Skipping Strategies for Efficient Structural Joins

Franky Lam William M. Shui Damien K. Fisher Raymond K. Wong
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{flam,wshui,damienf,wong}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0320
June 2003

Abstract

The structural join is considered a core operation in processing and optimizing XML queries. Various techniques have been proposed for efficiently finding structural relationships between a list of potential ancestors and a list of potential descendants. This paper presents a novel algorithm for efficiently processing structural joins. Moreover, previous work which performs well usually relies on external index structures such as a B-tree, which increases both the storage and memory overheads. Our proposal in this paper does not require any such data structures, and hence can be easily implemented and incorporated in any existing system. Experiments show that our method significantly outperforms previous algorithms.

1 Introduction

In recent years XML [3] has emerged as the standard for information representation and exchange on the Internet. However, finding efficient methods of managing and querying large XML documents is still problematic and poses many interesting challenges to the database research community.

XML documents can essentially be modeled as ordered trees, where nodes in the ordered tree represent the individual elements, attributes and other components of an XML document. The pre-order traversal upon the tree gives the *document ordering* of the XML document. Recently proposed XML query languages such as XPath [17] and XQuery [18], which have been widely adopted by both research and commercial communities for querying XML documents, heavily rely upon regular path expressions for querying XML data. For example, consider the MEDLINE [14] database as an example XML document. The regular path expression //`DateCreated`//`Month` returns all <`Month`> elements that are contained by <`DateCreated`>. Without any indexes to the document, the typical approach to evaluating //`DateCreated`//`Month` would require a full scan of the entire XML database, which can be very costly if the document is large.

Recently, the *structural join*, which involves finding structural relationships between a list of potential ancestor nodes and a list of potential descendant nodes, has been proposed. Structural joins are now considered a core operation in processing and optimizing XML queries. Various techniques have been proposed recently for efficiently finding the structural relationships between a list of potential ancestors and a list of potential descendants. Most of these proposals rely on some kind of database index structures such as B^+Tree. These index structures will increase their resources requirement (e.g., memory consumption) and maintenance overhead (e.g., for updates). Furthermore, most of them count on numbering schemes such that they incur significant relabeling costs during data updates.

Instead of improving the performance of the state of the art structural join based on external index structures, this paper proposes simple and yet effective ways of skipping unmatched nodes during the structure join processing. The key contributions of this paper are summarized as follows:

1. We propose an improvement to the current state of the art stack based structural joins [1] [1] based on various skipping strategies. In contrast to other work, our proposed extension does not require any external indexes such as B-trees, and hence imposes less overhead on the underlying database system.

2. Our proposed method does not employ any indexes such as B^+Tree and its entire operation cost is linearly proportional to size of the query output. Hence it can be extended to support XML stream data.

3. We present extensive experimental results on the performance of our proposed algorithms, using both real-world and synthetic XML databases.

4. We show experimentally that our approach can outperform the stack-based structural join algorithms by several orders of magnitude.

5. We discuss how updates can affect structural join processing and how our approach can reduce the negative side-effects of updates on XML databases.

6. Finally, we discuss the differences between the preorder/postorder and start/end approaches to maintaining ancestor-descendant information in XML databases.

The rest of this paper is organized as follows. Section 2 gives an overview of the background to the problem and some existing related work. We present our improvements to existing structural join

[1]The algorithm of [1] will be referred to as the STJ-D algorithm hereafter.

algorithms in Section 3. In Section 4, we describe the setup of our experiments and we also compare our results with some existing schemes for structural joins. In Section 5, we discuss the labeling scheme that we used for our structural joins. Finally, Section 6 concludes the paper.

2 Background and Related Work

XML data is generally modelled as a tree structure where elements, attributes and data are represented as nodes of the tree. Within this tree, parent-child and ancestor-descendant relationships represent the nesting of elements within the corresponding XML document. Querying XML data frequently involves the determination of the containment relationship between data nodes; for example, during the evaluation of a path expression a structural join may be used to determine whether an element A is the ancestor node of an element B. Thus, in order for structural join algorithms to operate efficiently, the database should be represented in a way which allows the structural relationship of nodes to be determined in close to constant time. This section describes different approaches for efficiently determining the ancestor-descendant relationship between two nodes. We also review related work on structural joins that make use of these schemes.

2.1 Structural Joins

Recently, several new algorithms, *structural join* algorithms, have been proposed for finding sets of document nodes that satisfy the ancestor-descendant relationship with another set of document nodes. Various approaches have been proposed using traditional relational database systems [10, 19] and on XML query engines such as proposed in [13].

The current state of the art structural joins on XML data is described in [1]. It takes as input two lists of elements, both sorted with respect to document order, representing the list of ancestors (AList[2]) and the list of descendants (DList[3]). The basic idea of the algorithm is to do a merge of the two lists to produce the output, by iterating through them in document order. While it is iterating through the two lists, it determines the ancestor-descendant relationship between the current top of a stack, which is maintained during the iteration, and the next node in the merge. Based on this and the manipulation of the stack, it produces the correct output. The cost of this approach is $O(|AList|+|DList|+|Output|)$.

More recent work extended this approach for better speed performance. For example, the XML Region Tree (XR-Tree) approach to index document structure on disk [11], uses a variant of $B^{+}Tree$ with different index key entry and lists to maintain the ancestor-descendant relationship between nodes. It then uses the stack-tree based join algorithm to carry out structural joins. The amortized I/O cost for inserting and deleting nodes for 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 of one displacement of a stabbed element. Although this approach can support structural joins of XML data by using the stack-tree based join, determing ancestor-descendant relationship between the two nodes is not constant. Therefore, for large ancestor and descendant node sets it may not be as efficient as the original STJ-D algorithm. Also, any large set of random updates requires frequent updates of large parts of the XR-tree. Therefore, maintaining the index for a large, changing XML database can be costly.

In [4], the stack-tree join algorithm was extended to match more general selection patterns on XML data tree. The work done by [1, 4, 5] are very related to this paper. For instance, the B^{+}algorithm proposed in [5] speeds up the stack-tree based join algorithm by using a combination of pre-built indexes such as $B^{+}Tree$ and R-Tree. It utilizes $B^{+}Tree$ indexes built on the element start-tag positions.

[2]Ancestor node list and AList are interchangeable hereafter.

[3]Descendant node list and DList are interchangeable hereafter.

Hence it can use $B^{+}Tree$ range queries to skip descendants that do not match during the structural join. However, these approaches are not effective in skipping ancestor nodes, as stated in [11]

2.2 Numbering Schemes

Apart from the index based schemes that were mentioned above, numerous work has been done on using numbering schemes to support queries on ancestor-descendant relationships. The following discusses them in more detail.

