InterJoin: Exploiting Materialized Views in XML Query Processing

Derek Phillips, Ning Zhang, Ihab F. Ilyas, and M. Tamer Özsu

School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada N2L 3G1
{djphilli,nzhang,ilyas,tozsu}@uwaterloo.ca

University of Waterloo
Technical Report CS-2005-29

Abstract. Efficient processing of XPath expressions is an integral part of XML data query processing. Exploiting materialized views in query processing can significantly enhance query processing performance. We propose a novel view definition that allows for intermediate (structural) join results to be stored and reused in XML query evaluation. Unlike current XML view proposals, our views do not require navigation in the original document or path-based pattern matching. Hence, they can be evaluated significantly faster and can be more easily costed as part of a query plan. In general, current structural joins cannot exploit views efficiently when the view definition is not a prefix (or a suffix) of the XPath query. To increase the applicability of our proposed view definition, we propose a novel physical structural join operator called InterJoin. The InterJoin operator allows for joining interleaving XPath expressions, e.g., joining //A//C with //B to evaluate //A//B//C. InterJoin treats structural joins as a logical operator, giving more join alternatives in XML query plan. We propose several physical implementations for InterJoin, including a technique to exploit spatial indexes on the inputs. We give analytic cost models for the implementations so they can be costed in an existing join-based XML query optimizer. Experiments on real and synthetic XML data show significant speed-ups of up to 200% using InterJoin, as well as speed-ups of up to 400% using our materialized views.

1 Introduction

In the last few years, much research has focused on efficiently answering path queries on an XML database. Regardless of whether the XML database is implemented natively [1, 2] or as a separate layer of an RDBMS [3, 4], efficient processing of XML queries requires specialized physical operators [5]. Several techniques have been proposed for computing the results of XPath queries in an

XML query processor. These include navigational techniques that scan an XML document to find the desired output nodes (e.g., Y-Filter [6], XTrie [7], and TurboXPath [8]), join-based techniques that apply structural joins to tag indexes (including binary structural joins [9, 5] and holistic joins [10]), and hybrid-techniques that combine navigation and join techniques (e.g., BlossomTrees [11]).

Many XQuery FLWOR statements contain several related XPath expressions. For example, consider the following query $\mathcal{Q}$, from the XQuery Use Cases:

```
Q: for $b in doc("bib.xml")//book
   where $b/publisher = "Addison-Wesley"
         and $b/@year > 1991
   order by $b/title
   return
     <book>
       { $b/@year }
       { $b/title }
     </book>
```

Query $\mathcal{Q}$ consists of several related XPath expressions. As in relational database systems, we can reduce the processing cost by identifying commonalities among expressions, and evaluating the expressions incrementally. In this paper, we address the problem of reusing materialized views by extending the applicability of structural joins.

1.1 Motivation

We consider the structural join operator as a logical operator that takes as input the results of any two "joinable" subexpressions (i.e. related by one or more descendant or child constraints) of an XPath expression. Current structural join implementations (such as MPMGJN [5] and the stack-based method [9]) restrict the space of join plans according to the limitations of the physical structural join operators. In particular, these methods are limited to joining the results of subexpressions that are adjacent in the XPath expression. For example, let P_1 and P_2 be subexpressions of an XPath expression P. If there is a single descendant or child axis relating the two expressions in P (e.g., $P = P_1/P_2$), then current structural join implementations can be used. However, if we have $P_1 = P_3//P_5$ and $P_2 = P_4//P_6$, and $P = P_3//P_4/P_5//P_6$ then current structural join implementations can not join these results efficiently.

