Querying and Maintaining Ordered XML Data using Relational Databases

William M. Shui **Franky Lam** **Damien K. Fisher** **Raymond K. Wong**

School of Computer Science and Engineering
and National ICT Australia
University of New South Wales
NSW 2052, Australia
{wshui, flam, damienf, wong}@cse.unsw.edu.au

Abstract

Although data stored in XML is of increasing importance, most existing data repositories are still managed by relational database systems. In light of this, recent XML database research has focused on extending relational database systems to handle XML data efficiently. While there are many issues in processing XML data efficiently, containment queries are the queries that often appear and need to be optimized. Recently, structural joins have been proposed to process containment queries efficiently. To date, structural join algorithms are mostly based on stacks and/or external B-Tree indices. Most of these prototypes have been implemented on object databases. This paper proposes an efficient structural join algorithm that can be implemented on top of existing relational databases. Experiments show that our method performs far more superior than previous work in both queries and updates.

1 Introduction

XML is being used increasingly often for the exchange of data from heterogeneous data sources, due to its self-describing nature. At present, most business data is stored in relational databases, mainly due to the decades of research effort that have been put into the field, the stability and extensive feature set of current commercial relational databases, as well as the wide range of existing business solutions based upon relational databases. Consequently, there has been significant research into finding ways to harness the versatility of the XML data model with minimal modifications to the mechanisms of relational databases. There are roughly four areas of interest in the domain of XML data management using relational databases:

1. Efficient publishing of relational data as XML views (Carey, Florescu, Ives, Lu, Shanmugasundaram, Shekita & Subramanian 2000, Shanmugasundaram, Kiernan, Shekita, Fan & Funderburk 2001), to allow legacy data to be accessed in a consistent fashion;
2. The translation of XML queries written in languages such as XQuery[1] into equivalent SQL queries (Krishnamurthy, Kaushik, Naughton & Chakaravarthy 2004, Manolescu, Florescu & Kossmann 2001, DeHaan, Toman, Consens & zsu 2003), to provide a uniform interface to disparate data sources;
3. Storing native XML data in a relational repository without losing any information, such as relative order (Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita & Zhang 2002, Yoshikawa, Amagasa, Shimura & Uemura 2001); and
4. Extending relational technology to allow efficient querying of stored XML documents (e.g., through the use of containment queries) (Zhang, Naughton, DeWitt, Luo & Lohman 2001).

This paper is mainly focused on the last two points, which investigate the issues surrounding the storage, maintenance and retrieval of native XML documents (as opposed to the first two, which concentrate on exposing existing data as XML). There are several problems with current relational XML querying and storage techniques:

1. Storing XML documents by shredding the data into rows and storing them in *edge tables* has been widely adopted for storing XML data in relational databases (Shanmugasundaram, Tufte, Zhang, He, DeWitt & Naughton 1999, Deutsch, Fernandez & Suciu 1999, Shimura, Yoshikawa & Uemura 1999, Shanmugasundaram, Shekita, Kiernan, Krishnamurthy, Viglas, Naughton & Tatarinov 2001). This approach can be easily extended to maintain information related to the *document order*[2] of each element (Yoshikawa et al. 2001). However, frequent insertions and deletions of nodes may require significant relabeling of elements in the edge table, which can have an adverse effect on database performance.
2. SQL can be used to perform containment queries of ordered XML data in relational databases (Yoshikawa et al. 2001, Tatarinov et al. 2002, Krishnamurthy et al. 2004, Manolescu et al. 2001, DeHaan et al. 2003, Carey et al. 2000, Shanmugasundaram, Kiernan, Shekita, Fan & Funderburk 2001). However, for branching path queries such as *//a[.//b]//c*, the SQL engine cannot take advantage of the intermediate result sets returned from the sub-queries without either creating a separate view or table, or using a (possibly recursive) sub-query. This approach can be expensive if the XPath expression contains many containment queries, and does not scale well. Also, having a large SQL query with nested sub-queries can sometime confuse the query optimizer and lead to bad query plans, as shown in (Tatarinov et al. 2002).
3. A large amount of effort has been put into efficient handling of XML data in native XML systems, especially in speeding up the efficiency of structural queries (Jagadish, Al-Khalifa, Chapman, Lakshmanan, Nierman, Paparizos, Patel, Srivastava, Wiwatwattana, Wu & Yu 2002, Al-Khalifa, Jagadish, Koudas & Patel 2002, Haifeng, Lu, Wei & Ooi 2003, Chien, Vagena, Zhang, Tsotras & Zaniolo 2002, Halverson, Burger, Galanis, Kini, Krishnamurthy,

[1] See http://www.w3.org/

[2] Document order of an XML document is the preorder traversal of its document tree as shown in Figure 1.

Rao, Tian, Viglas, Wang, Naughton & DeWitt 2003). This work is not directly applicable to relational databases, as it requires special purpose external index systems and storage schemes to perform fast structural joins. Therefore, the integration of external modules into commercial relational databases could be complex and inefficient.

In this paper, we provide a novel solution for the above problems and the main contributions of this paper are:

1. We introduce a relabeling scheme for efficient and scalable maintenance the document order of XML data in relational database systems. This approach minimizes the total number of relabelings performed during frequent insertions or deletions. We show experimentally, under a large number of insertions into the database, our relabeling scheme achieves amortized $O(\log n)$ for the number of node relabelings.

2. We introduce a new kind of structural join algorithm for answering XML containment queries on XML data stored in relational database systems. We present extensive experimental results on the performance of our proposed structural join algorithm and show that our approach works well for queries of widely varying degrees of selectivity; in particular our approach outperforms other approaches by a significant margin when the selectivity of the query is not high.

3. We also show our relabeling scheme and our structural join algorithm can complement each other as a whole system. Both of them work well on the well studied *Edge Table* XML storage approach in relational databases and they also support cursor interface to the database backend. Therefore, our proposed update maintenance and querying algorithms are capable of working with any off the shelf relational database systems without any internal modification.

The rest of this paper is organized as follows. Some background knowledge and related work on XML are described in Section 2. In Section 3, we present a relabeling scheme for maintaining the document order of XML data in relational databases. In Section 4, we introduce our structural join algorithm for answering XML containment queries on relational databases. Performance analaysis of our algorithms are presented and discussed in Section 5. Finally, Section 6 concludes this paper.

2 Background and Related Work

2.1 XML Data Model

In this paper, we will model XML documents as ordered, labeled, finite, unranked trees. For example, Figure 1 is a tree representation of a small fragment of the DBLP data set [3]. The document order corresponds to the preorder traversal of this tree and the relative order of nodes is indicated by the order of the siblings. For instance, the article comes before the book. The vectors of numbers under each node denote the values assigned using the *region encoding* labeling scheme.

The region encoding generally assigns a tuple of four numbers $\langle doc, start, end, level \rangle$ to each node where doc is the document identifier to which the node belongs [4]; the $start$ and end specify ranges which are assigned to each node in such a way that for any two nodes x and y in the same document. Using the $start$ and end identifiers, x is an ancestor of y if and only if $x.start < y.start$ and $y.end < x.end$. It is an invariant of the labeling scheme that for any node x, $x.start \leq x.end$. The ranges can easily be assigned by traversing through the document in a preorder traversal, with a stack of nodes which grows to at most the depth of the document. It should also be noted that the order of the identifier $start$ directly yields the document order of the XML document. The identifier $level$ gives the level of each node in the document. Coupled with $start$ and end, this allows one to determine parent-child relationships in addition to ancestor-descendant relationships. The correctness and completeness of the above labeling scheme is well known (Zhang et al. 2001, Haifeng et al. 2003, Chien et al. 2002, Halverson et al. 2003). Its advantages over other schemes will be discussed in the next section.

