An Evaluation of XML Indexes for Structural Join

Hanyu Li Mong Li Lee Wynne Hsu Chao Chen
School of Computing, National University of Singapore
{lihanyu, leeml, whsu, chenchao}@comp.nus.edu.sg

ABSTRACT

XML queries differ from relational queries in that the former are expressed as path expressions. The efficient handling of structural relationships has become a key factor in XML query processing. Many index-based solutions have been proposed for efficient structural join in XML queries. This work explores the state-of-the-art indexes, namely, B^+-tree, XB-tree and XR-tree, and analyzes how well they support XML structural joins. Experiment results indicate that all three indexes yield comparable performances for non-recursive XML data, while the XB-tree outperforms the rest for highly recursive XML data.

1. INTRODUCTION

The growth of XML repositories on the Web has led to much research on storing and indexing for efficient querying and updates of XML data. XML data are viewed as an ordered tree structure, and queries are specified using path expressions. The structural join is a core operation that evaluates containment relationships of the XML elements in the path expression query.

The state-of-the-art structural join solution, *Stack-Tree* [5], takes as input two ordered lists of elements that are involved in the join, and utilizes a sort-merge based algorithm to find all the pairs of elements that satisfy the containment relationship. By maintaining an in-memory stack, the two ordered lists are scanned once. This greatly improves the performance of structural join. However, the algorithm will incur unnecessary I/O cost for low selectivity queries since every element in the lists must be accessed before the join can be carried out.

A simple yet effective approach to skip the unnecessary data during the scanning process is to construct indexes on the input lists. These indexes aim to efficiently support functions such as *findDescendants* and *findAncestors* that are needed in structural joins. Major index-based solutions for structural joins are the B^+-tree [2], the XB-tree [1], and the XR-tree [3].

Contributions. This work presents a comprehensive comparative study of these index-based solutions. We implement the B^+-tree, the XR-tree and the XB-tree solutions, and perform experiments to evaluate their query and update costs, as well as their storage requirements.

The XB-tree is proposed primarily for evaluating *holistic twig joins* [1]. While the focus of the work in [1] is not on the index itself, nonetheless, the XB-tree has many nice properties that can be exploited for structural joins. We carry out an in-depth study of the XB-tree, and introduce the notion of a *valid path* that guarantees the efficiency of the multi-path *findAncestors* search in the XB-tree.

For the XR-tree, we observe that there is a tradeoff between the update cost and query performance. Thus, we also investigate two variants of the XR-tree: one that minimizes the update cost, and the other that minimizes the query and storage costs.

2. INDEX STRUCTURES

An example XML document for file systems is shown in Figure 1. The interval-based labelling scheme [4, 6] has been used to label the element nodes. The containment relationship between two XML elements can be quickly determined by the containment of their intervals. Figure 2 shows the most basic approach which constructs a B^+-tree on the start points of the directory element intervals [2].

The XR-tree [3] is essentially a B^+-tree that is built on the start points of the element intervals. Figure 3 shows the XR-tree that has been constructed for the directory elements in Figure 1. Every non-leaf node in the XR-tree is associated with a stab list. The stab list stores the intervals of element entries that can cover any key in the non-leaf node. To facilitate searching in the stab lists, each key in the non-leaf node is also associated with the first element interval in the primary stab list that contains the key. Note that besides storing an element e in the leaf node, the XR-tree also stores the element in the stab list of the top-most internal node that contains a key k such that $e.start \leq k \leq e.end$.

Additional costs are incurred to maintain the stab lists in the XR-tree. [3] shows that the update costs for insertion and deletion are $O(log_f^N + C_{DP})$ and $O(log_f^N + 3C_{DP})$ respectively where C_{DP} denotes the cost for one displacement of a stabbed element.

The XR-tree is unable to handle recursive XML elements well. When the XML data is highly recursive, the possible number of pages for one stab list is $2h_d$, where h_d is the maximum number of nestings of the element nodes indexed. This increases the update costs and storage requirements.

[1] put forth a preliminary proposal of the XB-tree which combines the structural features of both the B^+-tree and the R-tree. The XB-tree first indexes the pre-assigned intervals of elements in a tree structure. From this perspective, the XB-tree is similar to a one-dimensional R-tree. Next, the XB-tree organizes the start points of the intervals in the same way as the B^+-tree.