We propose the InterJoin operator, a novel physical implementation of structural joins that can interleave the results of any two joinable subexpressions. Augmenting structural join processors with the InterJoin operator expands the space of query plans that can be processed efficiently. InterJoin is a merge-based operator that takes advantage of optimization opportunities in two cases:

1. If the selectivity of two non-adjacent subpaths in an XPath expression is high, we can join these nodes first to reduce temporary result sizes. Consider the query `//book//author//name[. = 'John Smith']` on a DBLP-style database. We expect many occurrences of authors named 'John Smith'

in journal articles, so the most selective operation is a structural join that computes `//book//name[. = 'John Smith']`. We can evaluate this expression first and then use InterJoin to join the results with `author` nodes.
2. If a portion of a query is expensive and it can be reused for other queries, then we should compute the results and materialize them as a view. Future queries can be processed using this view to save expensive computation. For example, if `y` and `z` are expensive path queries and we need to compute `//y/A/B//z/K//L`, `//y//C/z`, and `/y//D//E//F//z`, we can materialize the results of `//y//z` as a view and use the structural join and InterJoin operators to compute the results for all three queries.

A common problem with existing XML view proposals is that they generally focus on the resulting nodes of an XPath query. For example, a view on `//A//B` stores information about the `B` node results but throws away any information about the `A` node(s) with which they matched. We propose storing the interval-encoding of all nodes matched in an XPath expression. This provides sufficient information to computer structural joins on the views.

A key feature of our proposed materialized views is that the view results use a tuple-based representation similar to that of operators in current XML query processors, such as Timber [12], Rainbow [13], and TurboXPath [8]. Thus, a materialized view is a valid input to existing operators, and the output of these operators can be materialized as a view. This allows significant flexibility in the way that views are integrated into a query plan.

It is important to note that the InterJoin operator is not intended to introduce a bushy query plan approach to evaluating XPath expressions; bushy query plans for structural joins have already been proposed by Wu et al. [14]. Instead, the InterJoin operator increases the number of joinable XPath expressions. For example, in a bottom-up relation-style query plan generator, the InterJoin operator allows for more ways to join results computed at level i in order to generate results at level $i + 1$. In other words, we propose using structural join as a logical operator accepting any joinable inputs, as opposed to a physical operator that expects inputs of a specific type. This allows us to explore a larger query plan space with fewer constraints on the legal inputs to each structural join operator.

1.2 Contributions

The main contributions of the paper are as follows:

- We propose a new definition for views that can be integrated into a join-based XML processor and can be used to boost query processing performance.
- We propose efficient algorithms for a novel InterJoin physical operator. The InterJoin operator extends the query plan space for structural join-based processors and provides more possibilities for the use of materialized views.
- We provide cost models for the algorithms so that the CPU and I/O costs can be estimated in a query plan. This allows the InterJoin operator to be integrated into a cost-based optimizer.

– We provide experimental results that demonstrate the improvements obtained using the InterJoin operator in a query plan, as well as the speed-ups that can be achieved with our proposed views.

The rest of the paper is organized as follows. In Section 2, we introduce some background concepts. In Section 3, we present the major contributions of the paper: our view definition proposal and several implementations of the InterJoin operator. In Section 4, we present cost estimates for the InterJoin operator. We experimentally study the performance of the algorithms in Section 5. Related work is described in Section 6. Finally, in Section 7 we state our conclusions.

2 Preliminaries

We consider path queries on XML databases consisting of forests of rooted, ordered, labeled trees. These documents can be stored using any of the proposed physical storage mechanisms (e.g., shredded into relations, B-Trees, etc.); however, we require a constant-time method to check if one XML node is a descendant or child of another. Assigning an appropriate interval-encoding label to each node provides this functionality. For example, the XPath Accelerator ID [15] allows for constant-time checks for these relationships.

Query Pattern Matching Our focus is on pattern matching, in the sense described by Wu et al. [14], where we have a rooted node-labeled tree $T = (V_T, E_T)$, representing the database, and a smaller rooted node-labeled tree $Q = (V_Q, E_Q)$, called the *query pattern*. The edges in Q are labeled as either *child* or *descendant*. Figure 1 shows an example of a simple query pattern tree (QPT) for an XPath query with single edges to represent child axes and double edges for descendant axes. Evaluating a query Q involves finding all nodes in T that represent a total mapping from nodes in Q to those in T. Formal definitions of these concepts are given in Section 2.1 of [14].

