find the largest key k_i in node, such that k_i ≤ s_a;
4:    if found, let node := k_i.rightChild; otherwise, let node :=
      k_0.leftChild;
5: end while
6: stop := FALSE;
7: while not stop do
8:    for all entries E_i in node do
9:       if s_a < E_i.start < e_a, output E_i;
10:      if E_i.start > e_a, let stop := TRUE;
11:   end for
12:   node := node.next;
13: end while
```

Theorem 3 *The operation* FindDescendants *over an XR-tree can be evaluated with optimal worst case I/O cost: $O(\log_F N + R/B)$, where N is the number of elements indexed, F is the fanout of the XR-tree, R is the output size, B is the average number of element entries in each leaf page.*

5.1.2 Searching for ancestors

Since XML elements are strictly nested, for a given element (s_d, e_d), finding all its ancestors is to find all elements E_i such that $E_i.start < s_d < E_i.end$. In other words, it is to search for all elements stabbed by s_d. Note that elements stabbed by s_d could be scattered in any leaf page left to the leaf page on the search path of s_d. Sequential search of leaf pages could be costly, which is exactly the motivation for XR-trees. Our basic idea is, during the navigation from the root to the leaf page, we search the stab lists of internal nodes to collect elements stabbed by s_d. After reaching the leaf page, we output those elements stabbed by s_d but not included in the stab lists of internal nodes. The process is outlined in Algorithm 4.

The algorithm for finding stabbed elements in a stab list is shown in Algorithm 5. Assume that s_d falls in $[k_i, k_{i+1})$. It is clear that s_d cannot stab any element in PSL_j, where $j > i+1$ because all such elements have their $start$ values larger than k_{i+1}, i.e. these elements are "behind" s_d. Therefore, we only need to check PSL_c of k_c, where $c <= i+1$.

Since element nodes in a PSL are strictly nested, if s_d stabs some element (s_j, e_j) in a PSL, then it must stab all its ancestors in the PSL. In other words, all elements

Algorithm 4 FindAncestors

Description: Find all ancestors of $E_D = (s_d, e_d)$ in XR-tree T.

S0 Let $node$ be the root of T;
S1 Search non-leaf pages
While $node$ is not a leaf page **do**
S11 Retrieve all elements stabbed by s_d in its stab list, by calling *SearchStabList* (Algorithm 5).
S12 Find the largest key k_i, such that $k_i \leq s_d$.
S13 If found, let $node$ be $k_i.rightChild$; otherwise, let $node$ be the left child of the first index entry.
S2 [Search within the leaf page] Search from the first element of $node$ to output elements that are stabbed by s_d but with $InStabList$ flags being no, until an element whose $start$ position is greater than s_d is encountered.

Algorithm 5 SearchStabList

Description: Search the stab list of an internal node I for all elements stabbed by s_d.

1: let k_i be the key in I, such that $k_i \leq s_d < k_{i+1}$;
2: **for** $c = i + 1$ to 0 **do**
3: **if** $ps_c \neq nil$ and $ps_c < s_d < pe_c$ **then**
4: Scan PSL_c and output the scanned elements until an element not stabbed by s_d is encountered;
5: **end if**
6: **end for**

stabbed by s_d are clustered at the beginning part of a PSL. Thus, when searching for stabbed elements in a PSL, we just scan the PSL and stop as soon as the current element is not stabbed by s_d, which explains line 4 in the algorithm.

The I/O efficiency of *SearchStabList* is guaranteed by the fact that, when checking some PSL_c for matches, we do not need to access the stab list until we are guaranteed to have at least one match (when s_d stabs PSL_c's first element stored in the index entry, i.e. $ps_c < s_d < pe_c$). Furthermore, with the *ps directory page*, it is possible to access the PSL of any key in 1-2 disk I/Os. The following theorem gives the I/O cost for retrieving the ancestors of an element.