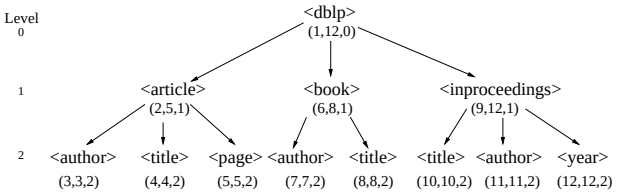


Figure 1: A sample XML document annotated using the region encoding labeling scheme

2.2 XML Labeling Schemes

Most XML query languages such as XPath, XQuery and XSLT mandate that the results of a query are returned in document order. Thus, in order to evaluate queries in such languages, there are two options: either maintain results throughout the evaluation in document order, or introduce a sort operator which can be called at appropriate times. Maintaining results in document order throughout query evaluation greatly restricts the choice of query plans, and is impossible if the query requires the data to be resorted along a different axis at some point. Hence, some sorting mechanism is a necessity in any XML query engine — such a sorting operator is also crucial to structural join algorithms, which generally take their input sorted into document order.

The standard approach to providing such a sorting operator is to assign each node a label denoting its relative order; several such schemes have been proposed previously. In fact, most schemes focus on a closely related problem, the *ancestor-descendant* problem, in which the task is to determine whether one node is the ancestor of another. The most popular ancestor-descendant labeling scheme is the *region encoding* described in the previous section. This method has several advantages, including fixed size labels which fit in a machine register, and extremely fast comparisons (reducing to only a few machine operations). A very similar approach was suggested in (Dietz 1982), where preorder and postorder identifiers are assigned to each node.

Recently, using prime numbers to encode the order of XML nodes has been investigated in (Wu, Lee & Hsu 2004). However, their encoding scheme requires indirection through large array in order to answer containment queries, which could become a bottleneck for large data sets. There are also few variable size labeling schemes for XML data that have been studied in (ElSayed, Dimitrova & Rundensteiner 2003, Tatarinov et al. 2002). Generally, these schemes are variants of the Dewey encoding scheme, using strings or integers as values. The advantage of these schemes is that they do not relabel any nodes on updates, but this comes at a cost: in (Cohen, Kaplan & Milo 2002), it gives a lower bound on the length of the labels in such schemes which is linear in the size of the database. Thus, it is difficult to allocate a fixed portion of each record in the database to the labels, complicating the storage layer.

[3] See http://www.informatik.uni-trier.de/~ley/db/

[4] We will assume throughout this paper that we are dealing with only a single document, and hence we omit the doc component of the label; our techniques trivially extend to the case of multiple documents.

2.3 Storing XML in RDBMS

Research projects such as SilkRoute (Fernández, Kadiyska, Suciu, Morishima & Tan 2002) and the XPERANTO project (Carey et al. 2000) have proposed methods for efficiently publishing relational data as XML (Chaudhuri, Kaushik & Naughton 2003). However, due to the unordered nature of the underlying relational data, support for storing and querying ordered XML data has been ignored by most of these implementations. Some tools has been developed for storing XML documents into relations (Bohannon, Freire, Roy & Simeon 2002, Deutsch et al. 1999, Florescu & Kossmann 1999, Shanmugasundaram et al. 1999, Shanmugasundaram, Shekita, Kiernan, Krishnamurthy, Viglas, Naughton & Tatarinov 2001, Shimura et al. 1999), but none of these provides comprehensive treatment of document order.

More recently, the problem of storing XML and maintaining document order in relational databases has been further studied. In (Tatarinov et al. 2002), it proposed three encoding schemes (*global*, *Dewey* and *local*) for handling document order and combined these labeling schemes with the edge shredding approach (Florescu & Kossmann 1999) to form a single relation for storing XML data. If the XML document comes with a schema, then a path table (Shimura et al. 1999, Yoshikawa et al. 2001) is also used, which acts just like DataGuides (Goldman & Widom 1998). With the global encoding scheme, the edge relation can be defined as follows, where *id* denotes the global position of a node with the document *Edge(id, parent_id, end, path_id, value)*. Experiments performed in (Tatarinov et al. 2002) showed that this global scheme had the best query performance. The local encoding scheme has an edge relation defined as *Edge(id, parent_id, sIndex, path_id, value)*, where *sIndex* gives the position of a node amongst its siblings. The experiments also showed that the local labeling scheme has the worst query performance and the best update performance. However, even with this scheme, if the fanout of the XML document is large, then the worst case performance is extremely poor.

In both (Tatarinov et al. 2002) and (O'Neil, O'Neil, Shankar Pal, Schaller & Westbury 2004), the use of Dewey style of encoding scheme[5] have been proposed, based on the argument that it will reduce the number of relabeling of XML nodes in the database. In (Tatarinov et al. 2002), it showed the performance of Dewey based encoding scheme (both querying and update) are in between the global and local schemes. However, Dewey based encoding scheme do not allow fixed size keys, this can have a greater adverse effect especially in sorting data nodes, where the sorting algorithm will not have a constant time node comparison operator (the size of keys may vary between nodes).

There has been some recent effort on translating XQuery to SQL for querying XML data in relational databases (Krishnamurthy et al. 2004, Manolescu et al. 2001). However, their performance is still bounded by the performance of SQL in answering ancestor-descendant queries, and hence further work on this problem is of great relevance.

2.4 Structural Joins

XML containment queries have been studied in the context of both relational and native XML databases. In particular, much effort has been invested in developing a structural join operator, which joins together two sets to produce ancestor-descendant pairs. The Multi-Predicate Merge Join (MPMGJN) (Zhang et al. 2001) was initially proposed for solving structural join problems in relational databases. In summary, it performs a merge-join, possibly performing multiple scans through the inner join operand (either the input ancestor list or descendant list) as necessary. The primary disadvantage of this method is the multiple scans necessary through the data.

The current state of the art structural join is described in (Al-Khalifa et al. 2002). On the assumption that the inputs sets are sorted in document order, their algorithm works in the optimal time of $O(|AList| + |DList| + |Output|)$, where $AList$ and $DList$ are the input sets, and $Output$ is the output set. The algorithm[6] they propose is stack based, in that it maintains a stack of nested ancestor nodes whilst iterating through the two sets.

Although the stack-based structural join has good space and time complexity, it cannot take advantage of selectivity information to speed up the structural join. This is because the STJ algorithm requires the traversal of all nodes in both the ancestor and descendant lists. More recent work have shown that using B^+-Tree indices built upon the start identifier of elements can allow efficient skipping of descendant nodes, and with this technique significant gains over the stack-based join algorithm were made when the selectivity of the query was high. In (Haifeng et al. 2003), the XR-Tree, a variant of the B^+-Tree, was introduced to allow fast ancestor node skipping. The XR-Tree achieves this by adding "stab lists" to internal nodes of the tree, which allow it to find all the ancestor nodes of a given descendant node by simply performing a key lookup of the descendant node. They also showed their descendant skipping is as efficient as the B^+-algorithm, but they outperformed B^+-algorithm in ancestor joins significantly.

The Zig-Zag structural join algorithm described in (Halverson et al. 2003) is an extension of the MPMGJN, which uses two indices built upon the start and end identifiers of each node. The two indices are used together to perform the structural join in a "zig-zag" fashion, by first skipping descendant nodes until a descendant of the current ancestor node is found, and then skipping on ancestor nodes until a match is found, et cetera. This skipping is achieved by alternating between the primary (*start*) and secondary (*end*) indices during the join process.