Dietz and Sleator worked on maintaining order in a linked list [8]. They proposed algorithms which permitted constant time queries on the relative order of nodes in a list, with only a constant time overhead on insertions and deletions in the list. This supported earlier work done on solving the *ancestor query problem* by comparing the relative preorder and postorder of two nodes [7].

More recently, a more elegant approach has been proposed [2], which obtains the same performance as Dietz and Sleator's algorithm. The maintenance of both the preorder and postorder on an XML document corresponds to the order maintenance problem, and hence this result gives the best possible theoretical bounds on our problem. Additionally, there is an upper bound on the size of the database for which the results hold and [2] estimates it to be at approximately $430,000$ elements for a particular parameter selection. Therefore, this algorithm is only an incomplete answer to the question of document ordering in large databases, where the number of nodes can easily run into the millions. However, the paper did not focus on ancestor query problem. Furthermore, the order maintenance problem forcus on maintaining the order between two nodes. It is not directly related to skipping unmatched nodes in structural joins.

Extensive research has been done on inverted indices [15] for Information Retrieval (IR) systems. A recent work [19] showed that we can use an inverted index to solve the containment queries of XML data nodes. The inverted index data structure maps text words of XML documents in a T-index and elements in an E-index such that elements are mapped to inverted lists. Occurrences of a word or an element are recorded in inverted lists, with each occurrence indexed by its $(DocID, Start : End, Level)$, where *DocID* is the document number, *Start* is the position of the word [4], *End* is the position where this word ends; and *Level* is the depth of the data node. This information is sufficient to compute ancestor-descendant relationships. In practice however, there are always frequent, randomly distributed inserts, deletes and updates of XML data. Any changes to the database will require the majority of the inverted index to be re-calculated. Therefore, this approach can be costly for maintaining an inverted index for a non-static XML database.

3 Skip-Joins

This section presents our algorithms to skip unmatched nodes for structural joins. It also describes various strategies that can be used for skipping, which lead to different performance outcomes. In order to present our algorithms in a meaningful way, we first classify all structural joins into three classes. Each class of structural joins has different applications for query optimization, and are sufficiently different that they should be optimized separately.

1. **Descendant Join (D-Join):** the first type of structural join filters a set of descendant nodes by selecting only those nodes that have an ancestor within the set of potential ancestors. For example, the query a//b should return a node set $R_{\mathcal{D}} = \{d \in R_{\mathcal{D}} \,|\exists a \in \mathcal{A}$ such that d is a descendant of $a\}$.

[4]Word and element will be used interchangeably hereafter

2. **Ancestor Join (A-Join):** this type of structural join filters a set of ancestor nodes by selecting only those nodes that have a descendant within the set of potential descendants. For example, the query `a//b` should return a node set $R_{\mathcal{A}} = \{a \in R_{\mathcal{A}} \mid \exists d \in \mathcal{D}$ such that a is an ancestor of $d\}$.

3. **Ancestor-Descendant Join (AD-Join):** the third type of structural join returns the set of ancestor-descendant node pairs. For example, the query `a[.//b=.//c]` may be evaluated by performing a structural join on the the sub-expression `.//b=.//c`, which would then make use of the set $R_{\mathcal{AD}} = \{\langle a, d\rangle \mid \forall a \in \mathcal{A}, \forall d \in \mathcal{D}$ such that a is an ancestor of $d\}$.

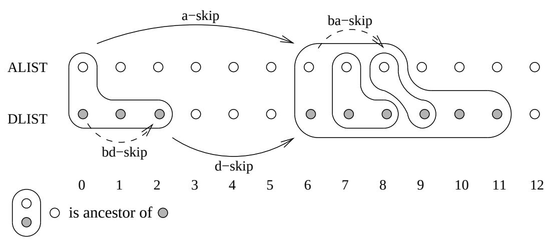

Figure 1: Possible skipping strategies during a structural join

The B^+ algorithm proposed in [5] suggested that the-state-of-the-art STJ-D algorithm proposed in [1] has the disadvantage of having to scan through the entire ancestor list for the join operation, and hence in some cases unnecessarily scans through ancestor nodes that do not contain any nodes in the descendant list. A similar phenomenon can occur during the scanning of the descendant list. Their solution to this was to use a B^+Tree, which requires a prebuilt index system for the database. In this section, we introduce some extensions to the STJ-D algorithm by introducing a skipping mechanism to skip ancestor and descendant nodes that do not match the structural pattern, and hence which may be safely ignored during the structural join.

Figure 1 represents a particular instance of a structural join. Depending on the type of query, we can utilize different skipping mechanisms to optimize the join. The circled regions denote the ancestor-descendant relationship between nodes. For example, $\{d_0, d_1, d_2\}$ is the set of descendants of a_0. For AD-Joins, the optimal scenario is if we can determine $R = \{\langle a, d\rangle \mid a \notin \{a_1 \ldots a_5, a_9 \ldots a_{12}\}, d \notin \{d_3 \ldots d_5, d_{12}\}\}$ while scanning $\mathcal{A}$ and $\mathcal{D}$. The a-skip arrow and d-skip arrow in the figure show the nodes which are not included in the result set of an AD-Join; hence, the structural join algorithm should try to minimize traversal of those nodes if possible. For A-Joins (respectively D-Joins), in the optimal case we should further skip all the matched descendants $\{d_1, d_2, d_8, d_{10}, d_{11}\}$ (respectively ancestors $\{a_7, a_8\}$). For example, as soon as we can determine that d_0 is a descendant of a_0, we do not need to traverse d_1 and d_2, because they only match with a_0. Similarly, for D-Joins, the traversal of a_7 and a_8 should be avoided since a_6 is their common ancestor, and so descendants of a_7 and a_8 are also descendants of a_6, thus skipping a_7 and a_8 will not affect the result.

Of course, in order to skip nodes we must make the assumption that we can perform these skips in constant time. We note that this assumption is not necessary for previous work such as that of [1]. However, we believe that very frequently the node sets being joined will be stored in array-like structures in memory or on disk. This is because even for relatively large data sets such as DBLP [6], the node sets remain only a few megabytes in size, and hence are easily manipulated as arrays.

The pseudo-code for the STJ-D algorithm is shown in Algorithm 1; this algorithm is used later as the control for our experiments. However, we have modified the algorithm from its original presentation such that it uses a preorder and postorder labelling scheme to determine ancestor and descendant relationships between nodes. We use this labelling scheme for data update maintainence instead of using the traditional $(StartPos : EndPos, Level)$ approach. This is further discussed in section 5.