Theorem 4 *The operation* FindAncestors *takes* $O(\log_F N + R)$ *worst case I/O cost, where* N *is the number of elements indexed,* F *is the fanout of the XR-tree,* R *is the output size.*

The correctness of the algorithm FindAncestors is assured by the following lemma (in the interest of space, we omit the proof):

Lemma 1 *Let* T *be an XR-tree of height* H*,* p_s *be a query point. Let* $I_{H-1} \rightarrow I_{H-2} \rightarrow \cdots \rightarrow I_1 \rightarrow L_0$ *be the path for* p_s *to navigate from* I_{H-1}*, the root of* T*, down to a leaf page,* L_0*. Let* R *be the set of elements stabbed by* p_s *in* T*. Then, for each element* $E \in R$*, it must appear in* L_0 *or the stab list of some* I_i*, where* $i \in \{H-1, H-2, \cdots, 1\}$.

5.2. Structural joins on XR-tree indexed data

We are now ready to describe an algorithm for joining XR-tree indexed data. Assume A and D are two element sets. Both sets are indexed using XR-trees. Since the leaf pages are sorted on the $start$ positions of the elements, join can be processed just like merge-join in relational systems. That is, using two pointers to move down the lists to find matches. Different from the typical merge-join algorithm, we can effectively skip elements that do not have matches with the two operations described previously. For any element, we can retrieve its ancestors or descendants using XR-trees, thus there is no need to touch those elements that are not ancestors or descendants. We also maintain a stack to cache the ancestors of the current descendant element. The algorithm is outlined in Algorithm 6.

Algorithm 6 Stack-based Structural Joins with XR-trees

Input:
A is the ancestor set and D is the descendant set.

1: $CurA := First(A)$;
2: $CurD := First(D)$;
3: $stack := \emptyset$;
4: **while** $CurA \neq EndOf(A)$ and $CurD \neq EndOf(D)$ **do**
5: **if** $stack \neq \emptyset$ **then**
6: pop all elements that are not ancestors of $CurD$;
7: **end if**
8: **if** $CurA.start < CurD.start$ **then**
9: A_d := FindAncestors(A, $CurD.start$);
10: for each $a_j \in A_d$, if $a_j \notin stack$, push it on the stack;
11: output pairs ($a \in stack$, CurD);
12: $CurA$:= first element in A whose $start > CurD.start$;
13: $CurD$:= next element in D after $CurD$;
14: **else**
15: **if** $stack \neq \emptyset$ **then**
16: output pairs $(a \in stack, CurD)$;
17: $CurD$:= next element in D after $CurD$;
18: **else**
19: $CurD$:= first element in D whose $start > CurA.start$;
20: **end if**
21: **end if**
22: **end while**

The algorithm keeps two pointers, $CurA$ and $CurD$, which point to the current elements to be checked in A and D, respectively. At the beginning, they are set to the first elements of A and D, respectively (line 1-2). We also maintain a $stack$ that holds the elements in A, such that each element in $stack$ is a descendant of the element below it and all elements in $stack$ are ancestors of $CurD'$, where $CurD'$ is the element before $CurD$ in D. In other words, $stack$ caches the ancestors that could possibly join with $CurD$. By definition, $stack$ is initially empty (line 3) because, trivially, there is no element before $CurD$.

The basic idea of the algorithm is simple. At each loop,

we try to skip ancestors or descendants without matches, based on current positions of $CurA$ and $CurD$. If $CurA$ is before $CurD$, we try to skip ancestors based on $CurD$; Otherwise, we skip descendants based on $CurA$. The process is repeated until one of the lists is exhausted.

Now we explain Algorithm 6 in details. By precondition, $stack$ keeps the ancestors of an element $CurD'$ preceding $CurD$. Since $CurD.start > CurD'.start$, some top elements in $stack$ may not be ancestors of $CurD$. Such elements cannot be ancestors of any element after $CurD$. Therefore, we pop these elements (line 5-7). The case where $CurA$ is before $CurD$ is coped with from line 9 to 13. We retrieve all ancestors of $CurD$, push them on the stack, output all matched pairs for $CurD$ and then move forward both $CurA$ and $CurD$. Since $stack$ is possibly keeping some of $CurD$'s ancestors, we should not push these ancestors on $stack$ (line 9-10). In the implementation, we used a variation of FindAncestors that can return $CurD$'s ancestors after the stack top. Line 15-19 deal with the case where $CurA$ is behind $CurD$. As a matter of fact, it is impossible to skip elements in D if the stack is not empty because the in-stack ancestors could possibly join with descendants between $CurD.start$ and $CurA.start$. As a result, we proceed $CurD$ by one if $stack$ is not empty (line 15-17). Otherwise, we move $CurD$ to the element right after $CurA$ (line 19). This can be done with a variation of FindDescendants, i.e. an open-ended range query $(CurA.start, +\infty)$, to get the first element found.