The staircase join algorithm described in (Grust, van Keulen & Teubner 2003) uses a preorder/postorder labeling scheme to label nodes within an XML document, which allows it to efficiently determine the ancestor-descendant relationship between two nodes, as noticed in (Dietz 1982). They also modified the SQL query optimizer to allow fast processing of containment queries. However, they only showed the performance of the algorithm in a main memory based database system, which did not consider all the overhead that would have been incurred in a disk-bound database system.

A more general class of structural join techniques are the *twig joins*, which try to answer more complicated queries in a single pass, e.g.:

```
//inproceedings[.//author and .//title]
```

To minimize the number of scans, various twig-join algorithms have been proposed to facilitate these type of queries (Jiang, Wang, Lu & Yu 2003, Bruno, Koudas & Srivastava 2002, Choi, Mahoui & Wood n.d.). Although our structural join algorithm can be easily extended to perform twig joins, we will omit the algorithm and performance analysis of this due to space limitations.

3 Maintaining Document Order

It is a requirement in both XPath and XQuery specifications that all result nodes are to be returned in sorted document order. Therefore, having a fast order comparison operator for XML nodes is crucial for an efficient XML query processor. If we use an edge table to store the XML data in a relational database system, then sorting

[5] See http://www.oclc.org/oclc/fp/about/about_the_ddc.htm

[6] The stack-tree-based family of structural join is interchangeable with STJ algorithm hereafter.

Algorithm 1 XRU-Insert

```
newNode is inserted between prevNode and nextNode.
XRU-Insert(prev, next, newNode)
1   if(the database is empty) then
2      newNode.start ←(MAX / 2); return ;
3   if(prev = null ) then
4      if(nextNode.start ≠ 0) then
5         newNode.start ←next.start/2; return ;
6   else if(next = null ) then
7      if(prev.start ≠ MAX) then
8         if(prev.start ≠ 0) then
9            newNode.start ←((MAX - prev.start + 1)/2);
10        else
11           newNode.start ←(MAX/2); return ;
12  else
13     if(prev.start + 1 ≠ next.start) then
14        if(prev.start ≠ 0) then
15           newNode.start ←prev.start +
16           (next.start - prev.start + 1)/2;
17        else
18           newNode.start ←next.start/2; return ;
19  Relabel(prev, next, newNode);
```

Algorithm 2 XRU-Relabel

```
dbSize:  total number of nodes in the database + 1
XRU-Relabel(prev, next, newNode)
1   T ←e^(log(dbSize/(MAX+1)) / (log(max+1)/log(2)));
2   prop ←2.0/T;
3   if(prev ≠ null ) then
4      val ←prev.start;
5   else
6      val ←0;
7   j ←2;
8   low ←val &~(j - 1); high ←(low + j);
9   nCursor ←SQL cursor for
           start ≥ low in ascending order;
10  pCursor ←SQL cursor for
           start < low in descending order;
11  for(power ←1;
       j > 0 and ((pCursor ≠ null ) or
            (nCursor ≠ null ));
       j ←j × 2, power ←power + 1)
    do
12     Move nCursor forward until nCursor = null or
       or nCursor.start > high;
13     Move pCursor forward until pCursor = null or
       or pCursor.start < low;
14     count ←the nodes between pCursor and nCursor;
15     if(count < prop^power) then
16        break;
17     low ←val &~(j - 1); high ←(low + j);
18  end for
19  if(j > MAX) then
20     count ←dbSize;
21  if(high > MAX) then
22     high ←MAX;
23  g ←(MAX - low)/count;
24  l ←low;
25  for each (node between low and high
       in ascending order) do
26     node.start = l; l ←l + g;
```

nodes into document order based on the start attribute is trivial.

However, an important question still exists: when we insert new nodes into the database, how do we relabel the start attributes of surrounding nodes so that we have room for the new node? If there are frequent updates, then we need to relabel the affected tuples in the database in order to maintain the order of existing elements. This could be prohibitively expensive, depending on the number of relabelings required.

In this section, we attack the problem of updating the edge table representation in such a way that we can continue to use the structural join algorithms presented in the previous section. We do this by providing an effective way of maintaining the document order of nodes in a relational database, which minimizes the adverse impact of this maintenance upon the insertion performance of the database.

3.1 Relabeling the Edge Table

The XRU relabeling scheme we will use is the extension of the algorithm described in (Bender, Cole, Demaine, Farach-Colton & Zito 2002). While the algorithm has good theoretical properties, it is unclear what its performance will be like in a relational database system. We demonstrate here that its performance is substantially superior to previous techniques given in the literature.

Let $u \in \mathbb{N}$ be the maximum identifier allowed for representing the *start* field of XML elements, and consider the complete binary tree $\mathcal{B}$ corresponding to the binary representations of all numbers between 0 and $u-1$. Thus, the depth of the tree is $\log|u|$, and the root-to-leaf paths are in one-to-one correspondence with the interval $\mathcal{I} = [0, u-1] \subseteq \mathbb{Z}$; more generally, any node of the tree corresponds to a sub-interval of $\mathcal{I}$. When the database has n nodes, this tree will have n leaf nodes used, corresponding to the identifiers used as *start* values for the nodes in the database. For a node $n \in \mathcal{D}$, we write *n.start* $\in \mathcal{B}$ for its numeric identifier. For a node in the identifier tree, we define its density to be the proportion of its descendants (including itself) which are allocated as identifiers.

Based on the above definition, we can split the relabeling algorithm into two parts: *XRU-Insert* and *XRU-Relabel*. XRU-Insert assumes that there is always a gap ≥ 1 between the identifiers of the two nodes adjacent to the new node. If this is the case, then when inserting a node *x* between node *y* and *z*, we can set *x.start* = $\lfloor \frac{1}{2}(y.start + z.start) \rfloor$. When the database is empty, we simply insert the new node and set the tag to $u/2$. Lines 3 to 11 in Algorithm 1 handle the situation where the new node is appended after the last node or inserted before the first node of the database. However, in the case where no gap is available for inserting the new node, *XRU-Relabel* is called to re-assign gaps between nodes.

The aim of the *XRU-Relabel* is to search through the ancestors of x, starting with its parent node and up. We stop when the ancestor node a has a density less than T^{-i}, where i is the distance of a from x in the virtual binary tree $\mathcal{B}$ and T is a density threshold that users can define. We then relabel all the nodes which have identifiers in the sub-range corresponding to a. The complete pseudo-code for conflict relabeling scheme is presented in Algorithm 2. (Bender et al. 2002) also suggested when T is the smallest value before root over-flows, it should yield good practical running time.

In order for this relabeling scheme to work, we need a fast method of finding the right ancestor such that its density is lower than the threshold. In practice, as the number of elements increases in the XML document, the potential number of nodes we have to scan to find the right node with the right density can be at worst case n, where n is the size of the database. This can pose a problem especially for the relational database. Generally speaking, if we use the *global* edge table described in (Tatarinov et al. 2002), finding the right node with the right density means continuously performing the following SQL statement.

```
SELECT count(id) from global where
   id between low and high;
```

After each execution of the statement, we have to check if *count* is less than the threshold. If so, we have found the right ancestor node with a suitable density. However, we have ignored the fact that each execution of this SQL statement requires an index scan of the *id* attribute in the relational database. This is the main motivation behind lines 9 to 26 in Algorithm 2, where the relational cursor interface is used for finding ancestor node with the desired density instead of a simple SQL statement. We first declare two cursors starting from *prev* and *next* respectively, *pCursor* acts as a pointer to the previous node in document order to the new XML node. The cursor is ordered in descending order, so that *fetch next* moves the cursor closer to the

node that has the smallest *start* value, i.e., the XML document root node. Similarly, *nCursor* points to the next node in document order to the new XML node and it is in ascending order. The action *fetch next* on *nCursor* moves the cursor closer to the last descendant node of the document. Since we only declare the cursor once and we only move the cursors in one direction, we can guarantee at most 2 passes are required for the relabeling process. Line 1 and 2 of Algorithm 2 build the density threshold at the time of node insertion, since the density threshold changes for each insertion. Line 8 and line 17 determine the current tag range at each iteration, since both *pCursor* and *nCursor* expand every iteration.