Algorithm 1 Slightly modified Stack-tree based structural join (STJ-D) proposed in [1]. *(NB: All algorithms in this paper are simplified for ease of presentation, e.g. boundary cases are omitted and all boolean operations returns false if a particular required element does not exist or out of range.)*

```
STACK-TREE-DESC($\mathcal{A}, \mathcal{D}$)
 1 $a \leftarrow 0, d \leftarrow 0, \mathcal{R} \leftarrow \emptyset, Stack \leftarrow \emptyset$
 2 while $d < |\mathcal{D}| \wedge a < |\mathcal{A}| \vee |Stack| > 0$ do
 3    if FOLLOWING(TOP($Stack$), $\mathcal{A}[a]$) $\wedge$
 4          FOLLOWING(TOP($Stack$), $\mathcal{D}[d]$) then
 5       POP($stack$)
 6    elif PREORDER($\mathcal{A}[a]$) $<$ PREORDER($\mathcal{D}[d]$) then
 7       PUSH($\mathcal{A}[a], stack$)
 8       $a \leftarrow a + 1$
 9    else
10       APPEND(($s, \mathcal{D}[d]$), $\mathcal{R}$), $\forall s \in Stack$
11       $d \leftarrow d + 1$
12    end if
13 end while

FOLLOWING($n, f$)
// Returns true iff $f$ belongs to following axis of $n$.
 1 return PREORDER($f$) $>$ PREORDER($n$) $\wedge$
 2       POSTORDER($f$) $>$ POSTORDER($n$)
ANCESTOR($d, a$)
// Returns true iff $a$ belongs to ancestor axis of $d$.
 1 return PREORDER($d$) $>$ PREORDER($a$) $\wedge$
 2       POSTORDER($d$) $<$ POSTORDER($a$)
```

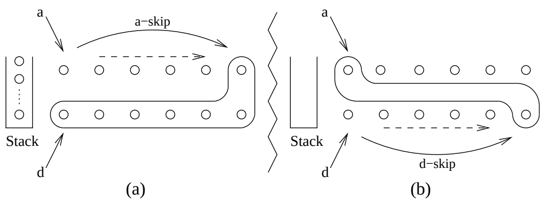

Figure 2: Skipping scenarios of AD-Join

3.1 Skip-Join for Ancestor-Descendant Join

Here, we propose an alternative stack-tree based structural join algorithm on two input lists AList and DList (both sorted in document order). Our approach is to make the assumption that we can skip quickly (as discussed previously), and then to use this assumption by utilizing a skipping mechanism during the traversal of $\mathcal{A}$ and $\mathcal{D}$. The basic idea is that during the structural join, whenever we advance the cursor of $\mathcal{A}$ we call the A-SKIP function to search for the next node $a' \in \mathcal{A}$, such that it a' is either an ancestor of the current descendant node d, or follows d in document order. Similarly, whenever we advance the cursor of $\mathcal{D}$, we call the D-SKIP function to search for the next node $d' \in \mathcal{D}$, such that d' is either a descendant of the current ancestor node a, or follows a in document order.

In Figure 2(a), which uses the original STD-J algorithm, all nodes under the dashed arrow need to be traversed by pushing them onto the stack and immediately popping them in the next iteration. By using an A-SKIP function, we try to minimize the number of lookups of these unnecessary nodes during list traversal and reducing the number of nodes pushed and popped from the stack. However, node a' may not necessarily be an ancestor of node d, as it may follow d in document order. Similarly, in Figure 2(b), by using the original STD-J algorithm, we again need to traverse all nodes above the dashed arrow. Hence, function D-SKIP is used to try to minimize the traversal of the descendant list.

Algorithm 2 Structural joins that return ancestor-descendant node pairs (AD-Join).

```
SKIP-JOIN-AD($\mathcal{A}, \mathcal{D}$)
 1 $a \leftarrow 0, d \leftarrow 0, \mathcal{R} \leftarrow \emptyset, Stack \leftarrow \emptyset$
 2 while $d < |\mathcal{D}| \wedge a < |\mathcal{A}| \vee |Stack| > 0$ do
 3     if FOLLOWING(TOP($Stack$), $\mathcal{A}[a]$) $\wedge$
 4             FOLLOWING(TOP($Stack$), $\mathcal{D}[d]$) then
 5         POP($stack$)
 6     elif PREORDER($\mathcal{A}[a]$) $<$ PREORDER($\mathcal{D}[d]$) then
 7         PUSH($\mathcal{A}[a], stack$)
 8         $a \leftarrow$ A-SKIP($a, \mathcal{D}[d], \mathcal{A}$)
 9     else
10         APPEND($(s, \mathcal{D}[d]), \mathcal{R}$), $\forall s \in Stack$
11         if $|Stack| > 0$ then
12             $d \leftarrow d + 1$
13         else
14             $d \leftarrow$ D-SKIP($d, \mathcal{A}[a], \mathcal{D}$)
15         end if
16     end if
17 end while
```

The algorithm is listed in Algorithm 2.

We will also, at times, need to use two additional skipping functions, BA-SKIP and BD-SKIP. Thesse functions are used when we can skip nested ancesetors or descendants, a situation which occurs as described previously during A-Joins and D-Joins. The definitions of the functions A-SKIP, D-SKIP, BA-SKIP and BD-SKIP have not yet been given, because they can vary according to the skipping strategy chosen. We will discuss several possible strategies later in this paper.

It should be pointed out that although it is possible to skip nodes in $\mathcal{D}$ even when the stack is not empty, the performance gain may not cover the penalty of the overhead. This is because, in practice, real world XML trees have very shallow depth, and hence the number of skippable nodes within nested regions are generally small.

3.2 Skip-Join for Ancestor Structural Join

Many XML queries require the efficient filtering of ancestor nodes. For example, the query `a//b[.//c]` returns a set of `b` nodes which all have an ancestor `a` and a descendant `c`. If we use the STJ-D algorithm to process this query, we have to first join `a//b`, then `b//c` and finally merge the two joins together. However, if we have an ancestor filtering algorithm, it can return a smaller set of `b` nodes that are ancestors of `c`. We then feed this smaller `b` set as the new $\mathcal{D}$ for joining with `a` nodes. Then, we can take advantage of our previously described skip-join algorithm for descendant filtering, where it will perform better with smaller descendant sets.

To further improve the performance of skip-joins on ancestor structural joins, we can take advantage of knowing that only ancestor nodes are wanted, and hence when the stack is not empty, we can skip all matched descendant nodes using BD-SKIP, because these nodes are not needed to increase the size of the result set. The detailed steps are described in Algorithm 3.