5.3. Evaluating parent-child relationship

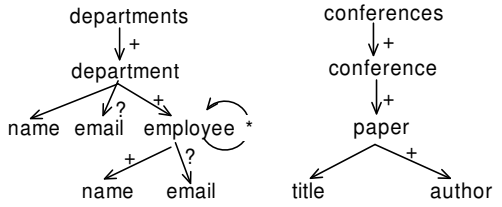
FindDescendants and FindAncestors can be easily extended to support parent-child relationship, i.e. FindChildren and FindParent (note that an element has at most one parent in the tree data model), by storing the $level$ attribute together with each element in leaf pages of XR-trees. The search process is similar, but we need to apply an additional condition: $ancestor.level = descendant.level - 1$. Similarly, FindChildren and FindParent can be used as primitives for a stack-based structural join algorithm evaluates parent-child relationship between two element sets.

6 A performance study

Comprehensive experiments were conducted to study the effectiveness of XR-tree indexing. In this section, we describe these experiments and present some of their results.

6.1. Experimental setup

We used synthetic data for our experiments in order to control the structure and join characteristics of XML documents[2]. Two DTDs, shown in Figure 6, were used to generate highly nested and less nested data sets respectively. In particular, the $Department$ DTD is the same as the one in [8]. About 90MB raw XML data was generated for each DTD using the IBM XML data generator with default parameters [11].



(a) Department DTD (b) Conference DTD

Figure 6. The DTDs for synthetic data

Experiments were performed to study the comparative performance of structural joins on non-indexed, B^+-tree indexed, and XR-tree indexed XML data. Table 1 summarizes the join algorithms for which the results are presented. We do not show the results for the variations of *B+*, namely B+sp and B+psp, because they have similar behavior as that of *B+*. We did not test R^*-tree based algorithms because they have been shown in [8] to be less robust than the *B+* algorithm.

Table 1. Notations for algorithms

Notation	Represented Algorithm
no-index	Merge-join (Stack-Tree-Desc [22])
B+	B^+-tree indices (Anc_Des_B+ [8])
XR-stack	The XR-tree based algorithm

The joins tested used *employee* vs. *name* (Department DTD) and *paper* vs. *author* (Conference DTD) element sets as the base element sets. To have a fair comparison with the work in [8], we adopted similar methodologies to generate other element sets with different joining characteristics from the base element sets.

We evaluated the performance of different join algorithms using two performance indicators, *number of elements scanned*, and *elapsed time*.

- *Number of elements scanned:* This metrics, the total number of elements scanned during a join, evaluates the capability to skip the elements that do not produce join results for a structural join algorithm.

- *Elapsed time:* It is used to investigate the overall performance of different algorithms.

[2]Experiments were also conducted with real-world XML data sets, such as XMark, DBLP, etc. We observed similar performance behaviors as synthetic data sets, according to nesting properties.

The testing system was built on top of our experimental database system, which includes storage manager, buffer pool manger, B$^+$-tree and XR-tree index modules. All the experiments were performed on a Pentium IV 1.60GHz PC with 256M RAM, 20G hard disk, running Windows XP. The storage manager directly accesses physical disk drives with direct I/O functionalities provided by Windows XP. We ran all the algorithms with varying buffer pool sizes and found that their performance was not essentially affected. The rationale is twofold: (1) All algorithms are stack-based and they require to scan the data at most one; (2) Probing of indexes (both B$^+$-tree and XR-tree) are ordered, thus, index pages are accessed at most once. We will discuss this in more details later. All the experimental results presented below were obtained with a fixed buffer pool size: 100 pages.

6.2. Varying join selectivity on ancestors

The objective of this set of experiments is to study the capabilities of various algorithms to skip ancestor elements. During the experiments, we kept the percentage of descendants that match at least one ancestor high (99%) and varied the selectivity on ancestors, i.e., the percentage of ancestors that have descendants. For this purpose, we can start with two lists of ancestors and descendants, and then effectively remove certain elements from the descendant list so that the desired selectivity on ancestors can be obtained.