4 SS-Join for Relational Databases

Algorithm 3 SS Descendant Join

```
aSize:  size of the ancestor nodes
dSize:  size of the descendant nodes
aPos:  current position of the aNode
dPos:  current position of the dNode
SS-D-Join(aNode, dNode)
 1 dPos ←0, aPos ←0, cNode ←null
 2 if(aNode = null or dNode = null) then
 3   return
 4 while( (dPos ≤ dSize) and (aPos ≤ aSize)) do
 5   if(cNode neq null and
        aNode.start > cNode.start and
        dNode.start > aNode.end) then
 6       cNode ←null
 7   else if(aNode ≠ null
       and aNode.start < dNode.start) then
 8     if(cNode = null) then
 9       cNode ←aNode
10       if(aPos ≥ aSize) then
11         aNode ←null
12       aNode ←skipAncestors(aNode, dNode)
13   else
14     if(cNode ≠ null) then
15       append(dNode, output)
16       if((dNode ←fetchNext(cursorD)) = null) then
17         break
18     else if(dPos < dSize and aNode ≠ null) then
19       dNode ←skipDescendants(aNode, dNode)
20     else
21       if(aNode.start > dNode.start) then
22         break
23       else
24         if(aNode.contains(dNode)) then
25           append(dNode, output)
26           dNode ←fetchNext(cursorD)
27           if(dNode = null) then
28             break
29 end while
```

Most existing structural join algorithms return matching (*ancestor, descendant*) pairs. In many cases, only one component of the pair is used in the evaluation of the query. For instance, the query *//a//b* only makes use of the descendant nodes from the join. While the full structural join, which returns both components, is still useful in some cases, it is clearly worthwhile investigating alternative, faster strategies which only return one of the components. Motivated by this goal, we propose in this section an alternative structural join algorithm (SS-Join), which is partially inspired by the Zig-Zag algorithm described in (Halverson et al. 2003). As with all existing structural join algorithms, the SS-Join assumes that input ancestor and descendant node lists are sorted in ascending document order (i.e., ordered by the *start* attribute of each node). There are two variants of the SS-Join algorithm, the *SS-A* join algorithm for ancestor queries, and the *SS-D* for descendant queries.

Algorithm 4 SS Ancestor Join

```
aSize:  size of the ancestor nodes
dSize:  size of the descendant nodes
aPos:  current position of the aNode
dPos:  current position of the dNode
SS-A-Join(aNode, dNode)
 1 if(aNode = null or dNode = null) then
 2   return
 3 while((dPos ≤ dSize) and (aPos ≤ aSize)) do
 4   if(aNode.start ≤ dNode.start) then
 5     if(aNode.end ≥ dNode.start) then
 6       append(aNode, output)
 7     aNode ←skipAncestors(aNode, dNode)
 9     if(aPos ≥ aSize) then
10       break
11   else
12     if(dPos ≤ dSize) then
13       dNode ←skipDescendants(aNode, dNode)
14     else
15       break
16 end while
```

4.1 The SS-D Join

The complete SS-D algorithm is given in Algorithm 3. Apart from some trivial checks, the core of the algorithm is between lines 4 and 29. The basic idea is to maintain an ancestor node *cNode*, and progressively check each descendant node against this ancestor node, looking for matches. The algorithm can be divided into 3 main sections. We first determine if the current ancestor node (*cNode*) should be removed by checking if it still contains either the next *aNode* or the current descendant node (*dNode*). In that event, then *aNode* becomes *cNode*.

If *aNode.start* $<$ *dNode.start*, then we know that *aNode* is either an ancestor of *dNode*, or precedes it in the document. If *cNode* is null, then we assign *aNode* to *cNode*. We keep *cNode* as the last qualifying ancestor node that contains the current *dNode*. This is guaranteed by step 1. The function skipAncestors then skips past all the ancestor nodes that are in the *preceding* axis of the current descendant node. As a result of this, function skipAncestors returns either null or an ancestor node of *dNode*.

Finally, if *dNode* is either a descendant node of *cNode*, then *dNode* is appended to output. However, if it is in the *preceding* axis of the current *aNode*, then the function skipDescendants returns the next descendant node that is either a descendant of *aNode* or is in *following* axis of *aNode*.

Algorithm 5 Skip Descendants

```
skipDescendants(aNode, dNode)
 1 if(dNode.start ≥ aNode.start) then
 2   if(dPos ≥ dSize) then
 3     dPos ←dPos + 1;
 4   return dNode;
 5 if((dNode ←fetchNext(cursorD)) = null ) then
 6   return null
 7 else if(dNode.start ≥ aNode.start) then
 8   return dNode;
 9 r ←⌈log(dSize − dPos)/log(2)⌉;
10 for(i ←0, g ←1;
     i < r and dPos < dSize; i ←i + 1) do
11   g ←g × 2;
12   if(g + dPos > dSize) then
13     g ←dSize − dPos;
14   moveCursor(cursorD, FORWOARD, g − 1);
15   dNode ←fetchNext(cursorD);
16   if(dNode.start ≥ aNode.start) then
17     break;
18   else
19     dPos ←dPos + (g − 1);
20 end for
21 return binarySearchDescendant(dPos, dPos + g, aNode);

binarySearchDescendant(min, max, aNode)
   binary search between min and max by moving
   the cursorD back and forth until
   cursorD[i].start >= aNode.start and the
   cursorD[i-1].start < aNode.start.
   Otherwise, return null.
```

Algorithm 6 Skip Ancestors

```
skipAncestors(aNode, dNode)
 1 if((aNode ←fetchNext(cursorA)) = null ) then
 2   return null ;
 3 if(aNode.end ≥ dNode.start) then
 4   return aNode;
 5 if(aPos ≥ aSize) then
 6   return null ;
 7 r ←⌈log(aSize − aPos)/ log(2)⌉;
 8 for(i ←0, g ←1;
     i < r and aPos < aSize; i ←i + 1) do
 9   g ←g × 2;
10   if(g + aPos > aSize) then
11     g ←aSize − aPos;
12   moveCursor(cursorA, FORWOARD, g − 1);
13   aNode ←fetchNext(cursorA);
14   if(aNode.end ≥ dNode.start) then
15     break;
16   else
17     aPos ←aPos + (g − 1);
18 end for
19 return binarySearchDescendant(aPos, aPos + g, dNode);
binarySearchAncestor(min,max,dNode)
   binary search between min and max by moving
   the cursorA back and forth until
   cursorA[i].end >= dNode.start and the
   cursorA[i-1].end < dNode.start.
   Otherwise, return null.
```

4.2 The SS-A Join

SS-A join is described in Algorithm 4. It first checks if the current ancestor node *aNode* is either in the *preceding* or *ancestor* axes of the current descendant node *dNode*. If so, we then check if the current ancestor node contains the current descendant node (using the property of region encoding scheme). It outputs *aNode* if *dNode* is a descendant of it. Otherwise, the function skipAncestors skips forward on the ancestor node list until it finds an ancestor node that is either in the *following* or *ancestor* axes of *dNode*. The ordered nature of the ancestor and descendant node lists guarantee that if the function skipAncestors cannot find an ancestor node that either *contains* or *follows dNode*, then all nodes in the ancestor node list are in the *preceding* axis of *dNode*. Lines 12 to 15 of Algorithm 4 describe the second part of the algorithm. It skips forward through descendant nodes when *aNode* is either in the *descendant* or *following* axes of *dNode*. The function skipDescendants returns either null or a descendant node that is either in the *following* or *descendant* axes of *aNode*.