Figure 4 illustrates the XB-tree that is constructed for the directory elements in Figure 1. Each internal node maintains a set of regions that completely include all the regions in

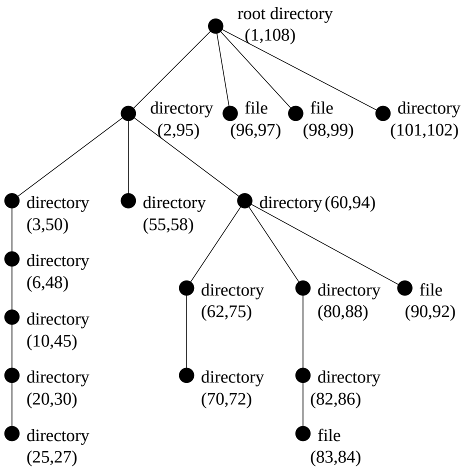


Figure 1: Example of an XML Tree.

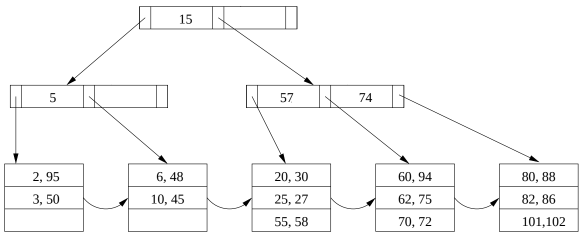


Figure 2: The B^+-tree

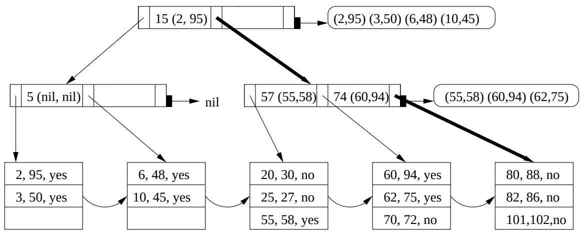


Figure 3: The XR-tree

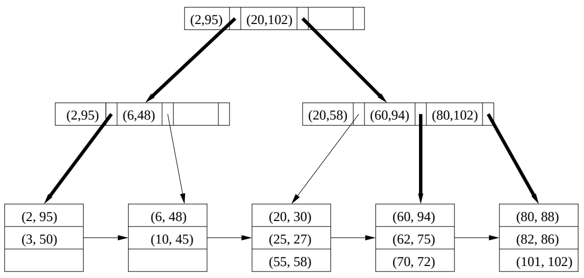


Figure 4: The XB-tree

their child nodes. The regions in the nodes of the XB-tree may overlap partially. However, it differs from the R-tree in that the start points are sorted in a strictly increasing order. In contrast to the XR-tree, the XB-tree does not store duplicate copies of data. This leads to lower update costs and more efficient space utilization.

3. BASIC SEARCH

The structural join operation requires two basic search: *findDescendants* and *findAncestors*. We will discuss how the three index-based solutions carry out these search procedures. We also investigate the conditions under which the *findAncestors* search in the XB-tree is optimal.

The *findDescendants* procedure retrieves all the data entries that can be covered by an interval. The B^+-tree, XR-tree, and XB-tree all utilize the efficient B^+-tree range search to find matching descendant occurrences. Suppose we want to find all the files which are contained in a particular directory element. We first traverse down the various index trees for the file element by comparing the start points of the keys in the index nodes. Next, a sequential scan on the leaf nodes is carried out until we encounter a data entry whose interval lies beyond the given ancestor directory element interval.

On the other hand, the *findAncestors* procedure looks for all the data entries that can cover an given interval. Consider Figure 1 again. Suppose we want to search for all the directory elements that contain the file element with the interval (90,92). The B^+-tree has to perform a sequential scan on the list of directory elements to find the elements whose intervals contain the interval of the given file element. That is, the search commences from the first element in the ancestor list, and ends when the start point of an ancestor element is greater than the start point of the descendant element. Thus, the directory elements labeled with the intervals (2,95) and (60,94) would be retrieved (see Figure 2). Clearly, this solution is not effective for the low ancestor-selectivity data.