Table 2 shows the total number of elements scanned (in thousand) for various algorithms. *NIDX* and *XR* are short for *no-index* and *XR-stack* algorithms respectively.

Table 2. Number of elements scanned (in thousand) when 99% of descendants join with varying proportion of ancestors

(a) *employee* vs. *name*

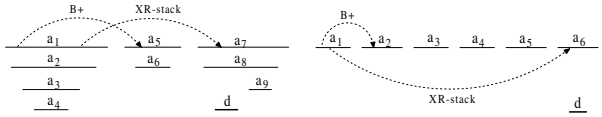
Join-A	NIDX	B+	XR
90%	1609	1547	1536
70%	1395	1207	1195
55%	1234	953	939
40%	1073	699	683
25%	913	444	427
15%	806	275	256
5%	698	105	85
1%	655	37	17

(b) *paper* vs. *author*

NIDX	B+	XR
1409	1409	1358
1208	1208	1057
1057	1057	830
906	906	604
755	755	377
654	654	227
554	554	75
513	513	15

It can be seen from Table 2 that the *XR-stack* algorithm is able to complete the join by scanning the least number of elements compared to the other algorithms. The benefit gets more obvious when the join selectivity on the ancestor set becomes lower. The *B+* algorithm is effective in skipping ancestor nodes for highly nested ancestor data (Table 2(a)) but it could be as less efficient as *no-index* for less nested ancestor data (Table 2(b)).

We further explain such ancestor-skipping capabilities of *B+* and *XR-stack* with two examples shown in Figure 7, one for highly nested ancestor data and the other for less nested ancestor data.



(a) Highly nested (b) Less nested

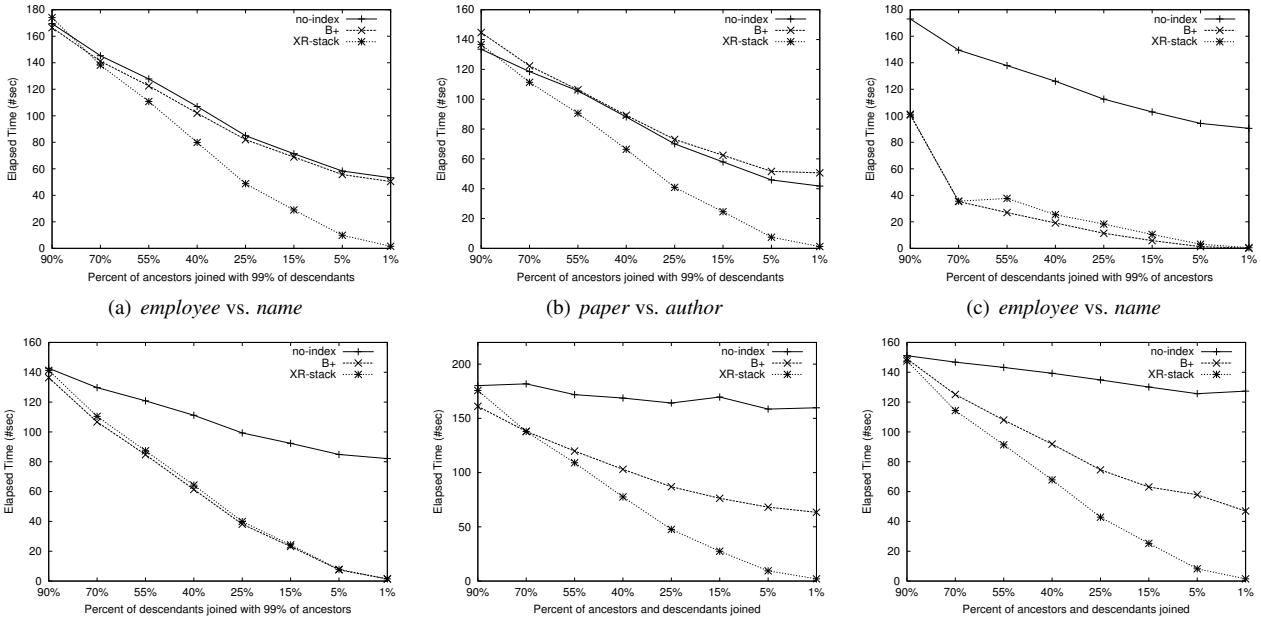
Figure 7. The ancestor-skipping capabilities of B+ and XR-stack algorithms for ancestors with different nesting properties

In both examples, the current examining ancestor and descendant elements are a_1 and d, respectively. The relationship between a_1 and d is $a_1.start < d.start$, i.e. a_1 is before d. Furthermore, the first ancestor of d is a_7 in Figure 7(a) and a_6 in Figure 7(b), respectively. Therefore, there is no need to scan elements between a_1 and the first ancestor, and those elements can be skipped. *no-index* always scans a_2, i.e. the one next to a_1. The next ancestors to be scanned for other two algorithms are shown by the dashed arrows. Since d is not a descendant of a_1, d cannot be a descendant of any descendant of a_1, based on the strictly nested property of XML data. The *B+* algorithm makes use of this property to skip all descendant elements of a_1 by locating the element in A having the smallest $start$ value that is larger than $a_1.end$. Therefore, in *B+* algorithm, the next elements to be examined are a_5 and a_2 in the two examples respectively. For highly nested data, a reasonable number of ancestors can be skipped, e.g. a_2, a_3 and a_4 in Figure 7(a). But for less/no nested data, the B+ algorithm could be as less efficient as *no-index*. In brief, the *B+* algorithm cannot take full advantage of the place of the current descendant to decide where the next candidate ancestor resides [8]. On the other hand, the *XR-stack* algorithm can directly locate ancestors of d by issuing a FindAncestors query against the XR-tree on A, as shown in Figure 7.