In summary, both SS-A and SS-D algorithms alternate between ancestor and descendant lists to find matching nodes by means of skipping. The details of the skipping mechanism are described next.

4.3 Index-Free Skipping

Apart from avoiding any stack during the structural join process, SS-Join also differs from existing structural join algorithms (with the exception of the stack-tree join (Al-Khalifa et al. 2002)) by not relying on index lookups for the skipping mechanism. As a result, it does not involve the overhead of maintaining (mostly external) indices. Algorithms 5 and 6 describe the skipAncestors and skipDescendants functions. It has been suggested that knowing the distribution of matching nodes for structural join can help the query optimizer to choose between using skipping or a simple sequential scan using nested loops (Halverson et al. 2003). However, making such decision is difficult if the node distribution information is not available (or if it is expensive to obtain).

The motivation behind the functions skipAncestors and skipDescendants is to allow fast join processing without any knowledge of the node distribution. Both skipping functions use the relational database cursor to skip. The basic idea is to move the cursor forward by 1, 2, 4, 8 tuples, and so on, with an exponentially increasing gap. If the matching nodes are close to each other, then we can obtain the next matching node within a few skips. In fact, the first skip is just a sequential scan by fetching the next tuple. The exponentially increasing gap for each skip allows the cursor to skip through a large amount of unmatched nodes in only a few iterations. In fact, the maximum number of exponential jumps is bounded by the number of bits used for storing the *start* field.

So, what if we over-skip the target node? We know that once we have over-skipped the target node, the search space for the target node is $2^k - 2^{k-1}$. Therefore, we move the cursor back and forth between (*min, max*), where *max* is the cursor position that it had over-skipped and *min* is the last cursor position before it had over-skipped.

One advantage of our skipping mechanism is that it can actually utilize the buffer pool. Most structural join algorithms are optimized by some B-Tree indices, which means that the buffer manager may not be able to fully utilize the pages loaded into the memory during the index lookup. However, skipping through tuples is just like skipping through leaf nodes of a B$^+$-Tree. So if we can skip leaf nodes without fetching the internal tree nodes into the buffer, then it results in better buffer utilization. Moreover, if the list is over-skipped, the recent skipped nodes are likely to be remained in the buffer pool. This benefit is demonstrated in the experiment section next.

5 Performance Evaluation

5.1 Experimental Setup

Our experiments were performed on a PC with Pentium IV, 2.8GHz, 512 MB physical RAM and 7200 RPM IDE hard disk. PostgreSQL version 7.3 was chosen as the back-end on Debian GNU Linux 3.0 with kernel build 2.4.22. We set the buffer pool to 8MB with 8K page size for PostgreSQL. All algorithms tested in our experiments use the same code base and were all implemented in C++.

To test the performance of our relabeling scheme, we used both *global* and *local* edge tables described in (Tatarinov et al. 2002). We use these two particular edge table because *local* edge table has the best order maintenance performance for updates in their experiments and *global* edge table has the worst relabeling performance. Moreover, the ideas expressed in (Tatarinov et al. 2002) are accepted by more recent work on querying ordered XML using SQL (Krishnamurthy et al. 2004, Grust et al. 2003).

We tested the performance of the relabeling scheme using both *global* and *local* encoding schemes with random and worst-case insertions. We also compared the performance of our scheme with the optimistic insertion strategy proposed in (Tatarinov et al. 2002). We tested the performance of both approaches based on the running time (in milliseconds) and average number of relabelings over total number of insertions.

We also tested the performance of SS-Join algoritm by comparing it with the STJ algorithms (both STJ-Desc and STJ-Anc) (Al-Khalifa et al. 2002), the B$^+$-Tree family of descendant joins (Chien et al. 2002, Haifeng et al. 2003) (including the ancestor join using XR-Tree) and the zig-zag structural join algorithm (Halverson et al. 2003). We used XR-Tree to represent the B$^+$-Tree family of structural joins, because (Haifeng et al. 2003) showed in their experiments that XR-Tree outperformed the B$^+$-Algorithm (Chien et al. 2002).

To give a fair indication of the real performance of the skipping strategies in structural joins. We modified all of the above structural join algorithms to return only the descendant or ancestor nodes instead of (*ancestor, descendant*) pairs. So, we have removed the use of stacks in the algorithm and avoid the overhead of maintaining stacks.

Our experiments used two real-world datasets (DBLP and MEDLINE) and one synthetic data set generated by XMark (Schmidt, Waas, Kersten, Carey, Manolescu &

Table 1: Test queries for structural joins

Query #	Path	Ancestors	Descendants	Result Count
D1	//inproceedings//pages	263,926	402,203	257,229
D2	//inproceedings//note	263,926	18	16
D3	/dblp/*//ee	420,606	175,613	175,613
D4	//MedlineCitation//Author	30,000	117,213	117,213
A1	//inproceedings[//pages]	263,926	402,203	257,228
A2	//inproceedings[//note]	263,926	18	16
A3	/dblp/*[//ee]	420,606	175,613	173,291
A4	//MedlineCitation[*]	30,000	117,213	30,000

Busse 2003) XML generator. Each datasets have between 5.5 - 9.8 million nodes. Table 1 lists the test queries used for the experiment. For each query, the running time is measured by taking an average of several consecutive runs with hot buffers. We stop each run when it exceeds 10 minutes. We also omit single path expression from the test queries, because this can be solved by a simple join of the *Path* table (Shimura et al. 1999, Yoshikawa et al. 2001) and the *Edge* table (Florescu & Kossmann 1999), as described in (Tatarinov et al. 2002). Basically, the *Path* table has the similar functions as DataGuides do in native XML systems.

Queries D1 to D3, A1 to A3 use the DBLP data set, D4 and A4 use the MEDLINE data set. Queries D1 and D2 test the performance of descendant joins over data sets that have high selectivity. However, for D1 it has a higher descendant to ancestor ratio than D2. D3 tests the performance of joins when the selectivity is low (i.e., almost all candidate descendant nodes are the matched nodes). D3 also has a quite low descendant to ancestor ratio. D4 works in the similar way as D1 to D3, except that the data set used is MEDLINE. Queries A1 to A3 test the performance of ancestor join algorithms with the same parameters as D1 to D3. A4 and A5 do the similar things on MEDLINE data set. Apart from DBLP and MEDLINE data sets, XMARK generated data set is used for testing performance of ancestor and descendant joins with varying selectivity.

5.2 Performance of Relabeling Algorithm

Figures 2(a), 2(b), 2(c) and 3 compare the time and the number of relabelings for both XRU and optimistic strategy in both worst case and random scenarios. It has been demonstrated that the random insertion of nodes using *local* encoding scheme is generally faster than both *Dewey* and *global* encoding schemes (Tatarinov et al. 2002). It is based on the assumption that elements usually have only a few child nodes, so any insertion only needs to relabel the sibling position of the affected nodes. However, the worst case of XRU using *local* encoding scheme is the same as that of *global* encoding scheme, when we keep inserting child nodes to an element in the database, which requires relabeling of sibling positions of child nodes at some stages. The horizontal expansion simulates the vertical expansion in *global* encoding scheme. Therefore, in Figures 2(a) and 2(b), we plot one curve for XRU using both *global* and *local* encoding schemes.

