Efficient Handling of Sibling Axis in XPath

G. V. Subramanyam P. Sreenivasa Kumar

Dept of Computer Science and Engg.
Indian Institute of Technology Madras
Chennai - 600 036, India
{gvs, psk}@cs.iitm.ernet.in

ABSTRACT

XML is widely used for representing and exchanging hierarchical data and queries naturally specify hierarchical patterns to select relevant parts of an XML document. These patterns have a sequence of selection predicates connected by operators representing structural relationships (ancestor-descendant or preceding-following sibling). In this context, the operation of structural join involves discovering pairs of nodes that are structurally related from the cross product of two sets of nodes.

Current XPath processing algorithms focus more on solving queries containing ancestor-descendant relationship but, pay little attention to the equally important preceding-following sibling relationship. In this paper, we propose a new solution for processing queries containing sibling axis by using a *sibling-list* at each level of the XML document tree. We show that using *sibling-lists*, the time complexity of the proposed join algorithms is linear in the sum of lengths of input lists. An extensive experimental evaluation shows that our join algorithms perform significantly better than the currently existing sibling join algorithms.

1. INTRODUCTION

XML has emerged as a popular method for representing semi-structured data and for exchanging data on the web. With its wide-spread use, large collections of XML data need to be efficiently stored, managed and queried. XPath [3] has become a popular language for querying XML data. XPath provides constructs for specifying ancestor-descendant and following-sibling and preceding-sibling patterns as structural predicates in the queries. Recently several algorithms for efficiently computing these structural relationships in XML data have been proposed in the database literature and are generally called as structural join algorithms. A majority of these algorithms focus on computing ancestor-descendant relationship and give secondary treatment to the sibling axis of XPath. However, there is a need for development of efficient algorithms for processing queries containing sibling axis also. The following query illustrates the use of sibling axis. The XPath query –

machine//part[type="T2000"]/following-sibling::part[1]

selects all *part*s that immediately follow *part*s of *type* value "T2000". The query comprises of both structural matching and predicate evaluation (*type="T2000"*). The structural matching addresses both ancestor-descendant relationship such as "*machine//part*" and preceding-following relationship such as "*part/following-sibling::part*".

Two types of storage models are usually assumed for XML data, namely relational database storage and native storage. The approaches that base their implementations on RDBMS translate XML queries into SQL queries. The translation is accomplished by providing an extra layer on the top of relational database system. Where as, in native storage based systems, XML queries are directly translated into algorithms that operate on the storage layer. The native XML database systems [1, 5, 10, 11] focus on evaluating queries that contain ancestor-descendant relationships i.e., regular path expressions with / or // operators, and are yet to propose techniques for handling sibling axis. The effort of solving sibling axis is largely dealt in the literature that uses relational databases for storing and querying XML documents . There are many approaches that shred (decompose) XML documents into relations [7, 9, 12, 14, 15]. One such approach is to store XML nodes as tuples in a relation with schema (*start*, *end*, *parentId*, *level*)[15]. Similar to this approach, [14] uses a relation with schema (*Id*, *parentId*, *endDescId*), in which the attribute *Id* (n^{th} node, when XML nodes are ordered in document order) and *endDescId* correspond to the attributes *start* and *end* of the schema given in [15], respectively. In [9], (*preorder*, *postorder*, *parentId*) is used as schema, where *preorder* and *postorder* correspond to *start* and *end* of the schema given in [15], respectively. The storing of XML data in relational database systems allows us to maintain indices on certain attribute(s). These indices help us to expedite query evaluation while joining the relations. Although it improves the join performance, it makes multiple scans over the tuples in the relations, resulting in a non-linear performance.

In this paper, we assume that XML document is stored as collection of element lists and each node in the list is represented by the popular node identification scheme (*DocId*,

ADVANCES IN DATA MANAGEMENT 2005
Jayant Haritsa, T.M. Vijayaraman (Editors)
© CSI 2005

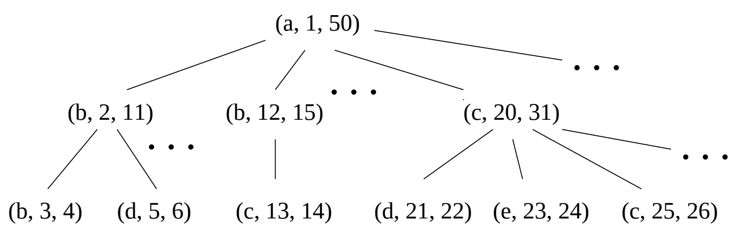

Figure 1: A Sample XML Document

start, *end*, *parentId*, *level*). In the above scheme, *DocId* uniquely identifies a document in the collection[1]. Since *start* value of a node is unique, the *parentId* of a node is taken as the *start* value of its parent. We devised a couple of algorithms that take linear time for processing queries containing sibling axis, by using non-indexed lists. Our join algorithms sequentially traverse the element lists and perform the join by using a *sibling-list* at each level of the XML document tree. The XML instance tree that we are considering is the one induced by the elements in the given element lists. At any instance, each *sibling-list* has elements under common parent.