We also measured the CPU time, the number of I/O's and the total elapsed time for each run. The results show that the total elapsed time is dominated by the I/O's performed, more specifically, the number of page misses. Figure 8(a) and 8(b) display the elapsed time for various algorithms.

As can be observed from the figures, *XR-stack* has the best overall performance. Its advantage margin gets larger with the decrease of join selectivity on the ancestor set.

Interestingly, we find that even when most of ancestors and descendants participate in the join, i.e. when only few elements can be skipped, despite of the possible overhead of index probing, *XR-stack* (and also *B+*) performed no worse than *no-index*. This can be best explained as follows: since both element sets (A and D) are sorted, the keys used to probe indexes are ordered. The consequence of ordered probing is that most tree probes cause no page misses and the index pages only need to be scanned at most once. Since



(a) *employee* vs. *name* (b) *paper* vs. *author* (c) *employee* vs. *name*

(d) *paper* vs. *author* (e) *employee* vs. *name* (f) *paper* vs. *author*

Figure 8. Elapsed time (in second) for different join selectivity. (a)(b): 99% of descendants join with varying proportion of ancestors; (c)(d): 99% of ancestors join with varying proportion of descendants; (e)(f): varying proportion of ancestors and descendants are joined.

the number of index pages is much less than the number of leaf pages that hold element entries, the overhead of loading index pages does not significantly affect the performance.

It is worth noting that the number of elements scanned is unnecessarily proportional to the elapsed time. The rationale is that while each element scan causes a pin to a buffer page, the elapsed time is dominated by page misses. Consecutive element scans on the same buffer page cause almost no additional running time. For example, although the *B+* algorithm managed to skip much more elements than *no-index* (Table 2(a)), it failed to avoid more disk page scans. As a result, its elapsed time is similar to *no-index* as shown in Figure 8(a).

6.3. Varying join selectivity on descendants

In the second group of experiments, we kept the join selectivity on ancestors high (99%) and varied the join selectivity on descendants. Table 3 shows the total number of elements scanned (in thousand) for various algorithms.

In terms of skipping descendants, *XR-stack* is as effective as *B+*, regardless of the nesting characteristics of the joining element sets (Table 3). This can be explained by the fact that skipping descendants requires the ability to find descendants for a given element, which is the same in XR-tree indexing and B^+-tree indexing.

Figure 8(c) and 8(d) show the overall performance of the algorithms tested. Compared to *B+*, *XR-stack* performed a bit worse. This attributes to the higher overhead of XR-tree indexing than B^+-tree indexing caused by two additional fields (ps, pe) in key entries, hence more index pages.