The XR-tree uses the B^+-tree equality search to traverse down the index tree for the ancestor directory element. The search key is the start point of the given descendant file element. During the search process, all the directory elements in the result set can be collected from the stab lists of the non-leaf nodes, and finally from the leaf page. Consider the index tree for the directory element in Figure 3. Suppose we want to find all the directory elements which are the ancestors of the file element (90,92). Using 90 as the search key, we will retrieve the element intervals (2,95) and (60,94) from the stab lists of the non-leaf nodes in the search path. All the elements in the result set will be obtained when we finally reach the leaf page (the first leaf page from the right). The search path is highlighted in Figure 3.

3.1 Optimal Search in the XB-Tree

To process the *findAncestors* queries, the XB-tree starts from the root node and descends to the leaf nodes in a manner that is similar to the *R*-tree, that is, only the paths with the intervals that can cover the given interval would be searched. Finally, the data entries in the leaf nodes whose intervals can cover the given interval are output as results. In the worst case, the *findAncestors* operation may need to search the entire XB-tree.

The *findAncestors* operation in the XB-tree is optimal when the intervals in the index nodes in the XB-tree are the minimum bounding intervals of the intervals in their child nodes. This is because every search path, except for the last search path, will yield at least one ancestor element that contains the given descendant element.

Definition (Valid Path): A root-to-leaf search path in an XB-tree for a *findAncestors* operation is a valid path if the leaf node of the path contains at least one ancestor element interval that covers the given descendant interval.

To illustrate, let us search the XB-tree in Figure 4 for the ancestors of the file element (90,92) in Figure 1. Both the root-to-leaf search paths $(2, 95) \rightarrow (2, 95) \rightarrow leaf$ and $(20, 102) \rightarrow (60, 94) \rightarrow leaf$ are valid paths since they lead to leaf nodes that contain the desired data entries. On the other hand, the search path $(20, 102) \rightarrow (80, 102) \rightarrow leaf$ is not a valid path.

Theorem 1: Let the intervals of the index nodes in an XB-tree be the minimum bounding intervals. Given an element interval, every search path of the *findAncestors* operation must be a valid path except for the last search path.

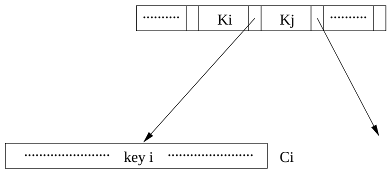


Figure 5: Multi-path Search

Proof: An index entry in an XB-tree consists of an interval and a child node pointer. Let K_i and K_j be two consecutive keys (or intervals) in an XB-tree index node. Let C_i be the child node that is pointed to by the pointer associated with K_i (see Figure 5). Suppose the intervals K_i and K_j overlap and the given descendant element interval I_d lies in the overlap. Then the path $K_i \rightarrow C_i$ is not the last path in the XB-tree. Since this is a XB-tree with minimum bounding intervals, there must exist a key key_i in the child node C_i such that $key_i.e = K_i.e$. Consider $K_i.e > I_d.e$ and $key_i.s < K_j.s < I_d.s$. We have $key_i.s < I_d.s < I_d.e < key_i.e$, which implies that C_i has at least the interval key_i which can contain the descendant element interval I_d. The same reasoning applies as we continue to traverse down the XB-tree from the child node C_i. Thus, we can guarantee that every search path, except for the last path, will yield at least one ancestor element that contains I_d.$\square$

Note that the last search path of a *findAncestors* operation is not necessary a valid path because its leaf node entry with the maximum end point may have a start point that is greater than the start point of the descendant element interval.

4. STRUCTURAL JOIN

Algorithm 1 shows a generic index-based structural join algorithm. It takes as input two lists of element intervals A and D. The lists are sorted by the start points (ascending order). These lists are scanned with the help of the indexes and merged to obtain all the pairs $(a_i,\ d_j)$, $a_i \in A\ d_j \in D$, such that a_i contains d_j.