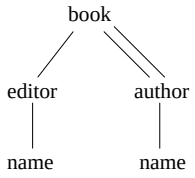

Fig. 1. A simple pattern tree.

Structural Joins A number of techniques have been proposed for binary structural joins (e.g., [9, 5]). A structural join operator outputs all node tuples that satisfy the relationship specified by a single XPath (descendant or child) axis, which is equivalent to an edge in a QPT.

There are two key requirements in order for this approach to work. First, in order to perform a structural join on the tuple outputs of two XPath expressions, the tuples must have the nodes of interest in document order. The other requirement for joining two XPath expression results is that they represent non-interleaving expressions from the XPath query. For example, the query `//A//B//C` can be evaluated either as `(A` $\bowtie_{sj}$ `B)` $\bowtie_{sj}$ `C` or as `A` $\bowtie_{sj}$ `(B` $\bowtie_{sj}$ `C)`, where `A,B,C` are lists of XML nodes and $\bowtie_{sj}$ is a structural join. The third option, `(A` $\bowtie_{sj}$ `C)` $\bowtie_{ij}$ `B`, where $\bowtie_{ij}$ is an InterJoin operator that joins tuples of `(A,C)` matches with `B` matches, is not handled efficiently with current structural join algorithms.

Matches and Tuple Orderings Throughout the paper, we will use the term *matches* to indicate the tuple outputs of a join operator. For example, the set of `(A,B)` matches corresponds to the tuples output by evaluating `//A//B` (or `//A/B`).

An important part of XML processing is handling the document order of XML nodes. Suppose we have a list of tuples of the form $(A_1, A_2, ..., A_k)$. We say that the list is sorted by $(A_{i_1}, ..., A_{i_\ell})$, if the tuples are first sorted by the document order of A_{i_1}, then the document order of A_{i_2}, and so on. We call the first and second sort nodes the *primary* and *secondary* sort nodes, respectively.

3 Materialized Views and InterJoin

In this section, we propose a method for storing materialized views that can be used in combination with other view proposals, such as [16]. Our views differ from existing proposals in that the information stored in our views allows them to be used in a join-based XML processor. To take full advantage of these views, we propose four implementations of a new InterJoin physical operator that can be used in a structural-join based processor.

3.1 Temporary Results as Materialized Views

Balmin et al. [16] propose four classes of XPath views. The most relevant to our discussion are the *path* views, which store a list of ancestor tags and a value for each node output by a path expression. Value predicates can be applied to the results of a query and subsequence matching can be used to filter according to (ancestral) structural constraints. Structural joins can not be applied to these views since they do not contain sufficient structural information.

We propose a class of views storing a tuple of XML node encodings corresponding to each axis step in an XPath expression. The node encoding method must support efficient ancestor and parent checking. These views now contain sufficient information to be used in structural join processors.

For example, if we compute a view for `//book[@price > 40]//author/name`, we can use structural joins, InterJoin, and selection operators to answer:

```
//book[@price > 100]//author/name,
//book[@price > 40]//author/name/lastName[fn:contains(., 'King')],
/mall/bookStore/book[@price > 40]//author/name
//book[@price > 40]/contributors/author/name.
```

3.2 InterJoin: Physical Structural Join Operators

In this section, we present several implementations of the binary InterJoin operator which assume that an interval encoding is used for the nodes. The algorithms use the following constraints to check if an `(A,C)` pair matches with a `B` node: (1) $A.open < B.open$, (2) $B.open < C.open$, and (3) $C.open < B.close$.