3.3 Skip-Join for Descendant Structural Join

For descendant structural joins, we do not need to keep the stack of ancestor nodes, as keeping only the top most ancestor will yield the same result set. As soon as we push any nodes onto the stack, we can immediately use BA-SKIP to skip all nodes in $\mathcal{A}$ until a' follows the node in the stack in document

Algorithm 3 Structural joins that return only matched ancestor nodes (A-Join).

```
SKIP-JOIN-A($\mathcal{A}, \mathcal{D}$)
$a \leftarrow 0, d \leftarrow 0, \mathcal{R} \leftarrow \emptyset, Stack \leftarrow \emptyset$
**while** $d < |\mathcal{D}| \wedge a < |\mathcal{A}| \vee |Stack| > 0$ **do**
    **if** FOLLOWING(TOP($Stack$), $\mathcal{A}[a]$) $\wedge$
          FOLLOWING(TOP($Stack$), $\mathcal{D}[d]$) **then**
        POP($stack$)
    **elif** PREORDER($\mathcal{A}[a]$) $<$ PREORDER($\mathcal{D}[d]$) **then**
        PUSH($\mathcal{A}[a], stack$)
        $a \leftarrow$ A-SKIP($a, \mathcal{D}[d], \mathcal{A}$)
    **else**
        APPEND($s, \mathcal{R}$), $\forall s \in Stack$
        $d \leftarrow$ D-SKIP($d, \mathcal{A}[a], \mathcal{D}$)
        **if** $|Stack| > 0$ **then**
            $d \leftarrow$ BD-SKIP($d$, TOP($Stack$), $\mathcal{D}$)
        **else**
            $d \leftarrow d + 1$
        **end if**
    **end if**
**end while**
```

order. Algorithm 4 shows the pseudo-code for this approach. We expect this type of structural join to perform well regardless of the size of AList and DList.

3.4 Skipping Strategies

Algorithm 5 describes the semantics of each skipping function. The goal of all these functions is to skip as many of the unmatched nodes as possible. However, for each of these skipping functions, different skipping strategies can be applied, each of which will result in different performance for the overall algorithm. For instance, one may find that it is more effective to skip nodes using a binary search when using A-SKIP, but better to skip nodes using an exponential technique when using BD-SKIP. We will investigate this in the experimental section of this paper. In this section, we propose and describe different skipping strategies in detail. All examples below assume we are discussing skipping strategies for the A-SKIP function, but each of them can be similarly applied to other skipping functions.

3.4.1 Binary Skipping

Here we propose a skipping strategy which uses a simple binary search, as described in Algorithm 6. As is illustrated in Figure 3, the function hops through $AList$ trying to find a node a such that a is an ancestor of both the current top node in the stack s and the current descendant node d and PREORDER(a) does not match the appropriate structural pattern. If no node is found, it returns the empty set and the join function stops scanning through the DList. We believe this approach can be efficient for processing queries such as //`Month[text()="03"]`; in this case, even if there exists a large set of `Month` elements, there may only be a small subset of them that matches the predicate. This yields a large ancestor to descendant node ratio and in most cases, large sections of the ancestor node list need not be visited at all.

3.4.2 Exponential Skipping

Since binary skipping uses a binary search approach, in the worst case it can take $\log n$ skips to locate the next AList node that matches the structural pattern. Thus, the worst case of the binary skipping strategy is if most nodes in the ancestor node list are matched. In this case, then every call to

Algorithm 4 Structural joins that return only matched descendant nodes (D-Join).

```
SKIP-JOIN-D($\mathcal{A},\mathcal{D}$)
 1 $a \leftarrow 0, d \leftarrow 0, \mathcal{R} \leftarrow \emptyset, s \leftarrow \phi$
 2 while $d < |\mathcal{D}| \wedge a < |\mathcal{A}| \vee s \neq \phi$ do
 3    if FOLLOWING($s, \mathcal{A}[a]$) $\wedge$
 4          FOLLOWING($s, \mathcal{D}[d]$) then
 5       $s \leftarrow \phi$
 6    elif PREORDER($\mathcal{A}[a]$) $<$ PREORDER($\mathcal{D}[d]$) then
 7       if $s \neq \phi$ then
 8          $a \leftarrow$ BA-SKIP($a, s, \mathcal{A}$)
 9       else
10          $s \leftarrow \mathcal{A}[a]$
11       end if
12       $a \leftarrow$ A-SKIP($a, \mathcal{D}[d], \mathcal{A}$)
13    else
14       APPEND($\mathcal{D}[d], \mathcal{R}$)
15       if $s \neq \phi$ then
16          $d \leftarrow d + 1$
17       else
18          $d \leftarrow$ D-SKIP($d, \mathcal{A}[a], \mathcal{D}$)
19       end if
20    end if
21 end while
```

Algorithm 5 Properties (and hence the algorithm) for each of the skipping strategies: A-SKIP, D-SKIP BA-SKIP and BD-SKIP

```
A-SKIP($a, d, \mathcal{A}$)
 1 return MIN($\{a' \in \mathcal{A} | a' > a$, ANCESTOR($d, a'$) $\vee$ FOLLOWING($d, a'$)$\}$)
D-SKIP($d, a, \mathcal{D}$), BD-SKIP($d, a, \mathcal{D}$)
 1 return MIN($\{d' \in \mathcal{D} | d' > d$, ANCESTOR($d', a$) $\vee$ FOLLOWING($a, d'$)$\}$)
BA-SKIP($a, s, \mathcal{A}$)
 1 return MIN($\{a' \in \mathcal{A} | a' > a$, FOLLOWING($s, a'$)$\}$)
BD-SKIP($d, s, \mathcal{D}$)
 1 return MIN($\{d' \in \mathcal{D} | d' > d, \neg$ANCESTOR($d', s$)$\}$)
```

the skipping function would require approximately $\log n$ skips to find the next node. Here, we propose an exponential skipping strategy, which tries to avoid this worst case scenario. The exponential skipping strategy first skips through the ancestor list using exponentially increasing gaps, for example, 1, 2, 4, 8, 16, etc. When it over-shoots the target node, we then switch to binary search with the $high$ and low boundaries of the search set to the current and previous hop position. The pseudo-code is described in Algorithm 7.

Figure 4 illustrates how the exponential skipping strategy augments the binary skipping by using slow start to increase the gap size exponentially until it over-skips past the next ancestor node. Since the next gap size is based on the past observed number of skipped nodes, the number of over-skipped

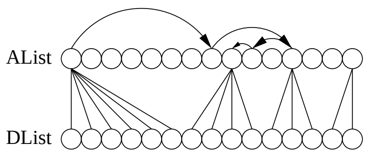

Figure 3: Binary skipping strategy of A-SKIP.