A stack which stores a sequence of ancestor intervals is maintained. Each interval in the stack is a descendant of

Algorithm 1 Ancestor-Descendant Structural Join

```
  Input: Lists A and D
  Output: All matching $(a_i, d_j)$, such that $a_i$ covers $d_j$
 1: $a = A.first$
 2: $d = D.first$
 3: $stack = \emptyset$
 4: while $!eof(D)$ and $!(eof(A)$ and $isEmpty(stack))$
 5:    pop all $a_i$ from $stack$, such that $a_i$ can't cover $d$
 6:    let $a_l$ be the last element (if any) popped
 7:    $next = \text{Max}(a_l.e, a.s)$
 8:    if $stack = \emptyset$ and $d.s < a.s$ then
 9:       $d$ = first $d_i$ in $D$, such that $d_i.s > a.s$
10:    else
11:       if $d_i.s > a.s$ then
12:          push $findAncestors(T_a, d, next)$ into $stack$
13:          $a$ = first $a_n$ in $A$, such that $a_n.s > d.s$
14:       output $(a_i, d)$ for all $a_i \in stack$
15:       $d = D.getnext()$
```

the interval below it. Two cursors a and d are used to track the current elements being processed in the input lists A and D respectively. The stack is initially empty and the cursors are first set to point to the first element in the lists (Lines 1-3).

In each iteration, the algorithm checks whether the current descendant element d has ancestors. If it is impossible for d to have ancestors, then we update d to point to the first element interval in the list D whose start point is greater than the start point of current ancestor element a (Line 8-9). Otherwise, we retrieve the set of ancestor intervals for the current descendant element d that is not already in the stack. This is accomplished by calling the function *findAncestors*. The results obtained are pushed into the stack (Lines 12). The cursor a is updated to point to the next possible position that is likely to have descendants elements (Line 13). The new elements intervals that are pushed into the stack, together with the element intervals that are already in the stack, constitute all the ancestors of the current d (Line 14). Finally, d is advanced to the next element interval in D.

The loop in the algorithm terminates when any of the following conditions is met:

1. List D is exhausted, indicating that the element d no longer contributes to the result set;
2. List A is exhausted and the *stack* is empty, indicating that there is no more element a available to match d.

4.1 Consecutive FindAncestors Search

In Algorithm 1, we need to process *findAncestors*(D_i) and *findAncestors*(D_{i+1}) consecutively, where D_i and D_{i+1} are the descendant element intervals. Figure 6 illustrates how the B^+-tree, XR-tree and XB-tree process these two consecutive *findAncestors* search. The two gray areas labelled D_i and D_{i+1} denote the the interval ranges of the ancestor elements in the leaf pages that contain the descendant element intervals D_i and D_{i+1} respectively. The search paths for the two consecutive search are denoted by dashed and solid lines respectively.

We observe that the B^+-tree can support consecutive *findAncestors* search efficiently. This is because the sequential scan for the second *findAncestors* search continues from the leaf page after the first search ends.

In contrast, the XR-tree cannot distinguish the ancestor elements which have been retrieved in the first *findAncestors*

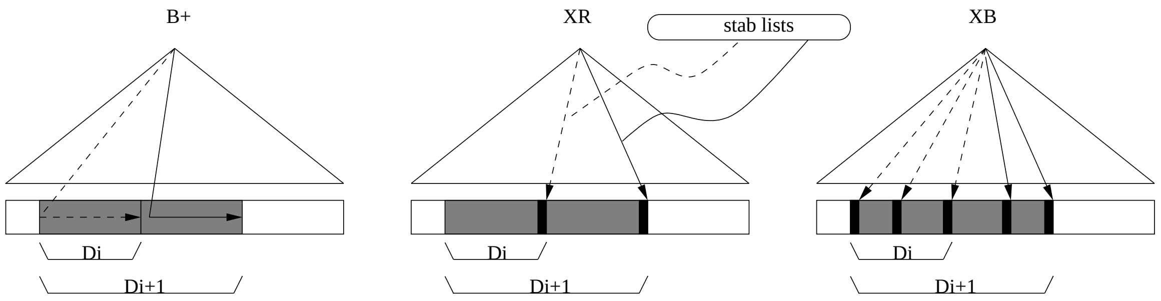


Figure 6: Consecutive *FindAncestors* Search for D_i and D_{i+1}

search when it evaluates the second *findAncestors* search. This is because the majority of the matching ancestor elements are obtained from the stab lists. We observe that multiple access of the element intervals in the stab lists is not a serious problem for the XR-tree when there is no recursion. However, its performance degrades with increasing levels of nesting, that is, when the data is highly recursive.