Experimental results show that our join algorithms are both CPU and I/O optimal, when compared with the join algorithms that use B$^+$-tree index on either *start* or *parentId*. These later approaches are the only suggested ones in the literature for handling sibling axis and we compare our algorithm with these. Our algorithms can be used in both native XML databases and RDBMS based XML systems to boost the performance of XML query processing. The contributions of the paper can be summarized as follows:

- We propose new algorithms, called `multi-list-tree` join algorithms, for handling sibling axis in XPath query processing and report on implementation of these algorithms.
- We refined the approaches suggested in the literature for handling sibling axis in the context of RDBMS based XML systems by proposing the use of B$^+$-tree index on *start* or *parentId*. We call these algorithms as `indexed-loop` join algorithms. We find that the `multi-list-tree` join algorithms take linear time and guarantee that each element in the input lists is accessed once. This is in contrast with the `indexed-loop` join algorithms, where it makes multiple scans over the elements in the lists.
- An extensive experimental evaluation of our join algorithms was conducted and results are compared with the `indexed-loop` join algorithms. Our comparative study shows that our join algorithms perform significantly better than the `indexed-loop` join algorithms.

The organization of this paper is as follows. We provide background material and related work in Section 2. We discuss `indexed-loop` join algorithms in Section 3.1; and provide a discussion on `multi-list-tree` join algorithms in Section 3.3. Section 4 describes performance study and we conclude in Section 5.

[1]The scope of the paper is limited to single document and extending the algorithms for multiple documents is rather straight forward

2. BACKGROUND AND RELATED WORK

2.1 Query Patterns

Queries in XPath or XQuery [4] use path expressions which are sequences of location steps to navigate through XML document tree [9]. Basically, queries specify patterns that match the relevant twigs in the XML document. As XML employs tree-structured data model for its representation, determining structural relationships (the relationship may be either ancestor-descendant or preceding-following) between any pair of elements and finding all element pairs that are structurally related between the two given element lists plays an important role in solving queries.

2.2 Numbering Scheme

To solve structural relationship between any two elements a numbering scheme is proposed [1, 6, 15] which is based on the region encoding of elements. In an XML document, each element is assigned with a tuple ($DocID$, $start$, end, $parentId$, $level$) and between any two elements (say, a and d), the ancestor-descendant relationship can be determined as follows: (i) If $a.start < d.start$ and $a.end > d.end$ then a is an ancestor of d. (ii) In addition to condition (i), if $level(a) = level(d) - 1$, then a is a parent of d. Similarly, preceding-following sibling relationship can be determined as (i) If $a.parentId = b.parentId$ then b is a sibling of a. Further, if $b.start > a.start$ then b becomes *following-sibling* of a otherwise, b becomes *preceding-sibling* of a. For each element, the (*start*, *end*) can be assigned by making a depth first traversal of an XML document tree [1, 15]. If the *start* value of a node v is η then its *end* value can be assigned with $\eta + count(subtree(v)) + 1$ $(+\lambda)$, where $\lambda \geq 0$ provides space for future insertions within the region encoding (*start*, *end*). The function *count* returns the number of nodes in the subtree rooted at node v.

Figure 1 shows a sample XML document tree containing nodes with *start* and *end* values. The root node a is assigned with (1, 50) and the children are (2, 11), (12, 15), (20, 31) and so on. The *parentId* of a node is the *start* value of its parent except for the root node, the *parentId* is assigned to zero.

2.3 Structural Joins

Structural join is a basic operation that operates on two element lists and output pairs that are structurally related. We generalize the definition of structural join to include sibling relationships also.

```
Algorithm: Btree-on-start ( PList, FList )
//PList: list of Preceding elements indexed using B$^+$-tree
//FList: list of Following elements sorted on their start values

1: for each $f$ in the FList {
2:     $p$ = first node in PList whose $p.start > f.parentId$;
3:     while ($p.start < f.start$){
4:         if ($f.parentId == p.parentId$){
5:             append ($p$, $f$) to the OutputList;
6:         }
7:         $p$ = next element in PList after $p$;
8:     }
9: }
```

Figure 2: Btree-on-start algorithm

Definition 1. For any two input element lists - say L_1 and L_2, a structural join is to output all pairs (p_i, q_j), $p_i \in L_1$ and $q_j \in L_2$, such that p_i and q_j are structurally related (ancestor-descendant or preceding-following).

The evaluation process involves joining the two given input element lists and output all the resultant pairs (p_i, q_j) sorted either in ascending order of $p_i.start$ or $q_j.start$. The query processor chooses the order dynamically during the runtime.