Table 3. Number of element scanned (in thousand) when 99% of ancestors join with varying proportion of descendants

(a) *employee* vs. *name*

Join-D	NIDX	B+	XR
90%	1657	1559	1550
70%	1527	1213	1206
55%	1429	953	947
40%	1332	693	689
25%	1234	433	430
15%	1169	260	258
5%	1104	87	86
1%	1078	17	17

(b) *paper* vs. *author*

NIDX	B+	XR
1459	1359	1359
1359	1057	1057
1283	830	830
1208	604	604
1132	377	377
1082	226	226
1032	75	75
1011	15	15

6.4. Varying join selectivity on both ancestors and descendants

In the last set of experiments, we varied the join selectivity on both ancestors and descendants. Among all the experiments, we show the results for the case where the join selectivity on the ancestor set and the descendant set varied together, staring from 90% down to 1%, with their sizes kept unchanged. This was done by effectively removing joined elements from the two sets and then filling in some dummy elements that do not join with any other elements.

The results for these experiments are shown in Figure 8(e) and 8(f).

The diversity of the three algorithms is best illustrated by this group of experiments, where there is potential to skip both ancestors and descendants. Since *no-index* always sequentially scans elements, it performs the worst. *B+* is able to skip descendants that do not participate in joins, but fails to effectively skip ancestors. Thus, it performs the second. *XR-stack* can fully explore the potential of skipping both ancestors and descendants based on the XR-trees built on joining element sets. Therefore, it provides the best performance among all.

7 Conclusions and future work

As we know, B-trees and their variants B^+-trees have been an unqualified success in supporting external dynamic 1-dimensional range searching in relational database systems [10], while R-trees [16] and R^*-trees [3] are successful in indexing high-dimensional data points. Despite the increasing popularity of XML, to the best of our knowledge, we have seen no index structures that specifically deal with strictly nested XML data.

In this paper, we proposed XR-tree, or XML Region Tree, which is a dynamic external memory index structure specially designed for such strictly nested XML data. XR-trees can support, for a given element E, retrieval of all its ancestors(or descendants) in an element set $\mathcal{E}$ indexed by an XR-tree with optimal worst case I/O cost. By factoring in such a unique feature of XR-trees, we devised a stack-based structural join algorithm: *XR-stack*. In a study that evaluated the performance of *XR-stack*, in comparison with the current state of the art, we showed that the *XR-stack* algorithm can most effectively avoid unnecessary element scans by skipping both ancestors and descendants that do not have matches.

This paper mainly focused on processing of *structural join*, which is a core operation for XML query processing. Regarding our future work, encouraged by the experimental results, we will be working on query evaluation strategies for complex XML queries (i.e. a combination of multiple structural joins) over XML data on which proper XR-tree indexes have been built.

References

[1] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, pages 560–569, 1996.

[2] T. Böhme and E. Rahm. XMach-1: A benchmark for XML data management. In *BTW*, 2001.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] M. D. Berg, M. V. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 1997.

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. In *W3C Working Draft 16 August 2002*, http://www.w3.org/TR/xquery/, 2002.

[6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-Trees. In *SIGMOD*, pages 237–246, 1993.

[7] S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Efficient complex query support for multiversion XML documents. In *EDBT*, pages 161–178, 2002.

[8] S.-Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, pages 263–274, 2002.

[9] J. Clark and S. DeRose. XML path language (XPath). In *W3C Recommendation 16 November 1999*, http://www.w3.org/TR/xpath, 1999.

[10] D. Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[11] IBM Corporation. XML data generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[12] P. F. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, pages 122–127, 1982.

[13] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. In *VLDB*, pages 1–12, 1990.

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

[15] T. Grust. Accelerating XPath location steps. In *SIGMOD*, pages 109–120, 2002.

[16] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[17] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB*, pages 396–405, 1997.

[18] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[19] J. McHugh and J. Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.

[20] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical report, CWI, Amsterdam, The Netherlands, 2001.

[21] J. Shanmugasundaram, K. Tufte, C. Zhang, H. Gang, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

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

[23] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, 2001.

[24] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[25] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.