The XB-tree can also support consecutive queries efficiently since it stores the interval information in the index nodes. We extend the *findAncestors* search in the XB-tree to a consecutive multiple *findAncestors* search. We observe that all the ancestor element intervals of the current descendant element d before the start point of the ancestor element a are already captured in the *stack* (Line 5 in Algorithm 1). Therefore, we only need to retrieve the ancestor results of d after $a.s$. We use an additional parameter *next* to restrict the values of the start points of ancestor element intervals to be greater than the value of *next* (Line 12). Since the common ancestors for both D_i and D_{i+1} are retrieved only once, this leads to much savings in I/O costs.

5. EXPERIMENTAL EVALUATION

In this section, we present the results of our experiments to evaluate the three index structures for XML structural joins. We implement the B^+-tree [2], XR-tree [3] and XB-tree [1] structural join algorithms in Java. We also implement a variant of the XR-tree that decreases the update cost, and call it XR-v. In the original XR-tree [3], all the elements that are stabbed by the keys in one index node are stored in the stab list of this index node. Since the stab list is an ordered element list, the cost to maintain a large stab list is high. In the XR-v, we store the elements that are stabbed by a key in separate lists. Although XR-v may increase the query cost compared to the original XR-tree, it is able to decrease the update cost when the stab list is large.

We generate synthetic XML documents that contain a set of uniformly distributed elements. A structural join is carried out on the elements "Ancestor" and "Descendant". We fix the number of "Ancestor" and "Descendant" elements at 120,000 and 240,000 respectively, and vary the selectivity values and levels of nestings. Table 1 shows the characteristics of the data set generated, and the range of values for the parameters. All the experiments are carried out on a Pentium IV 2.4 GHz CPU with 1 gigabyte RAM. We record the average results of 5 runs.

5.1 Query Performance

This set of experiments evaluates the query performance

Element	Number	Selectivity	Nesting
Ancestor	120,000	1%-70%	1-220
Descendant	240,000	1%-80%	1-140

Table 1: Characteristics of Dataset

of the various indices. The I/O cost incurred by the structural join is used as the performance metric. All the index structures are built by bulkloading the elements. The node occupancy, except for the root node, is kept at 50%.

Ancestor Selectivity. We first investigate the effect of low ancestor selectivity. The ancestor selectivity is limited to the range of 1%-5%, while the ancestor and descendent nesting levels, and descendant selectivity are fixed at 50, 5 and 10% respectively. Figure 7(a) shows the results when we keep the root nodes of the various indices in the buffer, while Figure 7(b) presents the results when a 100 KB buffer is used. The XB-tree gives the best performance in both situations by avoiding the sequential scans needed in the B^+-tree.

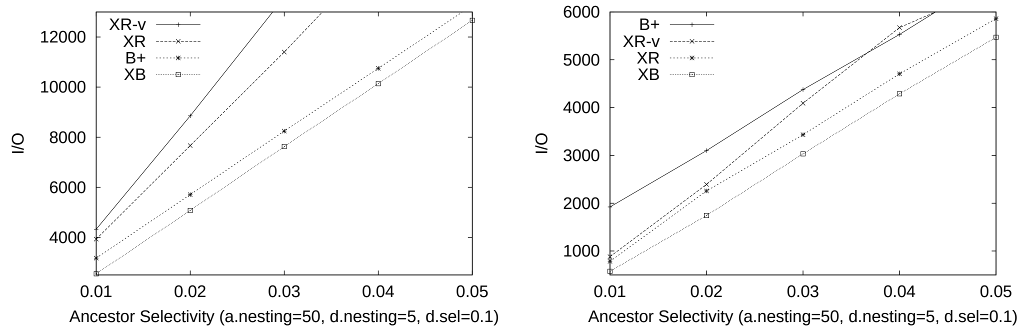
The XR-tree and its variant shows the worst performance as they need to access the stab lists multiple times in Figure 7(a). The XR-tree (XR-v) benefits the most from the increasing buffer size. With sufficient buffers, we can pin the stab list pages in the buffer, and the performance of the XR-tree (XR-v) will improve dramatically.

Figure 8 shows the results for higher values of ancestor selectivity: 10%-70%. Since the "Ancestor" elements involved in the join are uniformly distributed, we need to access most of the leaf pages in the index structures. Thus, it is not surprising that both the I/O costs for the XB-tree and the B^+-tree are almost the same.