Most of the query processing techniques that apply structural joins between the element lists assume tree-structured model of XML data [1, 5, 11, 15]. There are many techniques proposed using the more popular RDBMS [12, 14, 15] having SQL as the querying agent. The motivation behind these techniques is that a large amount of XML data as of now and in future is expected to be stored in relational database systems. The XML documents are decomposed into relational tables having (*DocID*, *start*, *end*, *level*) as the schema. Zhang et. al. [15] propose a notion of "containment" join using multi-predicate merge join (MPMJGN) that differs from the traditional relational-style merge join algorithms. Li and Moon [11] propose *EE*-join and *EA*-join algorithms for generating ancestor-descendant pairs by merging the two given element lists sorted on their (*DocID*, *start*) values. These algorithms make multiple scans over the elements in input lists during the join. An improved version of MPMJGN algorithm can be found in [1], where an ancestor stack is used to guarantee that each element is accessed once during the process of join. Recently, Bruno et. al [2] proposed a twig-join algorithm which is a generalization of stack-tree-algorithm [1]. It uses path stack for path expressions queries and twig-stack for tree-pattern queries.

The work more related to our discussion can be referred in [14], where XML documents are stored in relational tables; and the given path expressions are translated to SQL queries. When index is maintained on certain attributes of the relation, the query processor may choose to perform `indexed-loop` joins during the join process. Although the `indexed-loop` joins perform well in most cases, in worst case it makes multiple scans over the elements in the input lists.

3. STRUCTURAL JOIN ALGORITHMS

Here, we explain the structural join algorithms in the context of solving queries containing sibling joins. Initially, we discuss the join algorithms that are implemented using the index structures of the relational database systems. Then, we present the details of our proposed join algorithms for processing queries containing sibling joins.

Consider a query p/*following-sibling*::f that returns all pairs (p, f) such that f is *following-sibling* of p. Let PList (= [P_1, P_2, ...]) be the list of p–elements. These are candidate *preceding* nodes. Let FList (= [F1, F2, ...]) be the list of f–elements. These are candidate *following* nodes. Each list is sorted on their (*DocId*, *start*) values. These lists are stored natively in a storage structure indexed using *tag* name as a key, which could retrieve appropriate list when the specific tag appears in the query pattern. An element list can be stored as a sequence of elements represented as objects with attributes *DocId*, *start*, *end*, *parentId* and *level*. When the query processor finds a pattern such as p//f or p/*following-sibling*::f in a query, it retrieves PList (the list of p nodes) and FList (the list of f nodes), and then performs the join by sequentially traversing the lists.

3.1 Relational Implementation

Here we assume that an XML document is decomposed into relational tables with one table for each distinct tag that appears in the document. Each node in an element list is represented as a tuple in the corresponding relational table with schema (*start*, *end*, *parentId*, *level*). The table containing the list of p – nodes is denoted as T_p. Queries in XPath are translated into equivalent SQL queries before accessing the relations. For example, the query – p/*following-sibling*::f, has its equivalent SQL statement as given below:

Q_1: **SELECT** P.*, F.*
FROM T_p as P, T_f as F
WHERE P.parentId = F.parentId and F.start > P.end;

The tables T_p and T_f in the **FROM** clause are theta-joined to output each F in T_f that is *following-sibling* to some P in T_p. The *preceding-sibling* axis can be evaluated similarly. The following discussion explains the process of joining relations efficiently by maintaining indices on certain attributes of the relations.

```
Algorithm: Btree-on-parentId ( PList, FList )
//PList: list of Preceding elements indexed using B+-tree
//FList: list of Following elements sorted on their start values

1: for each $f$ in the FList{
2:     $p$ = first node in PList whose $p.parentId = f.parentId$;
3:     while ($p.parentId == f.parentId$){
4:         if ($p.start < f.start$){
5:             append $(p, f)$ to the OutputList;
6:         }
7:         $p$ = next element in PList after $p$;
8:     }
9: }
```

Figure 3: Btree-on-parentId algorithm

3.1.1 *B$^+$-Tree index on* start

Consider the query Q_1, when no index is maintained on the attributes of T_p and T_f, the query processor may choose to perform nested-loop join (or probably sort-merge join), which is computationally expensive. If we consider maintaining index on *start* attribute, the join performance improves; this is due to the reduction in the number of tuple accesses.

The algorithm shown in the Figure 2 generates output of the structural join in the *Following* order. The two input lists are joined on the multiple inequality conditions that characterize the preceding-following relationship based on (*DocId*, *start*, *end*, *parentId*, *level*) node representation. In relational database context, the tables that participate in the join use the same node representation as schema.

As the result is output in the *Following* order, the *Following* list (FList) acts as outer operand joining with the inner operand, which is *Preceding* list (PList). For each f in the FList, a lookup is made in the B$^+$-tree of PList. Similarly, if the output to be generated is in the *Preceding* order, then PList becomes the outer operand. As B$^+$-tree is constructed taking *start* of a node as key, all the nodes occurring at the leaves of the B$^+$-tree are sorted on *start* value.