5.2.1 Global Encoding Scheme

The worst case scenario for XRU relabeling scheme occurs when new nodes are inserted continuously at the same position, for example, continuously inserting the node at the very beginning of the XML document. This causes the gap between the new node and the next node to shrink much quicker than it would otherwise. Therefore, the chance of having conflicts for assigning new tags increase as more nodes are inserted. When a conflict does occur, XRU moves its two cursors in both directions to determine the range of nodes that need to be relabeled and update their *start* attributes accordingly. This contributes to the amortized $O(\log(n))$ relabeling cost. The average number of relabelings performed in XRU in worst case scenario using the *global* encoding scheme is shown in Figure 2(a). The curve conforms to our previous claim that our extension of (Bender et al. 2002) can also achieve amortized $O(\log n)$ relabeling in the worst case scenario.

The worst case average insertion time for XRU algorithm using *global* is shown in Figure 2(b) as cross points, the curve increases almost in a sub-linear fashion instead of the expected amortized $O(\log n)$. This can be explained by the fact that every time we do relabeling of a node, we also need to re-arrange the B$^+$-Tree index (for the *start* attribute). Thus, the more we relabel, the more nodes we have to re-arrange in the index. However, in practice it is still an acceptable time with regards to the data size.

Under random insertions when using XRU and *global* encoding scheme, we see that both the number of relabelings and insertion time dropped significantly. This is due to the nature of random insertion, where the insertion point for nodes occurs at different parts of the XML document. Therefore, the chance of having conflicts when inserting the new node is significantly less than the worst case scenario. The initial peak in Figure 2(c) suggests that a number of relabeling occurred at the very early stage, such that the subsequent insertions did not cause any conflicts between tags. Although the random insertion time for XRU is still linear, the average insertion time only increases at a very slow rate. In fact it is approximately 10 times faster than the worst case scenario. In practice, the worst case scenario happens when we are doing bulk insertion, which is a special case and we can avoid this by pre-allocate tags to new nodes before node insertions.

5.2.2 Local Encoding Scheme

Figure 2(a) shows that we have achieved the same performance as that of the *global* encoding scheme. Figure 2(b) shows that the average insertion time in the worst case scenario is also the same as the performance of XRU using *global* encoding. Figure 2(c) shows XRU has the same pattern of node relabeling using *local* encoding. However, readers might have noticed that the average insertion time for random insertion using XRU and *local* encoding is significantly faster than using *global* encoding. This can be contributed by the fact that in *global* encoding, each element has a different document order tag and the tags are unique. However, using *local* encoding, many nodes can have the same sibling position. So the fanout of the B$^+$-Tree is much smaller than the B$^+$-Tree used in *global* encoding scheme.

5.2.3 XRU vs. Optimistic

So far, we have compared performance of XRU using both *global* and *local* encoding schemes under different scenarios (worst case and random). The experimental results showed XRU using *global* encoding scheme performs slower than *local* encoding scheme in general, except for the worst case scenarios.

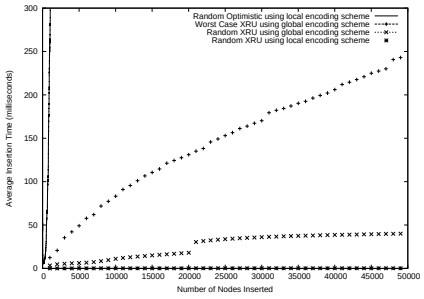
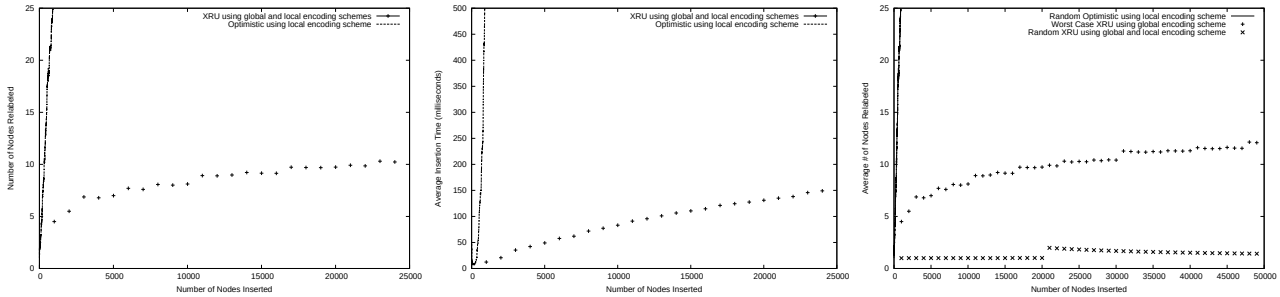


Figure 3: Average running time under random insertions

In (Tatarinov et al. 2002), they showed that optimistic



(a) Average # of relabelings under worst case scenario

(b) Average running time under worst case scenario

(c) Average # of relabelings under random insertions

Figure 2: Performance of updates

relabeling strategy performed well using *local* encoding scheme. Here, we compare the optimistic relabeling strategy using *local* encoding scheme against XRU relabeling scheme using *global* encoding. The line curve in Figure 2(a) and 2(b) show that under worst case scenario, optimistic relabeling is clearly outperformed by XRU. This is mainly contributed by the fact that when a conflict occurs during optimistic insertion, we have to re-label almost all sibling nodes after the affected node. With the same reason behind, Figures 2(c) and 3 showed that under random insertion, optimistic approach is also outperformed by XRU strategy.

5.3 Performance of Querying XML

5.3.1 Descendant Queries

Figure 4(a) shows the running time for each descendant queries. It is interesting to see that for queries D1, D3 and D5, Zig-Zag algorithm performed significantly worse than other structural join algorithms. This is because Zig-Zag join is an extension of the MPMGJN, which uses both *start* and *end* indices to achieve the zig-zag pattern of skipping between the ancestor and descendant lists. In the case of relational database, we need to maintain two indices for both *start* and *end* attributes of a node, using an index structure such as B^+-Tree. During the structural join process, when the current ancestor node is in the *preceding* axis of the current descendant node, we need to skip forward to find an ancestor node such that $aNode.end \geq dNode.start$. In this case, the relational database does a B-Tree index lookup on *end*. Generally, if we only skip descendant nodes, it should still be efficient since only the primary index (i.e., the *start* index) is needed for the lookup. However, in Zig-Zag, its zig-zag action causes the buffer manager to load in pages from the secondary index (i.e., from *end* index) to do a skip. Since the zig-zag algorithm only moves forward, the buffer pages loaded cannot be well utilized (at most it can be used to do more index lookup if the key is located on the same page). The overhead of frequent swapping the buffer pages in and out between the primary and secondary indices proved costly in the experiment. Readers may have noticed that for queries D2 and D4, Zig-Zag join performed quite well. In fact, it outperformed both XR-Tree and STJ-Desc joins. This is because for D2, the descendant node list is quite small (only 18 nodes instead of 402203 ancestor nodes), and the majority of the result descendant nodes are closely clustered. Therefore not many skipping actually happened. Finally, Zig-Zag join only skips either ancestor or descendant list if the very next node fetched from the list does not pass its test. This is well illustrated in D4, where the selectivity is 0 percent, and each ancestor on the ancestor list contains (i.e., matches) a descendant node from the descendant list.