When the buffer size is increased, and the ancestor selectivity is above 40%, the XR-tree and XR-v outperforms the XB-tree and B^+-tree (see Figure 8(b)). Unlike the B^+-tree and the XB-tree which need to access most of the leaf pages, the XR-tree and the XR-v only access the leaf page that contains the last ancestor element that covers the descendant element. The rest of the ancestor elements that contains the descendant element can be obtained from the stab lists which are most likely to be in the buffer.

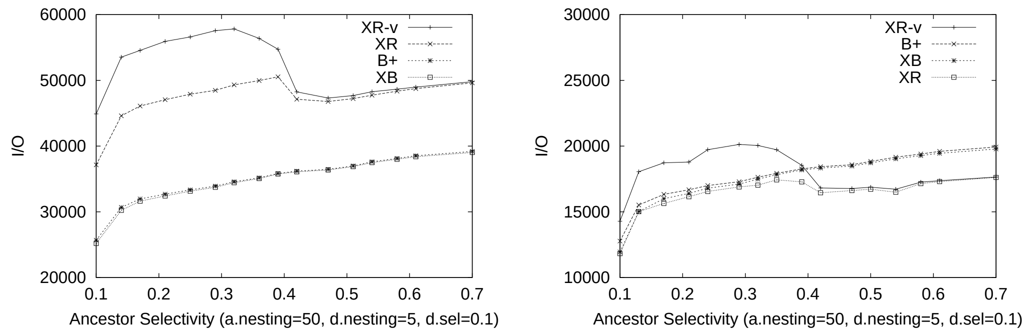
Descendant Selectivity. Next, we examine the effect of varying the descendant selectivity. The results are shown in Figure 9. Again, the performance of the XR-tree and the XR-v largely depends on the buffer size for the same reasons given above.

The I/O costs do not increase when the descendant selectivity is above 10% for all of index structures in both



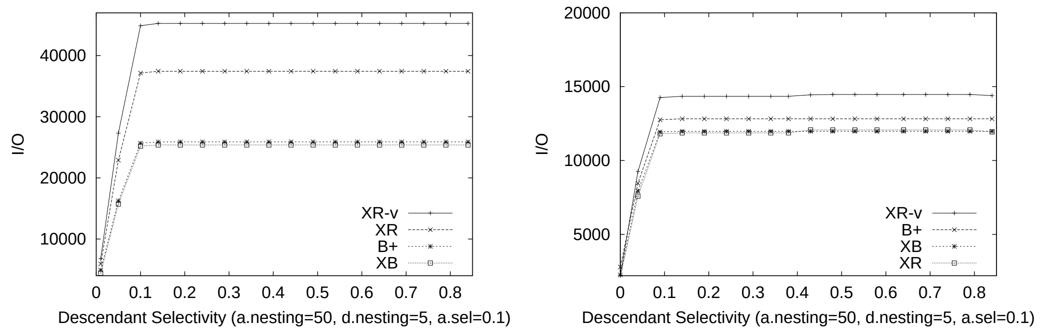
(a) Root in buffer only (b) Buffer size = 100K

Figure 7: Low Ancestor Selectivity



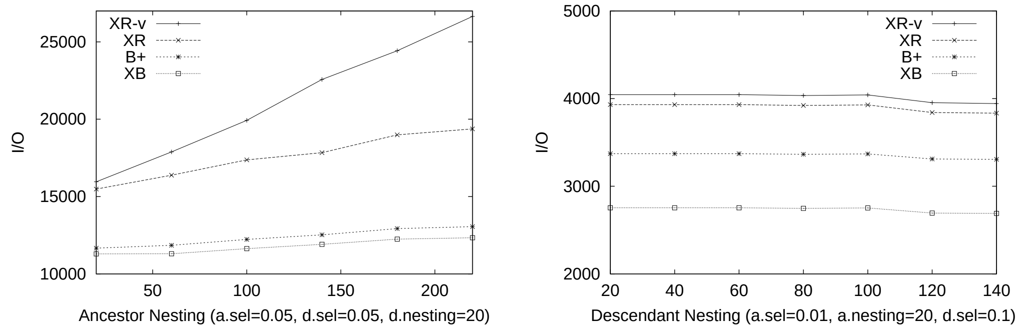
(a) Root in buffer only (b) Buffer size = 100K