The region encoding of any element – say a, represented using (*start*, *end*, *parentId*, *level*), permits us to find all the descendants with specific *tag* name – say b, by making a range search (i.e., $a.start < b_i < a.end$) on B$^+$-tree constructed on b list. During the join process, for any element f in FList, all the elements in the PList, for which f is the *following-sibling*, can be found by (i) retrieving a smallest element p (locate using B$^+$-tree) such that $p.start > f.parentId$ (line 2); and (ii) then making a cursor move over the leaves of B$^+$-tree, starting at element p, to output all those pairs (p_i, f) to the `OutputList`, such that $p_i.start < f.start$ and $p_i.parentId = f.parentId$ (lines 3–7)

3.1.2 *B$^+$-Tree index on* ParentId

The underlying relational database system may choose to maintain B$^+$-tree index on *parentId*; this results in the reduced number of tuple accesses and in turn provides better performance than the algorithm shown in Figure 2. The `indexed-loop` join algorithm shown in Figure 3 proceeds as follows: For any element f in FList, all the elements in the PList, for which f is the *following-sibling*, can be found by (i) retrieving an element p such that $p.parentId = f.parentId$ (line 2); and (ii) then making a cursor move over the leaves of B$^+$-tree, starting at element p, to output all those pairs (p_i, f) to the `OutputList`, such that $p_i.start < f.start$ and $p_i.parentId = f.parentId$ (lines 3–7).

3.2 Analysis of Indexed-loop Join Algorithms

In traditional relational database indexed-loop joins, one can not establish a linear time complexity when joining predicate involves multiple attributes. In this Section, we provide an analysis on the complexities of both `indexed-loop` join algorithms – B$^+$-tree index on *start* and *parentId*.

3.2.1 *Analysis of Btree-on-start algorithm*

During the join process, a node – say *f*, in the outer operand, which is typically a FList, is joined with nodes in the inner operand – PList. Consider the case when no two fs in the FList have some common sibling nodes from PList, then the algorithm runs in linear time showing $\bigcirc$(|PList| + |FList|) i.e., each node in the element lists is accessed only once.

Consider the case as shown in Figure 4, when F_1 has to be joined with P_1 and P_7, a lookup is made in B$^+$-tree of PList to retrieve all elements E_i such that $F_1.parentId < E_i.start < F_1.start$. In this instance, all the elements from P_1 to P_{10} are retrieved; in which only P_1 and P_7 are actually joined with F_1 as these are under common parent(*root*). Here it is observed that we make unnecessary visit of elements P_2 to P_6 and P_8 to P_{10}. Similarly, the elements P_1 to P_{10} are visited again while processing the join with F_2. For each child F_i of *root* occurring after node P_{10}, we as well make unnecessary visit of elements P_2 to P_6 and P_8 to P_{10} during the process of join. This illustrates the worst case behavior of the algorithm and is found to be $\bigcirc$(|PList| × |FList|).

3.2.2 *Analysis of Btree-on-parentId algorithm*

To avoid the unnecessary visit of the nodes P_2 to P_6 and P_8 to P_{10} during the processing of the scenario shown in Figure 4, we maintain a B$^+$-tree index on the nodes of XML document taking *parentId* as key[2]. During the execution

[2]Since multiple nodes in XML tree could have same par-

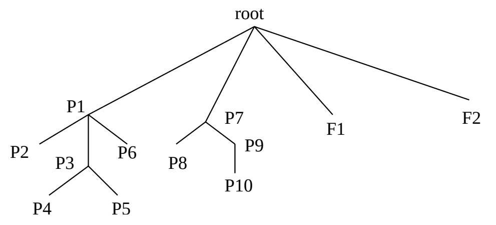

Figure 4: Sample scenario for Btree-on-start algorithm

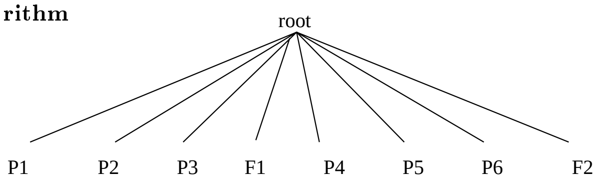

Figure 5: Sample scenario for Btree-on-parentId algorithm

of the scenario shown in Figure 5, node F_1 is joined with nodes P_1, P_2, and P_3 by a making range search on B^+-tree selecting nodes E_i such that $F_1.parentId = E_i.parentId$ and $E_i.start < F_1.start$. Similarly, node F_2 is joined with $P_1, P_2, \ldots, P_6$. Here we observe that nodes P_1, P_2, and P_3 are accessed twice – for both F_1 and F_2. If we have some node F_3 lying next to F_2 as sibling, then F_3 is joined with all the nodes $P_1, P_2, \ldots, P_6$ by revisiting them. This shows that the elements in the PList are visited multiple number of times; and the run time complexity becomes $\bigcirc(|PList| \times |FList|)$

3.3 Multi-List Tree Join

We now describe the `multi-list-tree` join algorithms that take time linear to the sum of lengths of input lists. Just like the algorithm shown in Figure 3, at any instance, we process only with the set of nodes under common parent. But, the main drawback of the `indexed-loop` join algorithms is that, for elements in FList, we need to make multiple accesses of the elements in PList. We address this drawback and propose a new solution that joins nodes in the element lists using *sibling-lists*. A *sibling-list*, at any instance, has the property that it holds a set of nodes from PList that are under common parent; and it is maintained at each level of the XML document tree.