Algorithm 6 Binary skipping of ancestor nodes

```
A-BINARY-SKIP($min_{\mathcal{A}}, max_{\mathcal{A}}, d, \mathcal{A}$)
while $min_{\mathcal{A}} \leq max_{\mathcal{A}}$ do
    $\bar{x} \leftarrow \lfloor \frac{min_{\mathcal{A}} + max_{\mathcal{A}}}{2} \rfloor$
    if POSTORDER($\mathcal{A}[\bar{x}]$) $>$ POSTORDER($d$) then
        if POSTORDER($\mathcal{A}[\bar{x} - 1]$) $<$ POSTORDER($d$) then
            return $\bar{x}$
        else
            $max_{\mathcal{A}} \leftarrow \bar{x} - 1$
        end if
    else
        if PREORDER($\mathcal{A}[\bar{x}]$) $>$ PREORDER($d$) then
            $max_{\mathcal{A}} \leftarrow \bar{x} - 1$
        else
            $min_{\mathcal{A}} \leftarrow \bar{x} + 1$
        end if
    end if
end while
return $|\mathcal{A}|$
```

Algorithm 7 Skipping at increasing interval

```
A-EXPONENTIAL-SKIP($a, d, \mathcal{A}$)
$min_{\mathcal{A}} \leftarrow a + 1$, $max_{\mathcal{A}} \leftarrow |\mathcal{A}| - 1$, $\delta \leftarrow 1$, $\bar{x} \leftarrow min_{\mathcal{A}}$
while $\bar{x} < |\mathcal{A}|$ do
    if POSTORDER($\mathcal{A}[\bar{x}]$) $<$ POSTORDER($d$) then
        $min_{\mathcal{A}} \leftarrow \bar{x}$, $\bar{x} \leftarrow \bar{x} + \delta$, $\delta \leftarrow 2\delta$
    else
        $max_{\mathcal{A}} \leftarrow \bar{x}$
        break
    end if
end while
return A-BINARY-SKIP($min_{\mathcal{A}}, max_{\mathcal{A}}, d, \mathcal{A}$)
```

nodes will be at at most equal to the size of the gap. Therefore, no matter how many nodes we have to skip, the slow start nature of this type of skipping strategy guarantees that in the worst case only one extra traversal is executed, and that this worst case happens when the number of nodes we must skip is either one or three.

3.5 Skipping For Streaming Data

Chien *et al* [5] mention that it is not possible for any pre-built indexes to perform faster than sequential scan based structural join algorithms for streaming data. This is simply because they cannot send any feedback to the producer modules. As we have shown, our algorithms do not use any pre-

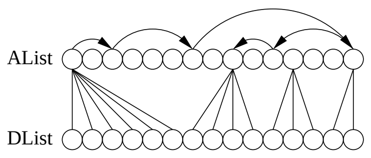

Figure 4: Exponential skipping strategy of A-SKIP.

Table 1: Experimental data set

Name	# of Elements	Size(MB)	Depth
DBLP	3,803,281	160	6
MEDLINE	2,768,743	130	7
XMark	2,921,323	204	12

Table 2: Notations for algorithms

Notation	Algorithm
STJ-D-Join	Stack-Tree-Join-Desc [1]
AD-Join$_e$	Skip-Join-AD with exponential skipping strategy
A-Join$_e$	Skip-Join-A with exponential skipping strategy
D-Join$_e$	Skip-Join-D with exponential skipping strategy
AD-Join$_b$	Skip-Join-AD with binary skipping strategy
A-Join$_b$	Skip-Join-A with binary skipping strategy
D-Join$_b$	Skip-Join-D with binary skipping strategy

built indexes for node skipping. Therefore, we can easily adapted our techniques to suit the on the fly join strategies needed for streaming data, Of based on the assumption that the incoming streams are in document order. For processing streaming data, we set a fixed buffer size for the stream input, and page size proportional to the buffer size, thus simulating the same environment we would normally have for skip-joins. In the event the current position is under the high boundary, we just load in more pages from the buffer pool, until it passes the buffer size. Then, we set the current position to the buffer size and do a sanity check on whether we have skipped pass the desired ancestor or descendant node. If not, we flush the buffer and load in more data from the stream.

4 Experimental Results

In this section, we present our experimental results on the performance of structural join algorithms on both real-world and synthetic XML data sets. We compare the performance of all join algorithms proposed in this paper with the original Stack-Tree-Desc (STJ-D) described in [1]. We will then discuss the impact of updating data on our approach in the next section.

4.1 Experimental Setup

The experiments were carried out on a machine with dual Intel Itanium 800MHz processors, 1 GB of RAM, and a 40 GB SCSI hard-drive. The machine ran the Debian GNU Linux operating system with the 2.4.20 SMP kernel.

The data set for our experiments consisted of the data sets from DBLP [6] and MEDLINE [14], and a data set randomly generated by XMark [16]. The statistics of the data sets used for the experiments are detailed in Table 1. Table 2 summarizes the join algorithms to be compared in our experiments and their shorthand notations, which are referred to in this section.

We implemented the join algorithm using the XPath processor from the SODA XML database engine (available from `http://www.sodatech.com`). For the purpose of maintaining control over the experiment, we disabled all database indexing and we implemented our join algorithm and

Table 3: Document and query expression used for experiments

Query#	Database	Query	Output Size (# of nodes)
A1	DBLP	`//dblp`	1
A2	DBLP	`//article`	128,533
A3	DBLP	`//inproceedings`	240,685
A4	DBLP	`/*/*/*`	3,424,646
A5	MEDLINE	`//MedlineCitation`	30,000
A6	XMark	`//listitem`	106,508
A7	XMark	`//keyword`	122,924
A8	XMark	`//bold`	125,958
D1	DBLP	`//title[.="The Asilomar Report on Database Research."]`	1
D2	DBLP	`//author[.="Jeffrey D. Ullman"]`	227
D3	DBLP	`//author`	820,037
D4	DBLP	`/*/*`	375,225
D5	DBLP	`//sup`	1,155
D6	DBLP	`/*/*/*/*/sup`	50
D7	MEDLINE	`//Year`	92,624
D8	MEDLINE	`//Year[.="2000"]`	5,426
D9	XMark	`//listitem`	106,508
D10	XMark	`//keyword`	122,924
D11	XMark	`//bold`	125,958

the STD-J algorithm using exactly the same code base in C. For each of the experiments, we scanned the database to filter out all elements which do not fit the required element name before the structural join. Both the AList and the DList along with their ordering information were stored in memory and no swapping to disk was performed throughout the experiment. We only measured the time spent on the structural join algorithm itself.

We defined a set of XPath expressions that capture various access patterns, which are listed in Table 3, along with the number of nodes that satisfy each XPath expression. Our experiments joined pairs of the result sets listed in this table together using a structural join. For example, we used the expression $A1//D1$ (as defined in the table) to compute the result of the path expression `//dblp//title[.="The Asilomar Report on Database Research."]`.

4.2 Results and Observations

In this section, we compare the performance of different types of structural joins using our proposed skip-join algorithms against the existing STJ-D join algorithm. Each join query is performed on the AList and DList nodes using an STJ-D algorithm and our proposed skip-join algorithms (i.e., AD-Join, A-Join and D-Join). The full results are presented in Table 4. Columns $|\mathcal{A}|$ and $|\mathcal{D}|$ give the size of the two lists, AList and DList, being joined. Column $\mathcal{R}$ gives the size of the output of join operations.