For query D3 in Figure 4(a), we observe that STJ-Desc performs significantly worse than XR-Tree descendant join and SS-D join. This can be explained by Figure 4(c), which shows the performance of each algorithm under different selectivity[7]. The STJ-Desc performs well when the selectivity is low. However, as the selectivity increases, the number of unmatched nodes that STJ-Desc has to traverse also increases, which slows down its performance.

For descendant joins in relational databases, both XR-Tree/B^+-Tree and SS-Desc joins perform better than STJ-Desc and Zig-Zag algorithms. This is achieved by the descendant skipping mechanisms at both of the former algorithms. In order for XR-Tree or B^+-Algorithm to skip also the ancestor nodes, they have to access the secondary index just as the Zig-Zag join algorithm does. However, SS-Desc skips both ancestor and descendant nodes by moving both ancestor and descendant cursors forward with increasing gap size. After each skip, we do a check by comparing *start* and *end* between the cursors. This does not require secondary index lookup and hence minimize the number of page swaps between buffer manager and the disk.

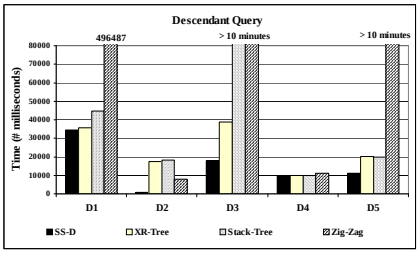
Figure 4(c) indicates that as the selectivity increases, the performance difference between XR-Tree descendant join and the SS-Desc join decreases. At the very first point in Figure 4(c), i.e., where the selectivity is 0%, all of the algorithms perform extremely well and approximately the same. This is because it responds to a query, D2, with the descendant node list containing only 18 nodes.

5.3.2 Ancestor Queries

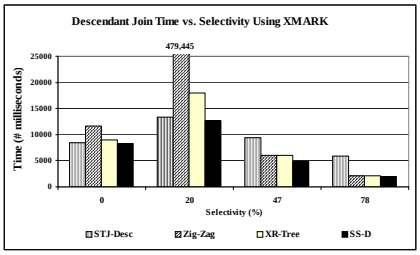
For ancestor joins, STJ-Anc appears to be the slowest on most ancestor join queries with the exception of query A5, where the selectivity is the lowest (i.e., every *MedlineCitation* has a descendant node). We also notice that Zig-Zag join performs quite well for ancestor joins with the exception of query A3. However, Figure 4(f) shows that A3 is a query with high selectivity. This suggests that we probably have to skip many ancestor nodes during the join process. For Zig-Zag algorithm, this means it has to frequently alternate the skipping between the ancestor and descendant node lists by accessing both primary and secondary indices. This result is still consistent with the outcome from the descendant queries experiment. On the other hand, Figures 4(f) shows Zig-Zag algorithm performs well when selectivity is low. This is due to the fact that it does less switching between the primary index lookup and the secondary index lookup.

For ancestor queries, our tests show that both SS-A and XR-Tree ancestor join algorithms perform well for

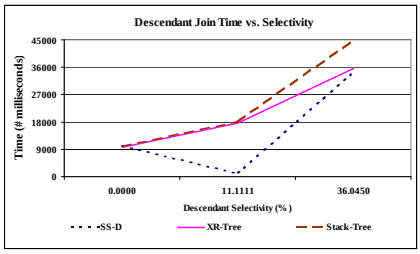
[7]For neatness, we omit plots for algorithms that have running time over 10 minutes here.



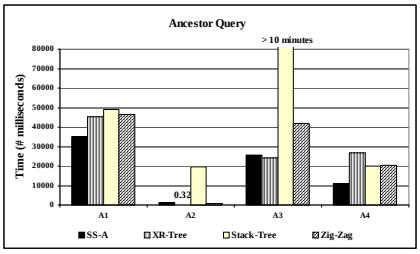
(a) Running time for different descendant queries



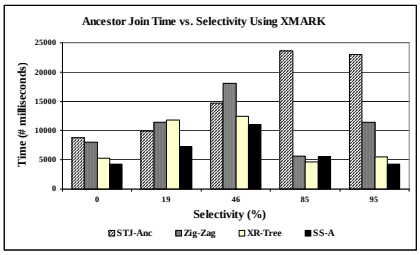
(b) Descendant joins with XMark and different levels of selectivity



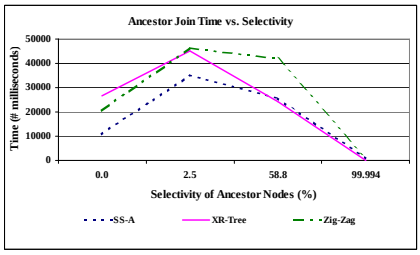
(c) Descendant joins with different selectivity



(d) Running time for different ancestor queries



(e) Ancestor joins using XMARK and different levels of selectivity



(f) Ancestor joins with different selectivity

Figure 4: Performance of different join algorithms for ancestor and descendant joins

Table 2: Ancestor joins at different % of selectivity

Selectivity (%)	0	19	46	85	95
STJ-Anc (milliseconds)	8768	9979	14660	23608	22943
Zig-Zag (milliseconds)	8045	11480	18102	5603	11401
XR-Tree (milliseconds)	5221	11776	12401	4608	5528
SS-A (milliseconds)	4268	7274	11049	5570	4304
$\%\Delta$ SS-A vs. the fastest out of STJ-Anc, Zig-Zag and XR-Tree	18.25	27.11	10.90	-20.88	22.14

Table 3: Descendant joins at different % of selectivity

Selectivity (%)	0	20	47	78
STJ-Desc (milliseconds)	8398	13275	9421	5969
Zig-Zag (milliseconds)	11639	479445	5981	2168
XR-Tree (milliseconds)	8938	17943	6011	2162
SS-D (milliseconds)	8325	12671	4827	1960
$\%\Delta$ SS-A vs. the fastest out of STJ-Anc, Zig-Zag and XR-Tree	0.87	4.55	19.29	9.34

queries with any selectivity and ancestor to descendant ratio. In particular, XR-Tree outperforms our SS-A by a small percentage when selectivity is high. However, in all other cases SS-A outperforms XR-Tree. XR-Tree is able to achieve a fast ancestor join because it maintains a *stab-list* by linking the list to the keys of their XR-Tree index. This allows the index to return all ancestors of a node with a key in minimal number of lookup. This effect is somewhat similar to skipping. However, when selectivity is low, every key will have a stab list, which adds extra overhead to the cost of the join operation. Figures 4(f) shows that SS-A behavior the same as SS-D algorithm does, where both of them skip ancestor as well as descendant nodes with increasing gap size.

5.3.3 Random Data Tests

We designed a range of queries from the randomly generated XMark data set to produce tests with different selectivity. Figures 4(e) and 4(b) show that SS-D outperforms other algorithms in any selectivity and SS-A also out-performs other algorithms in almost any selectivity. Tables 2 and 3 list the running time of SS-Join against other algorithms. The $\%\Delta$ shows the percentage difference in running time ($\Delta > 0$ if SS is faster, $\Delta < 0$ if SS is slower) of the SS-Join algorithm against the best performer out of Zig-Zag, XR-Tree and Stack-Tree Joins. It can easily be noticed that at 85% selectivity, XR-Tree outperforms SS-Join. This is because XR-Tree does well in ancestor joins when selectivity is high, which allows them to take advantage of the *stab-list* and save time by skipping ancestors.