3.3.1 Multi-list-tree-following

The algorithm shown in Figure 6 takes two element lists: PList and FList, where each list is sorted on their *start* values. The output of the join is to generate all pairs (p, f) such that f is *following-sibling* of p and produce the result in *Following* order.

The description of the algorithm is as follows: p and f act as cursors for PList and FList respectively. They start at the beginning of these lists and sequentially traverse until the end of one the lists is reached. At run time, a *sibling-list* is maintained at each level so as to join with the *Following* nodes in FList. The *sibling-list* at level i is referred as i^{th} *sibling-list* i.e., SList[i]. Each*sibling-list* has an *id* specifying

ent, we allow duplicate keys when B^+-tree is constructed on *parentId*

```
Algorithm: Multi-list-tree-Following ( PList, FList )
//PList: list of Preceding elements sorted on their start values
//FList: list of Following elements sorted on their start values
//SList: list of elements that are under common parent

1:  p = first(PList);
2:  f = first(FList);
3:  while (not end of PList or FList) do
4:    if (p.start < f.start) then
5:      if (SList[p.level] ≠ ∅) then
6:        if (SList[p.level].Id == p.parentId) then
7:          append p to SList[p.level];
8:        else
9:          SList[p.level].makeEmpty();
10:         SList[p.level].Id = p.parentId;
11:         add p to SList[p.level];
12:       end if
13:     else
14:       SList[p.level].Id = p.parentId;
15:       add p to SList[p.level];
16:     end if
17:     p = next element in PList after p;
18:   else
19:     if (SList[f.level] ≠ ∅) then
20:       if (SList[f.level].Id == f.parentId) then
21:         Output f with all elements in SList[f.level];
22:       else
23:         SList[f.level].makeEmpty();
24:       end if
25:     end if
26:     f = next element in FList after f;
27:   end if
28: end while
```

Figure 6: Multi-list-tree join algorithm that output pairs in the *Following order*

parentId of the nodes in the list at that instance.

During the join, when a node p from PList occurs before f, we check whether or not the *sibling-list* at $p.level$ is empty (line 5). If it is empty, we update the list's *id* with $p.parentId$ and then simply add p to the empty *sibling-list*, SList[$p.level$] (lines 14-16). Otherwise, in line 6, we check if SList[$p.level$].id equals to $p.parentId$; if it is, we append p to the nodes in SList[$p.level$] (line 7). If it doesn't satisfy this condition, we first empty the list, then update the list's *id* with $p.parentId$, and then add node p to the empty list (lines 9-11). We continue the above process with the next node in PList after p.

If a node f from FList occurs before p, we check whether SList[$f.level$].id equals to $f.parentId$; if it is found to be equal, we output f with all the elements in the list (line 20-21). Otherwise, we empty the corresponding list (line 23); this is because, if f is not a sibling of nodes in SList[$f.level$], then it is guaranteed that no node that occurs after f in FList is a sibling of nodes in SList[$f.level$]. The procedure continues with the next node in FList after f.

Example *[Multi-list-tree-following Algorithm]*

The Figures 7 (b) – (e) show the instances of the lists when the `Multi-list-tree-following` procedure works on the XML dataset shown in Figure 7(a). The result of the algorithm is to output all (p_i, f_j) pairs such that $p_i \in$ list of all p-elements and $f_j \in$ list of all f-elements and f_j is *following-sibling* of p_i. The input lists are logically merged

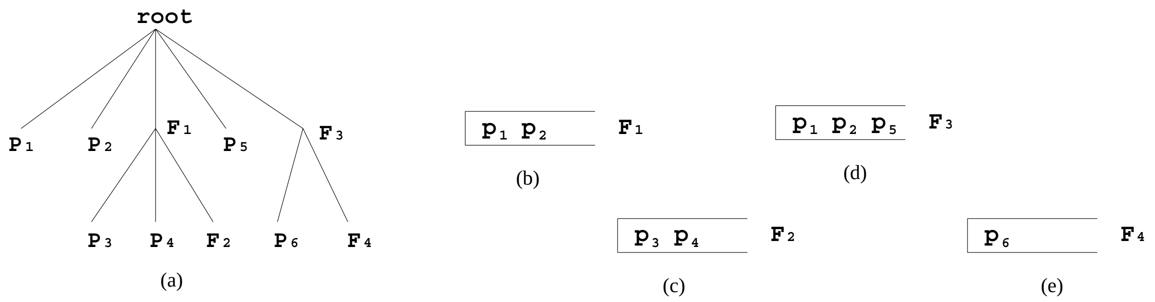

Figure 7: (a) XML dataset, (b)–(e) instances during the evaluation of `multi-list-tree` join algorithm