As we have mentioned earlier in the paper, there are three different types of structural joins for XML data: AD-Join returns ancestor-descendant node pairs, A-Join returns matching ancestor nodes only and D-Join returns matching descendant nodes only. For example in Q1, column A-Join$_e$ gives the time it took to execute the query `article[.//title[.="The Asilomar Report on Database Research."]]` using the exponential skip strategy on an ancestor only structural join. Column D-Join$_e$ gives the time it took to execute `//article//title[.="The Asilomar Report on Database Research."]`, but this time returning only descendant nodes. Column AD-Join$_e$ again gives the time of a join on the same query using an exponential skip strategy, but this time for a descendant only join.

Table 4: Runtime for different structural joins on different queries

			$(\# \, of \, nodes)$			(μs)						
Q#	$\mathcal{A}$	$\mathcal{D}$	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{R}\rvert$	STJ-D	AD-Join$_e$	A-Join$_e$	D-Join$_e$	AD-Join$_b$	A-Join$_b$	D-Join$_b$
Q1	A2	D1	128,533	1	1	66,518	131	139	145	117	122	118
Q2	A3	D2	240,685	227	116	119,747	1,197	1,224	1,186	2,359	2,434	2,391
Q3	A3	D3	240,685	820,037	557,868	450,257	470,990	357,005	313,501	815,160	1,528,636	1,485,116
Q4	A1	D4	1	375,225	375,225	197,807	200,628	224	169,168	205,983	235	167,333
Q5	A4	D5	3,424,646	1,155	1,155	1,754,825	14,374	14,501	13,984	28,833	29,855	29,504
Q6	A4	D6	3,424,646	50	50	1,742,093	2,796	2,943	2,806	3,237	3,246	3,231
Q12	D1	A2	1	128,533	0	44,349	331	351	348	309	380	381
Q7	A5	D7	30,000	92,624	92,624	72,243	75,152	76,683	48,895	114,355	152,286	147,618
Q8	A5	D8	30,000	5,426	5,426	21,567	8,137	7,776	6,823	20,762	26,058	25,253
Q9	A6	D9	106,508	106,508	39,244	77,193	85,852	84,167	63,065	180,529	187,655	109,155
Q10	A8	D10	125,958	122,924	6,636	117,640	103,772	102,221	98,883	337,705	342,628	219,683
Q11	A7	D11	122,924	125,958	7,485	116,578	103,638	101,965	99,464	342,435	338,975	218,537

For queries Q1, Q2, Q5, Q6 and Q8, the reason for the superior performance of the skip-join variants over the STJ-D join is because of their ability to skip through the ancestor node list quickly to filter out unmatched ancestor nodes. There is a close correlation in the performance speedup of skip joins with the $\frac{|\mathcal{A}|}{|\mathcal{D}|}$ ratio. Q12 is a special case where the result of a structural join is empty — in this case, the STJ-D algorithm is outperformed by our skip-join algorithms by two orders of magnitudes. This is again due to the effect of skipping descendant nodes.

However, for the rest of the queries listed in Table 4, the performance of skip-joins are only comparable to STJ-D join algorithm; this is due to the fact that the number of unmatched nodes is small. In the majority of cases, however, our skip-joins still outperform the STJ-D join algorithm. This comparable behavior can be attributed to two factors:

1. The input lists (both the ancestor and descendant lists) for these queries have approximately equal cardinalty, and the number of nodes in the result set is large. This means that the number of mismatched nodes is low, and hence the chance to have a large region that can be skipped is also small.

2. When there are large numbers of common nodes between A and D, that is, the two iterators walk in parallel rather than the "optimal" (for the skip strategies) zig-zag iteration pattern (i.e., one iterator is fixed as a pivot whilst the other iterator does a large skip).

In the case of Q3 and Q4, the AD skip join is outperformed by the STJ-D join by a small percentage of approximately 4%. The Q3 query evaluates `//inproceedings//author` on DBLP; in DBLP, both "inproceedings" and "author" are very frequent, and within every "inproceeding" element, there is always a minimum of one "author" element. Therefore, skipping through ancestor or descendant lists is useless for this query. As a result, the skipping algorithms reduce to the behavior of STJ-D. The extra in time taken is due to a number of small redundant skips. Q4 evaluates `//dblp//*`, which has an extremely small ancestor list of only one element, and an extremely large descendant list (the entire database). Again, in this scenario, skipping is not useful and the extra operations become an overhead that STJ-D does not have. However, as can be seen from the results, the overhead is only 4%, which is still quite acceptable given the gains on other queries. We also note that both the A-Join and D-Join are actually faster than STJ-D, mainly because the results are only single nodes, and hence there is reduced usage of the stack (depending on the number of input ancestor nodes). Similar results hold for Q7.

The queries Q9, Q10 and Q11 are performed on random data sets created by XMark. The data set is highly nested, with the practically rare and unnatural property that two distinct element names interleave each other multiple times on a single path (e.g. `//keyword//bold//keyword//bold`). Note the ratio of $\frac{|\mathcal{D}|}{|\mathcal{R}|}$ on Q9 is two, which means that, on average, mismatched descendant nodes and matched nodes interleave each other. This pattern makes skipping difficult and hence the algorithms yield similar performance to that of an STJ-D join. It is interesting to see that D-Join does perform slightly better on random and highly nested data, because the D-Join algorithm does not interact with the stack.

So far, all exponential skip-joins have similar performance on all queries with the exception of Q4. However, the A-Join outperforms the other two skip join techniques by a significant margin. This is because for Q4, all DList nodes are matched under the same ancestor node, therefore almost all descendant nodes are skipped using BD-SKIP.

We can also see from our experiments that stack manipulation does add notable overheads to the stack-based structural joins. For example, in Table 4, in queries where the STJ-D join outperforms both the AD-Join and A-Join, and a D-Join outperforms all other types of skip-joins. This is due to the fact that the stack is not maintained in D-Join, because only matching descendants are returned, whereas a stack has to be maintained in both AD-Join and A-Join.

Let $\mathcal{A}_r$ and $\mathcal{A}_s$ are the sets of nodes in $\mathcal{A}$ that will and will not be included in the result set, and similarly let $\mathcal{D}_r$ and $\mathcal{D}_s$ be the corresponding subsets of $\mathcal{D}$. If we use an AD-Join as an example, on the basis that the skipping strategy of all skipping functions are perfect, i.e., there are no unnecessary skips, then the minimum bound of the runtime cost is $\mathrm{O}(|\mathcal{A}_r| + |\mathcal{D}_r| + |\mathcal{R}|)$, which, if no skipping occurs, becomes $\mathrm{O}(|\mathcal{A}| + |\mathcal{D}| + |\mathcal{R}|)$ as in that case $\mathcal{A}_r = \mathcal{A}$ and $\mathcal{D}_r = \mathcal{D}$. In other words, the worst case performance of our proposed skip-joins is the same as that for STJ-D algorithm.