InterJoinNR: InterJoin over Non-recursive Documents The simplest case for performing an InterJoin of `(A,C)` and `B` matches occurs when the `B` nodes are non-recursive (i.e., no `B` node is a descendant of another `B` node). In this case, every `(A,C)` pair can match with at most one `B` node. We can perform an InterJoin by keeping a single `B` node buffered and loop through all `(A,C)` pair inputs. This requires that the input `(A,C)` pairs be ordered by `(C)` and that the `B` list be in order. The output will be ordered by `(C,B)` and if the `A` nodes are also non-recursive, then the output will have the `A`, `B`, and `C` nodes all in document order.

Note that if the `B` nodes are recursive, we can not use a similar technique of buffering a single `(A,C)` pair. This is because a `B` node will match with several `(A,C)` pairs and all or some of these may match with one of its descendant `B` node. In this case, we can use one of the other InterJoin implementations.

InterJoinSL: InterJoin with the Ancestor-Match Property We can efficiently compute an InterJoin when the inputs satisfy the *ancestor-match property*. This property guarantees that if A_1 is an ancestor of A_2 and `(`A_2`,`C_1`)` is a match, then `(`A_1`,`C_1`)` is a match. Note that if there is a predicate relating the subexpressions that generated the `A` and `C` match lists (e.g. in a complex view), then this property may not hold. The following XQuery Q returns results where the `(cat, item)` pairs do not have the ancestor-match property:

```
Q:  for $c in doc("cat.xml")//cat
      for $i in $c//item
      where $i/@price < 5 and $c/@price + $i/@price > 10
```

In a case where the inputs are known to have the ancestor-match property, we can use the InterJoinSL algorithm shown in Algorithm 1. This algorithm keeps a stack of `A` nodes and a single list of `C` nodes corresponding to those nodes matched by the bottom `A` node on the stack. The other `A` nodes on the stack keep a pointer (in the form of a list index) to the last `C` node with which they matched. The ancestor-match property guarantees that any `A` node on the stack will only match consecutive nodes beginning at the head of the `C` list. The algorithm requires that the `(A,C)` inputs be ordered by `(A,C)` and outputs the results ordered by `(B,A)`. A simple modification to the algorithm allows output tuples to be ordered

Algorithm 1 InterJoin with a single C node list.

```
InterJoinSL(AC : NodePairList, B : NodeList)
 1  Initialize empty stack S and list CList
 2  while AC, S, and B not empty
 3    do minOpen ← min(AC_top.A.open, B_top.open);
 4       removed ← 0
 5       while CList.first() < minOpen
 6         do CList.removeFirst()
 7            removed ← removed +1
 8       for each a ∈ S
 9         do a.end ← a.end − removed
10            if a.end < 1
11                 S.pop()
12       if AC_top.A.open < B_top.open
13            a ← new stack node;
14            a.end ← 1;
15            curA ← AC_top.A.open;
16            AC.next()
17            while AC_top.A.open = curA
18              do a.end ← a.end +1;
19                 AC.next()
20            S.push( a )
21       else
22            for each a ∈ S
23              do for each i = 1 to a.end
24                   do Stop when CList[i].open > B_top.close
25                      Output (a, B_top, c)
26                 B.next()
```

by `(B,C)`. A further modification (similar to the ones proposed in [9]) allows us to buffer tuples and output the results ordered by `(C,B)` and `(A,B)`.

InterJoinML: InterJoin using Multiple C-node Lists The second implementation uses a stack of `A` nodes where each item in the stack maintains its own list of `C` nodes. These lists correspond to the matched `(A,C)` pairs in the input. The algorithm requires that the input `(A,C)` pairs be ordered by `(A,C)` and produces outputs ordered by `(B,A)`. Buffering can be used to output matches ordered by `(A,B)`. The pseudocode is given in Algorithm 2.

InterJoinPQ: InterJoin using a Priority-Queue The third implementation uses a priority queue keyed on `(C.open, A.open)` and requires that the inputs be ordered by `(A)`. The pseudocode is given in Algorithm 3.