in document order and the element with smallest *start* is processed first. At each level of the XML document tree, a *sibling-list* (SList) is maintained and is initially empty. In the instance shown in Figure 7(b), we add P_1 and P_2 to the empty SList[2] (*sibling-list* at level 2, where *root* is at level 1). We output F_1, which is a next element in the merged list, with elements in SList[2]; this is because F_1 is sibling of both P_1 and P_2. Similarly in Figure 7(c), P_3 and P_4 are added to SList[3] and output with the next element in the merge list, which is F_2. In Figure 7(d), as P_5 is sibling of elements in SList[2], we append P_5 to its respective sibling list; and output F_3 with P_1, P_2, and P_5. Figure 7(e) shows the scenario where P_6 is not a sibling of elements in SList[3]. In this case, we first empty the list and then add P_6 as a new element to the empty list. Since F_4 is sibling of P_6, we output F_4 with P_6. During the process of the join, we can observe that the elements in the input lists are accessed once.

3.3.2 Multi-list-tree-preceding

In this section, we provide a discussion on next join algorithm: `Multi-list-tree-preceding` that takes PList and FList as input; and outputs all (p, f) pairs such $p \in$ PList and $f \in$ FList and p is a *preceding-sibling* of f. The resultant pairs are produced in *Preceding* order.

As with the formal semantics [8] of XPath, each XPath axis has natural inverses: descendant = ancestor^{-1}, preceding = following^{-1} and preceding-sibling = following-sibling^{-1}. Taking advantage of the above property, the algorithm for *preceding-sibling* axis can be developed by traversing the input lists sequentially from end of the list to the beginning. This is similar to joining of nodes in the XML document tree in reverse document order i.e., the element with largest *start* is processed first. The algorithm shown in Figure 6 can be easily extended to process *preceding-sibling* axis with out loss of linearity in complexity during run time.

Figure 8 shows the algorithm for solving queries containing *preceding-sibling* axis. During the process of join, we maintain a *sibling-list* of *Following* nodes (nodes from FList) that join with the appropriate nodes in the PList. The anatomy of the algorithm is as follows: p and f act as cursors that start at the end of the input lists and sequentially traverse to the beginning of the lists. During the join, when a node f from FList occurs, we check whether or not the *sibling-list* at *f.level* is empty (line 5). If it is empty, we update the list's *id* with *f.parentId*, and then add f to the empty *sibling-list*, SList[*f.level*] (lines 14-16). Otherwise, in line 6, we check if SList[*f.level*].*id* equals to *f.parentId*; if

```
Algorithm: Multi-list-tree-Preceding ( PList, FList )
//PList: list of Preceding elements sorted on their start values
//FList: list of Following elements sorted on their start values
//SList: list of elements that are under common parent

1:  p = last(PList);
2:  f = last(FList);
3:  while (not beginning of PList or FList) do
4:    if (f.start > p.start) then
5:      if (SList[f.level] ≠ ∅) then
6:        if (SList[f.level].Id == f.parentId) then
7:          append f to SList[f.level];
8:        else
9:          SList[f.level].makeEmpty();
10:         SList[f.level].Id = f.parentId;
11:         add f to SList[f.level];
12:       end if
13:     else
14:       SList[f.level].Id = f.parentId;
15:       add f to SList[f.level];
16:     end if
17:     f = next element in FList before f;
18:   else
19:     if (SList[p.level] ≠ ∅) then
20:       if (SList[p.level].Id == p.parentId) then
21:         Output p with all elements in SList[p.level];
22:       else
23:         SList[p.level].makeEmpty();
24:       end if
25:     end if
26:     p = next element in PList before p;
27:   end if
28: end while
```

Figure 8: Multi-list-tree join algorithm that output pairs in the *Preceding* order

it is, we append f to the nodes in SList[*f.level*] (line 7). If it doesn't satisfy this condition, we first empty the list, then update the list's *id* with *f.parentId*, and then add node f to the empty list (lines 9-11). We continue the process with the next element in FList before f.

If a node p from PList occurs, we check whether or not SList[*p.level*].*id* equals to *p.parentId*; if it is found to be equal, we output p with all the elements appeared in the list (lines 20-21). Otherwise, we empty the corresponding list (line 23). The procedure continues with the next element in PList before p.