In Table 4, the last three columns show the performance time of AD-Join, A-Join, D-Join using the binary skipping strategy instead of the exponential skipping strategy. From the results, the performance for Q1, Q2, Q5, Q6, Q8 and Q12 matches exponential skipping. However, with the exception of Q1, all binary skip joins are slower than exponential skip joins. This is because of the $\log n$ nature of binary search; that is, even for AList or DList that have small gaps between matching nodes, it will still cost $\log n$ skips to search for the next node. However for Q1, the binary skipping strategy permits larger jumps through the AList, and hence has better performance than exponential skip.

4.3 Summary

To summarise our experimental results, our proposed skip join algorithm performed very well for Q1, Q2, Q5, Q6, Q8 and Q12, where the returning result node sets are small in size and there are large differences in size between AList and DList. Compared to the STJ-D algorithm, we were able to achieve up to three orders of magnitude in performance for the above queries. In general, the times for A-Join and D-Join are always faster than for the STJ-D algorithm, and in most cases faster than AD-Join. Therefore, we recommend the query optimizer should utilize A-Join and D-Join more for structural joins. However, the AD-Join still performns very closely to STJ-D Join for most queries where large result sets are returned. In the case of Q1, Q2, Q5, Q6, Q8 and Q12, an AD-Join was still able to outperform an STJ-D join by several orders of magnitudes. The experimental results also show that the exponential skipping strategy outperforms the binary skipping strategy, and that therefore we should adopt exponential skipping strategy as a default for future implementation.

5 Label Maintenance for Changing Data

All skipping strategies proposed in this paper rely on a fast method of determining ancestor-descendant relationships. We have not as yet discussed ways of providing these methods, whilst efficiently handling updates. Our previous work [9] examined the issue of efficiently handling document ordering in the presence of updates, and briefly showed how this work could be applied to the ancestor-descendant problem.

There are two strategies which have been mentioned in the literature previously that can handle updates efficiently. In the first strategy, we maintain both pre-order and post-order identifiers for each node in the database. As described in Dietz's work [7], the ancestor-descendant relationship between two nodes can then be determined by comparing these. For example, a node x is an ancestor of a node y if and only if $pre(x) < pre(y)$ and $post(x) > post(y)$. In the second strategy, each node is assigned a start and end identifier, such that node x is an ancestor of node y if and only if $start(x) < start(y)$ and $end(x) > end(y)$.

Updates in either strategy can be reduced to the order maintenance problem we studied in [9]. For the pre-order/post-order approach, we maintain two distinct lists of size n, where n is the number of nodes in the database. For the start/end approach, we maintain one distinct list of size $2n$. We showed in our previous work that the constant time solutions to the order maintenance problem can be outperformed by the $O(\log n)$ solutions in heavy read situations. Hence, it is likely that the logarithmic solution will be used in practice.

As the $O(\log n)$ solution is likely to be used, it appears that the start/end approach is at a disadvantage to the pre-order/post-order approach, due to the fact that it is maintaining a larger list. That is, the cost of maintaining the start/end list will be $2C \log 2n$ for some constant, but for the pre-order/post-order lists will be $2C \log n$. However, one possible disadvantage of the pre-order/post-order approach is that it is not clear that whether the updates in the pre-order list will overlap with the updates in the post-order list. To be more precise, it is not clear whether the set of pages loaded in for the pre-order relabelling will overlap with the set of pages loaded in for the post-order relabelling.

We now provide an informal justification as to why performing pre-order and post-order traversals at the same time will generally have less than twice the swapping penalty of performing one of the traversals. We will give our argument in terms of complete, uniform trees, which are of depth d, where each non-leaf node has exactly k children, and where each level is completely filled with nodes. Such trees, where k is much larger than d, model real-world XML documents quite closely.

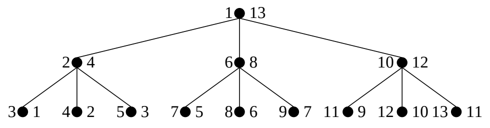

Figure 5: A complete tree ($k = 3, d = 3$). The numbers to the left of each vertex are the pre-order numbers; to the right are the post-order numbers.

Figure 5 gives an example of such a tree, including the two numeric identifiers attached to each node that are of interest to us, $pre(x)$ the pre-order identifier, and $post(x)$, the post-order identifier.

We now give some properties of such trees, useful in our justification.

Lemma 1 *$pre(x) - post(x)$ is an invariant for all x on the same level of the tree.*

Proof: Let x be any node on the same level (except the last node of the level), and y be its successor. In pre-order traversal, all the nodes of x's sub-tree lie between x and y. In post-order traversal, all the nodes of y's sub-tree lie between x and y. Because the tree is complete, these quantities are equal. Hence, $pre(y) = pre(x) + \alpha$ and $post(y) = post(x) + \alpha$, so $pre(y) - post(y) = pre(x) - post(x)$.

Lemma 2 *For the m^{th} node on the i^{th} level, we have $pre(x) = i + (m-1)\frac{k^{d-i+1}-1}{k-1}$.*

Proof: The first node in each level must be the i^{th} node in pre-order traversal. Moreover, we saw in the proof of Lemma 1 that the increment for $pre(x)$ and $post(x)$ between nodes on the same level is fixed. Hence, we have $pre(x) = i + (m-1)\alpha$, which x is the m^{th} node on level i. It is clear that α is equal to the number of nodes in a subtree of depth $d - \alpha + 1 = \frac{k^{d-i+1}-1}{k-1}$. Hence, $pre(x) = i + (m-1)\frac{k^{d-\alpha+1}-1}{k-1}$.