5.4 Summary of Experimental Results

In general, we have showed that having fixed gap intervals between nodes for maintaining document order in relational databases works well only to an extent. Under frequent and random insertions, the XRU relabeling strategy assigns dynamic gaps for each insertion for achieving amortized $O(\log(n))$ cost, and it clearly out-performs the other approaches. Our experiments also show that both SS-A and SS-D perform well across different levels of selectivity, especially when the selectivity is between low to medium. Furthermore, our SS-A algorithm outperforms other ancestor join algorithms by at least 15-20% when the ratio of ancestor nodes to descendant nodes is between low to medium (i.e., 0 to 1).

6 Conclusion

We presented an efficient method to support containment queries for ordered XML data using a relational database system. Different from previous work, overheads for stack and B-Tree indices were avoided as neither of these data structures were used in our approach. We also extended our approach with an efficient update maintenance algorithm with performance that is better than previous approaches by a significant margin. Finally, the extensive experiments were presented and showed that our approach performed the best overall under various scenarios.

References

Al-Khalifa, S., Jagadish, H. V., Koudas, N. & Patel, J. M. (2002), Structural Joins: A Primitive for Efficient XML Query Pattern Matching, *in* 'ICDE'.

Bender, M. A., Cole, R., Demaine, E. D., Farach-Colton, M. & Zito, J. (2002), Two simplified algorithms for maintaining order in a list, *in* 'ESA', Vol. 2461 of *Lecture Notes in Computer Science*, Rome, Italy, pp. 152–164.

Bohannon, P., Freire, J., Roy, P. & Simeon, J. (2002), From XML schema to relations: A cost-based approach to XML storage, *in* 'ICDE'.

Bruno, N., Koudas, N. & Srivastava, D. (2002), Holistic twig joins: optimal XML pattern matching, *in* M. J. Franklin, B. Moon & A. Ailamaki, eds, 'SIGMOD Conference', ACM, pp. 310–321.

Carey, M. J., Florescu, D., Ives, Z. G., Lu, Y., Shanmugasundaram, J., Shekita, E. J. & Subramanian, S. N. (2000), XPERANTO: Publishing Object-Relational Data as XML, *in* 'WebDB (Informal Proceedings)', pp. 105–110.

Chaudhuri, S., Kaushik, R. & Naughton, J. F. (2003), On Relational Support for XML Publishing: Beyond Sorting and Tagging, *in* A. Y. Halevy, Z. G. Ives & A. Doan, eds, 'SIGMOD Conference', ACM, pp. 611–622.

Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V. J. & Zaniolo, C. (2002), Efficient structural joins on indexed XML documents, *in* 'VLDB Conference', Berlin, Germany, pp. 263–274.

Choi, B., Mahoui, M. & Wood, D. (n.d.), On the Optimality of Holistic Algorithms for Twig Queries, *in* 'DEXA', pp. 28–37.

Cohen, E., Kaplan, H. & Milo, T. (2002), Labeling Dynamic XML Trees, *in* 'PODS Conference', ACM Press, New York, pp. 271–281.

DeHaan, D., Toman, D., Consens, M. P. & zsu, M. T. (2003), A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding, *in* A. Y. Halevy, Z. G. Ives & A. Doan, eds, 'SIGMOD Conference', ACM, pp. 623–634.

Deutsch, A., Fernandez, M. & Suciu, D. (1999), Storing semistructured data with STORED, *in* 'SIGMOD Conference', pp. 431–442.

Dietz, P. F. (1982), Maintaining order in a linked list, *in* 'STOC', pp. 122–127.

ElSayed, M., Dimitrova, K. & Rundensteiner, E. A. (2003), Efficiently Supporting Order in XML Query Processing, *in* 'WIDM'.

Fernández, M., Kadiyska, Y., Suciu, D., Morishima, A. & Tan, W.-C. (2002), 'SilkRoute: A framework for publishing relational data in XML', *ACM Transactions on Database Systems* **27**(4), 438–493.

Florescu, D. & Kossmann, D. (1999), 'Storing and querying XML data using an RDMBS', *IEEE Data Engineering Bulletin* **22**(1), 27–34.

Goldman, R. & Widom, J. (1998), DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, *in* 'VLDB Conference', pp. 436–445.

Grust, T., van Keulen, M. & Teubner, J. (2003), Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps, *in* 'VLDB Conference', pp. 524–525.

Haifeng, J., Lu, H., Wei, W. & Ooi, B. C. (2003), XR-Tree: Indexing XML Data for Efficient Structural Join, *in* 'ICDE', IEEE Computer Society.

Halverson, A., Burger, J., Galanis, L., Kini, A., Krishnamurthy, R., Rao, A. N., Tian, F., Viglas, S., Wang, Y., Naughton, J. F. & DeWitt, D. J. (2003), Mixed Mode XML Query Processing, *in* 'VLDB Conference', pp. 225–236.

Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y. & Yu, C. (2002), 'Timber: A native xml database', *VLDB Journal* **11**(4), 274–291.

Jiang, H., Wang, W., Lu, H. & Yu, J. X. (2003), Holistic Twig Joins on Indexed XML Documents, *in* 'VLDB Conference', pp. 273–284.

Krishnamurthy, R., Kaushik, R., Naughton, J. F. & Chakaravarthy, V. T. (2004), Recursive xml schemas, recursive xml queries, and relational storage: Xml-to-sql query translation, *in* 'ICDE'.

Manolescu, I., Florescu, D. & Kossmann, D. (2001), Answering XML Queries on Heterogeneous Data Sources, *in* 'VLDB Conference', pp. 241–250.

O'Neil, P. E., O'Neil, E. J., Shankar Pal, I. C., Schaller, G. & Westbury, N. (2004), ORDPATHs: Insert-Friendly XML Node Labels, *in* 'SIGMOD Conference', ACM, pp. 903–908.

Schmidt, A., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I. & Busse, R. (2003), Assessing XML data management with XMark, *in* S. Bressan, A. B. Chaudhri, M.-L. Lee, J. X. Yu & Z. Lacroix, eds, 'EEXTT', Vol. 2590 of *Lecture Notes in Computer Science*, Springer, pp. 144–145.

Shanmugasundaram, J., Kiernan, J., Shekita, E. J., Fan, C. & Funderburk, J. (2001), Querying XML Views of Relational Data, *in* 'VLDB Conference', pp. 261–270.

Shanmugasundaram, J., Shekita, E. J., Kiernan, J., Krishnamurthy, R., Viglas, S., Naughton, J. F. & Tatarinov, I. (2001), 'A General Techniques for Querying XML Documents using a Relational Database System', *SIGMOD Record* **30**(3), 20–26.

Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D. J. & Naughton, J. F. (1999), Relational databases for querying xml documents: Limitations and opportunities, *in* 'VLDB Conference', pp. 302–314.

Shimura, T., Yoshikawa, M. & Uemura, S. (1999), Storage and Retrieval of XML Documents Using Object-Relational Databases, *in* 'DEXA', pp. 206–217.

Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E. & Zhang, C. (2002), Storing and Querying Ordered XML Using a Relational Database System, *in* 'SIGMOD Conference'.

Wu, X., Lee, M. L. & Hsu, W. (2004), A Prime Number Labeling Scheme for Dynamic Ordered XML Trees, *in* 'ICDE'.

Yoshikawa, M., Amagasa, T., Shimura, T. & Uemura, S. (2001), 'XRel: a path-based approach to storage and retrieval of XML documents using relational databases', *TOIT* **1**(1), 110–141.

Zhang, C., Naughton, J. F., DeWitt, D. J., Luo, Q. & Lohman, G. M. (2001), On supporting containment queries in relational database management systems, *in* 'SIGMOD Conference'.
*citeseer.nj.nec.com/zhang01supporting.html