The algorithm outputs tuples ordered by `(B,C)`. If we buffer tuples, then the output can be ordered by `(C,B)`. A modification to the InterJoinPQ algorithm allows it to handle input tuples ordered by `(C)`. In this case, the priority

Algorithm 2 InterJoin with multiple C node lists.

InterJoinML(AC : NodePairList, B : NodeList)

```
1  Initialize empty stack S
2  while AC, S, and B not empty
3    do minOpen ← min(AC_top.A.open, B_top.open)
4       for each a ∈ S
5         do while a.CList[1].open < minOpen
6              do a.CList.removeFirst()
7       if AC_top.A.open < B_top.open
8            a ← new stack node
9            curA ← AC_top.A.open
10           while AC_top.A.open = curA
11             do a.Clist.add(AC_top.C.open)
12                AC.next()
13           S.push( a )
14      else
15           for each a ∈ S
16             do for each c ∈ a.CList
17                  do Stop when c.open > B_top.close
18                     Output (a, B_top, c)
19                B.next()
```

Algorithm 3 InterJoin with a priority queue.

InterJoinPQ(AC : NodePairList, B : NodeList)

```
1  Initialize empty priority queue PQ
2  while AC, PQ, and B not empty
3    do minOpen ← min(AC_top.A.open, B_top.open)
4       while PQ_top.C.open < minOpen
5         do PQ.removeTop()
6       if AC_top.A.open < B_top.open
7            PQ.add( AC_top ); AC.next()
8       else
9            pq ← PQ_top
10           while pq.C.open < B_top.close
11             do Output (pq.A, B_top, pq.C); pq ← PQ.next()
12           B.next()
```

queue is keyed on `(A.open, C.open)` and we need to read `(A,C)` tuples until $AC_{top}.C.open > B_{top}.open$. The output can be ordered by `(B,A)` or `(A,B)`.

InterJoinRT: Index-InterJoin Operator (R-Trees) We can implement the InterJoin operator as a spatial join between node intervals. For inputs `(A.open,C.open)` and `(B.open,B.close)`, the two intervals must overlap such

Algorithm 4 R-Tree search for B matches.

RTreeSearch(RT : RTree, AC : Rectangle)

```
1  PQ ← empty priority queue
2  PQ.add(RT.root)
3  while PQ not empty
4    do if PQ_top is a leaf
5            Output match with PQ_top
6       else
7            temp ← PQ.removeTop()
8            for each child ch of temp
9              do if ch intersects AC
10                    PQ.add(ch)
```

that `C.open` is in the `B` interval but `A.open` is not, or equivalently that `B.open` is in the `AC` interval but `B.close` is not.

We can impose these constraints using region queries in the plane. Given a point `(A.open,C.open)`, we build a rectangle with top-left and bottom-right corners `(A.open, ∞)`, `(C.open,C.open)`. A `B` node matches with `(A,C)` precisely when the point `(B.open,B.close)` is contained in the rectangle. Similarly, if we are given a `B` node, then a point `(A.open,C.open)` will match if it is contained in the rectangle `(0, B.close)`, `(B.open,B.close)`. Figure 2 depicts how these constraints appear in 2-D.

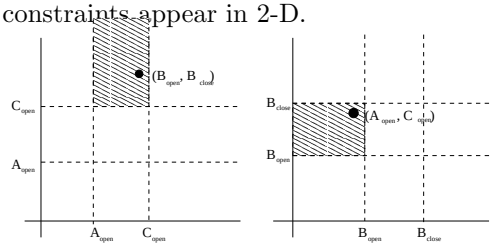

Fig. 2. InterJoin join conditions for the R-tree based algorithms in 2-D.

We can now build an R-tree on points representing items from one input list and then probe the R-tree using rectangles built from the other input list. We can enforce order on the results by modifying the R-tree search algorithm to use a priority queue. Algorithm 4 describes the new R-tree search algorithm and Algorithm 5 presents InterJoinRT, using `(A,C)` pairs in the R-tree. Using the search algorithm, we can output results ordered by `(B,A)`, `(B,C)`, and `(A,B)`, regardless of the input orders. Note that one advantage to the R-tree approach is that neither of the input lists need to be sorted. Algorithm 6 gives the analogous algorithm when the R-Tree is built on `(B.open,B.close)` pairs.