When the query contains multiple structural joins [13], the result of one join is passed to the subsequent join. Since the output of a join must be in the document order, we need to produce pairs either in *Following* order or *Preceding* order

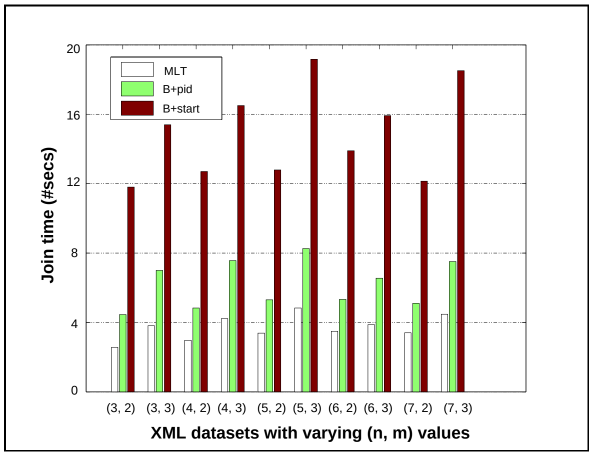

Figure 9: Join time of the query: *p/following-sibling::f* when tested on XML datasets of varying (n, m) values

in the ascending order of *start* value. But, the algorithm shown in Figure 8 produces pairs in reverse document order. The algorithm can be easily modified to produce pairs in the ascending order of *start* value. When a pair $(p,\ f)$ is produced during the join, instead of appending it to the end of the `OutputList`, we insert the pair at the beginning of the `OutputList`. So, at any instance, the `OutputList` is sorted on the ascending order of the *start*. Here we note that the result of a join is not pipelined to the subsequent join until the last pair is output.

3.3.3 *Analysis of Multi-list-tree Join Algorithms*

`Multi-list-tree` join algorithms are easy to analyze. For some element $f \in$ FList, we examine whether the *id* associated with SList[$f.level$] is same as $f.parentId$. If it satisfies, we output f with all the elements in the list. Here, we can observe that the equality check is made only with the *id* of the corresponding *sibling-list*, irrespective of the number of elements contained in it. As the time for generating output is directly proportional to the output size, we get the total join time for the algorithm as $\bigcirc(|\text{PList}| + |\text{FList}| + |\text{Output}|)$ in worst case.

4. PERFORMANCE STUDY

In this section, we explain the comparative study of our `multi-list-tree` join algorithms with `indexed-loop` join algorithms. Our results demonstrate that our join algorithms out-perform `indexed-loop` join algorithms.

4.1 XML Datasets

For our experiments, we used synthetic data so as to have control over the XML tree structure and number of siblings of each node. The generated datasets confirm to the DTD that has the following pattern: For every element p or f in the XML document tree, we have either children of pattern $(p^n,\ f)\times m$ or PCDATA, where n refers to number of p nodes and m refers to number of such $(p^n,\ f)$ patterns. For example, a pattern with $n = 3$, $m = 2$ looks like $(p,\ p,\ p,\ f,\ p,\ p,\ p,\ f)$.

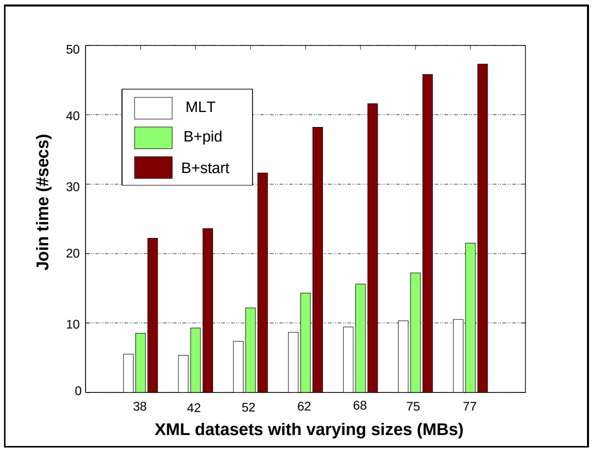

Figure 10: Join time of the query: *p/following-sibling::f* when tested on XML datasets of varying sizes with n = 6, m = 3

The datasets are generated by varying $(n,\ m)$ values. Each XML data is of size around 30 MB and approximately contains 1 million nodes. We also generated datasets with constant $(n,\ m)$ value but varying in size from 38 to 77 MB. We parse the XML documents (using GNOME XML Parser [16]) and represent each element using the numbering scheme (*start*, *end*, *parentId*, *level*). The numbers are assigned using an in-memory stack that grows to the maximum depth of the XML document tree. A `counter` is maintained and is incremented for each stack operation – *push* or *pop*. When a node is pushed on to the stack, we assign the current value of the `counter` to *start* value of the node. Similarly, when node is popped out, we assign the current value of the `counter` to the *end* value of the node.

We evaluated the join algorithms over the generated XML datasets of varying $(n,\ m)$ values and sizes. And the performance of the algorithms are compared using a sample query: *p/following-sibling::f*. We use a query with single structural join as the aim of the paper is to demonstrate improved performance for a single structural join, rather that tackle a sequence of structural joins. Appropriate join order selection may be required in the latter case.

4.2 Experimental Testbed

The experiments were conducted using Red Hat v7.3 on P-IV 1.6GHz processor with 256 MB RAM and 40GB hard disk having a disk page size of 4KB. The disk is locally attached to the system and is used to store XML data and the element lists. All these algorithms are coded in C++ and compiled using GNU C++ compiler.