Figure 8: High Ancestor Selectivity



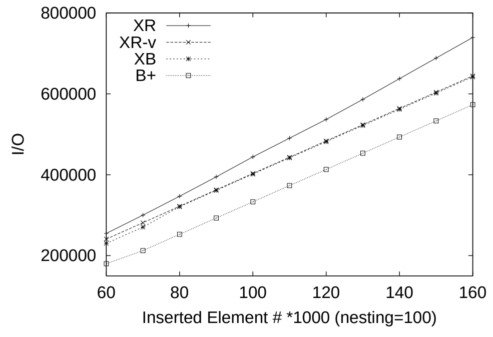
(a) Root in buffer only (b) Buffer size = 100K

Figure 9: Descendant Selectivity

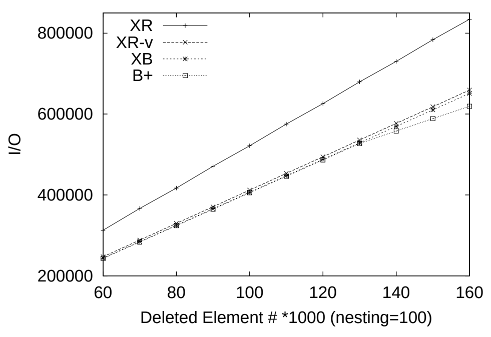


(a) Ancestor (b) Descendant

Figure 10: Levels of Nestings



(a) Insertion



(b) Deletion

Figure 11: Update Cost

graphs. The "Descendant" elements involved in the join are uniformly distributed in the entire "Descendant" element set. Hence, the increasing number of elements involved in the join would not lead to additional I/O costs.

Nesting Levels. In this experiment, we examine the effect of varying levels of ancestor and descendant nesting. Only the root nodes of the index structures are kept in the buffer. Figure 10(a) shows that increasing the levels of ancestor nesting will increase the size of the stab lists and contribute to the rapid performance deterioration of the XR-tree (and the XR-v). In contrast, the curves for the B^+-tree and the XB-tree are almost flat. Figure 10(b) shows that the various index structures are independent of the levels of descendant nesting.

5.2 Update Performance

This set of experiments examines the insertion and deletion costs. The buffer is turned off here. The index structures are initially empty, and we randomly insert 160,000 element intervals. Figure 11(a) shows the insertion results. The XR-tree has the highest I/O cost since it has to maintain a set of ordered stab lists. Compared to the XR-tree, XR-v performs better due to its smaller individual stab lists.

Next, we randomly delete 160,000 elements from the index trees and record the number of I/Os. Figure 11(b) shows the deletion costs. The performance of deletion comes very near to that of insertion. Considering the characteristics of the different index structures, the results of the update experiments are as expected.

5.3 Space Utilization

Finally, we investigate the space consumption of the various index structures. Each index is built by bulkloading 240,000 elements, and every node in the index is 50% full except for the root node. We only vary the number of nesting levels, since this is the only parameter that may affect the space utilization. Figure 12 indicates that the XR-v requires the most storage space due to its individual stab lists. As the number of nesting levels increases, the space consumption of the XR-tree and XR-v will increase slightly while the sizes of the XB-tree and B^+-tree remain stable.

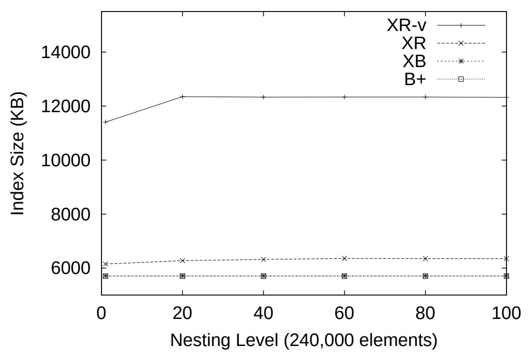


Figure 12: Space Consumption

6. CONCLUSION

In this paper, we have compared and analyzed the performance of the B^+-tree, XB-tree and XR-tree for XML structural join. We examined the conditions under which the *FindAncestor* search in the XB-tree is optimal. We also extended the XB-tree to efficiently evaluate multiple consecutive *FindAncestors* search. Experiment results indicate that all three indexes give comparable performances for non-recursive XML data, while the XB-tree outperforms the rest for highly recursive XML data.

7. REFERENCES

[1] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[2] S-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, 2002.

[3] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ICDE*, 2003.

[4] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB*, 2001.

[5] 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*, 2002.

[6] 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*, 2001.