Table 5: Values of $E(|pre - post|)$

k	d	$E(\|pre - post\|)$
2	2	1.33
2	10	12.26
10	10	16.20
100	2	1.98
1000	5	7.99
100,000	2	2.00

Lemma 3 *For the m^{th} node on the i^{th} level, we have $post(x) = m\frac{k^{d-i+1}+1}{k-1}$.*

Proof: The first node on the level comes after all nodes in its subtree. Hence:

$$\begin{aligned} post(x) &= \frac{k^{d-i+1}-1}{k-1} + (m-1)\alpha \\ &= \frac{k^{d-i+1}-1}{k-1} + (m-1)(\frac{k^{d-i+1}-1}{k-1}) \\ &= m\frac{k^{d-i+1}-1}{k-1} \end{aligned}$$

Lemma 4 *For the i^{th} level, $pre - post = i - \frac{k^{d-i+1}-1}{k-1}$.*

Proof: Let x be any node on the i^{th} level, say the m^{th}. Then:

$$\begin{aligned} pre - post &= pre(x) - post(x) \text{ by Lemma 1} \\ &= i + (m-1)\frac{k^{d-i+1}-1}{k-1} - m\frac{k^{d-k+1}-1}{k-1} \\ &\quad \text{by Lemmas 2 and 3} \\ &= i - \frac{k^{d-i+1}-1}{k-1} \end{aligned}$$

We now consider the average absolute distance between the pre- and post-order identifiers, $E(|pre - post|) = \sum_{i=1}^{d} \frac{k^{i-1}(k-1)}{k^d-1}(|i - \frac{k^{d-i+1}-1}{k-1}|)$. We tabulate the values of this quantity for various values of k and d in Figure 5.

Of greatest interest in Table 5 are those values for which k is large and d is very small, because these correspond to the topology of typical real-world XML documents such as DBLP. As can be seen in these cases the difference is very small.

We are now in a position to give an informal justification for the original claim, that simultaneously updating the pre-order and post-order lists will not be twice as expensive as updating one of them. Let us consider the situation where we are inserting into a node into an XML document, and this insert requires relabeling n_{pre} surrounding nodes in the pre-order list, and n_{post} surrounding nodes in the post-order list. While n_{pre} and n_{post} will be very different, it is a property of many ordering algorithms, such as those of Bender *et al* [2], that the expected number of relabellings is bounded in the amortized worst case, with the bound depending on the variant employed. Hence, we can expect that *on average* $n_{pre} \approx n_{post}$, and we assume below that they are equal (and hence refer to the quantity simply as n).

We can then restate our claim as follows: if we access a contiguous sub-list of n nodes in the pre-order traversal, how many of these would also lie in the contiguous sub-list of n nodes in the post-order traversal, starting from the same node? If the proportion is high, then we would expect that doing the traversals together would involve strictly less than twice the number of disk accesses that would be required if the proportion is low. We now argue that the proportion is, on average, quite high.

We have shown above that for complete trees with fanout k and degree d, the quantity $E(|pre(x) - post(x)|)$ is generally small in the cases we are interested in. In fact, although it is omitted from this paper, a similar result also holds for complete trees with where the fanout is fixed on each level, e.g., the first level has fanout k_1, the second level has fanout k_2. These kinds of trees very closely model real world XML repositories such as DBLP and MEDLINE. Hence, if a node lies in position $pre(x)$ in the pre-order traversal of the database, we expect the node will have a position somewhere near $pre(x)$ in the post-order traversal. Thus, we conclude that it is likely that a large proportion of the nodes accessed during a pre-order traversal will also be accessed during a post-order traversal. It is also intuitively obvious that as the number of nodes n being relabeled increases, the proportion also increases.

It is important to emphasize at this point that when we refer to $pre(x)$ and $post(x)$, we are *not* talking about the numeric tag assigned by the document ordering algorithm. While such tags are indeed ordered consistently, gaps of arbitrary size may lie between adjacent tags, and hence the results above do not apply. Instead, $pre(x)$ refers to the *absolute* position of the node in pre-order traversal, when taking a snapshot of the (possibly dynamic) database in time, and likewise for $post(x)$. As long as this static snapshot can be reasonably approximated by a complete tree of small depth, which we assert happens frequently in practice, then our result should hold quite closely. Thus, our informal justification applies to *any* ordering scheme which uses a range relabeling strategy.

To summarize, we have shown that the disk accesses required for updating the pre-order and post-order lists are shared between the updates for both lists, and this reduces the cost of the total update cost for this ancestor-descendant scheme. This reduction, coupled with the extra update expense required for the start/end scheme when using an $O(\log n)$ update algorithm, strongly suggests that a pre-order/post-order scheme is the better strategy for use in large XML databases.

6 Conclusions

This paper has focused on improving the algorithms for structural join, a core operation for XML query processing. We presented a simple, yet efficient, improvement to the work of Al-Khalifa *et al* [1], which skips unnecessary nodes in the ancestor and descendant lists. In constrast to from [5], our method does not require any auxiliary index structure and hence is significantly easier and cheaper to maintain. It can also be implemented in non-database applications such as an XSL processor, which does not normally have a built-in B-Tree index, as well as into a streaming XML data processor.

Furthermore, informal justifications of the effect of updates on the structural join problem have been presented. Since the ordering scheme we used in this paper is based on the use of preorder and postorder identifiers, the update cost is identical to those analyses and experiments performed in [2] and our other work [9]. By employing the use of gaps in a theoretically sound fashion, the amortized update cost is much lower than the update cost in other tree-based labeling schemes such as [11].

Finally, extensive experiments on both real-world data and synthetic data have shown that our extension has improved the performance of the state-of-the-art structural join algorithm [1] by orders of magnitude at the best.

We believe that there is still a wide range of interesting research problems in this area. In particular, we are currently investigating the extension of our work to produce a query optimization framework in the presence of ordering. Similar work in this area includes [4, 12].

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*. IEEE Computer Society, 2002.

[2] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, September 17–21 2002.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (second edition). `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD Conference*, pages 310–321. ACM, 2002.

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

[6] DBLP. `http://www.informatik.uni-trier.de/~ley/db/`.

[7] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, 1982.

[8] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *ACM STOC*, pages 365–372. ACM, 1987.

[9] D. K. Fisher, F. Lam, W. M. Shui, and R. K. Wong. Fast ordering for changing XML data. Technical Report UNSW-CSE-0317, School of CSE, University of New South Wales, Sydney, Australia, 2003.

[10] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(1):27–34, March 1999.

[11] J. Haifeng, H. Lu, W. Wei, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Join. In *ICDE*. IEEE Computer Society, 2003.

[12] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of VLDB*, to appear, 2003.

[13] J. McHugh and J. Widom. Query optimization for XML. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB*, pages 315–326. Morgan Kaufmann, 1999.

[14] MEDLINE. `http://www.nlm.nih.gov/bsd/licensee/data_elements_doc.html`.

[15] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[16] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Assessing XML data management with XMark. In S. Bressan, A. B. Chaudhri, M.-L. Lee, J. X. Yu, and Z. Lacroix, editors, *EEXTT*, volume 2590 of *Lecture Notes in Computer Science*, pages 144–145. Springer, 2003.

[17] W3C Recommendation. XML Path Language (XPath) Version 1.0. *`http://www.w3.org/TR/xpath`*, Nov. 1999.

[18] W3C Working Draft. XQuery 1.0: An XML query language. *`http://www.w3.org/TR/2002/WD-xquery-20021115`*, Nov. 2002.

[19] 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 Conference*, 2001.