4.3 Implementation Details

The main objective of our experiments is to characterize the performance of the join algorithms discussed in this paper. To evaluate our join algorithms, we implemented the `indexed-loop` join algorithms using B^+-tree index structure available in Berkeley DB [17]. For each element list, we maintained two B^+-trees: one constructed using *start* as key and other using *parentId* as key. Algorithms that use index may require making multiple scans over the elements in one of the input lists. Moreover, these algorithms require to have index over the lists and will not be able to solve queries containing multiple structural joins. For example, in query *a/b/following-sibling::c*, only the elements in *b*-list, that are children of elements in *a*-list, join with the elements in *c*-list. As there is no index available on the resultant subset of elements in *b*-list, it is not possible to effectively process subsequent joins.

We implemented the `multi-list-tree` join algorithms also by utilizing the B+-tree constructed using *start* as key; this is done to have the same overhead incurred by use of the Berkeley DB in our algorithms as well as in the `indexed-loop` join algorithms. During the processing of *following-sibling*, we sequentially traverse (using cursors available in Berkeley DB) the elements at the leaf nodes of the B^+-tree from first to last (last to first while processing the *preceding-sibling* axis). Furthermore, `multi-list-tree` join algorithms access each element in the element lists only once.

4.4 Experimental Results

This subsection details the experimental evaluation and performance plots of the join algorithms. We use the following notations to present the join algorithms: B+start and B+pid refer to `indexed-loop` join algorithms that operate on B^+-tree constructed using *start* and *parentId* as a key, respectively. MLT refers to `multi-list-tree` join algorithm that process the join using *sibling-list* at each level of the XML document tree.

We compare the performance of `multi-list-tree` join algorithms with `indexed-loop` join algorithms by operating them on different XML datasets: (i) varying (n, m) values with fixed data size and (ii) varying data sizes with fixed (n, m) value. The performance plot of join algorithms operating on varying (n, m) values is shown in Figure 9 and those operating on varying sizes is shown in Figure 10; the output cardinality is around 2 to 4 million pairs. In any instance, the performance of `multi-list-tree` join algorithms is significantly better than that of `indexed-loop` join algorithms. The improvement in the performance is due to its ability to perform join by making a single scan over elements in the input lists and never having to buffer disk pages during the process of join.

From the plots shown in Figure 9 and Figure 10, we can observe that `Btree-on-parentId` algorithm performs 2 to 3 times better than `Btree-on-start` algorithm; this is due to the lessening of number of element scans made during the process of join. As `indexed-loop` join algorithms have to access each element occurring in the element lists multiple times, we need to maintain a buffer that temporarily stores recently visited elements.

We ran these experiments several times to report the warm cache results; and the CPU times plotted are the average of many runs of these algorithms.

5. CONCLUSIONS

In this paper, we propose structural join algorithms that solve XPath queries containing sibling joins. We show that the new join algorithms, called `multi-list-tree`, take time linear in the sum of lengths of input lists. Experimental results show that our join algorithms perform better than the `indexed-loop` join algorithms for sibling joins. We notice that even the B^+-tree based index results in multiple scans over the element lists. The proposed join algorithms utilize *sibling-list*s at each level of the XML document tree; and also make a single scan over the elements in the input lists.

6. 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 Processing. In *ICDE*, pages 141-152, 2002.

[2] N. Bruno, N. Koudas, and D. Srivatsava. Holistic twig joins: Optimal XML pattern Matching. In *SIGMOD*. pages 310-321, 2002.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML Path Language (XPath) 2.0 Technical Report, *W3C Working Draft*. Available at *http://www.w3.org/TR/XPath20/. 2001.*

[4] D. Chamberlin, D. Florescu, J. Robie, J. Simon, and M. Stefenscu. XQuery: A Query Language for XML. *W3C Working Draft*. Available at *http://www.w3.org/TR/XQuery 2001.*

[5] S-Y. Chein, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zanilo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*. pages 263-274, 2002.

[6] P. F. Dietz. Maintaining Order in a Linked List. In *ACM Symposium on Theory of Computing*, pages 122-127, 1982.

[7] D. Florescu and D. Kossman. Storing and Querying XML data using RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27-34, 1999.

[8] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB*, 2002.

[9] T. Grust. Accelerating XPath Location Steps. In *SIGMOD*, pages 109-120, 2002.

[10] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ICDE*, pages 253-264, 2003.

[11] Q. Li and B. Moon. Indexing and Querying XML data for Regular Path Expressions. In *VLDB*, pages 361-370, 2001.

[12] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. J. Dewit, and J. F. Naughton. Relational Databases for Querying XML Documents : Limitations and Opportunities. In *VLDB*, pages 302-314, 1999.

[13] G. V. Subramanyam and P. Sreenivasa Kumar. Efficient Processing of Multiple Structural Join Queries. In 21st BNCOD, 2004.

[14] 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.

[15] C. Zhang, J. Naughton, D. Dewit, Q.Luo, and G. Lohman. On supporting containment queries in Relational Database Management systems. In *SIGMOD*, pages 425-436, 2001.

[16] Available at *http://www.xmlsoft.org.*

[17] Available at *http://www.sleepycat.com.*