The R-tree structure can be generalized from the 2-dimensional case using the open index of several nodes in a query. For example, the points in an R-tree for the materialized view of `//A//C//F//H` will be 4-dimensional points of the form `(A.open, C.open, F.open, H.open)`. The input to the search algorithm

Algorithm 5 InterJoin with R-tree on AC nodes.

InterJoinRT_AC(AC : NodePairList, B : NodeList)

```
1  RT ← empty R-tree
2  for each pair (a,c) in AC
3    do Add point (a.open,c.open) to RT
4  for each item b in B
5    do Build rectangle R := (0, b.close), (b.open, b.close)
6       for each point ac in RT that intersects R
7         do Output (ac.a,b,ac.c)
```

Algorithm 6 InterJoin with R-tree on B nodes.

InterJoinRT_B(AC : NodePairList, B : NodeList)

```
1  RT ← empty R-tree
2  for each item b in B
3    do Add point (b.open,b.close) to RT
4  for each pair (a,c) in AC
5    do Rectangle R := (a.open, ∞), (c.open, c.open)
6       for each point b in RT that intersects R
7         do Output (a,b,c)
```

will now be a multi-dimensional box with boundaries in two of the dimensions and minimum and maximum values of 0 and ∞ in every other dimension. This is a projection of the multi-dimensional points into the two dimensions of interest and a region query in the plane.

With the new R-tree search procedure, we can use this materialized view to perform InterJoin operations at any point in the original query. For example, we can answer queries such as `//A//B/C//F//H` and `//A//C//F//G//H`.

3.3 Complex Interleaving and Predicates

So far, we have considered the case where InterJoin is used to join path expressions consisting of single nodes. In many cases, interleaving joins are required. For example, if we have a materialized view for `//V//Z` and another for `//W/X/Y`, we may want to use these views to answer the query `//V//W/X/Y/Z`. This is a more difficult problem because we need to check for the `V//W` relationship and the `Y/Z` relationship simultaneously. Similarly, if we have a materialized view for `//H//J` and another for `//I//K`, we may want to use these views to answer the query `//H//I//J//K`.

We solve this problem in an ad-hoc manner. For the first example, we use the InterJoin operator to join the `(V,Z)` tuples in the first view with the `(W,X,Y)` tuples, using the `W` nodes for the join (i.e. as the `B` nodes in the algorithm descriptions). Before a match is output, we check each potential output match to ensure that the corresponding `Y` node is a parent of the `Z` node. Another

approach is to apply an InterJoin using the `(V,Z)` entries in the first tuple and the `Y` entry in the second tuple. Filter the results to ensure that each `W` node is a descendant of `V`.

The second example can be solved in a similar manner by applying an InterJoin with the `(H,J)` entries from the first view tuples and the `I` entries from the second view tuples, using a filter to ensure that the `K` node in a potential match is a descendant of the `J` node. This situation can be generalized to arbitrary levels of interleaving, however every interleaving increases the number of filters.

Predicates can also be included with our materialized views and InterJoin operators to answer queries. For example, we can create a view on `//person//age[. > 50]` which will consist of tuples of the form `(person,age)` for all people aged over 50. This can now be used to answer queries adding additional structural or value predicates, such as the following queries:

```
//department//person//age[. > 50]
//person/info/personal/age[. > 50]
//person//personal[@private=0]//age[. > 50]
```

4 Analysis

In this section, we investigate the complexity of the InterJoin algorithms and develop cost models of the CPU and I/O costs. The cost models allow for the algorithms to be integrated into a cost-based structural join processor.

4.1 Complexity of the InterJoin Algorithms

The InterJoinSL algorithm maintains a stack of A nodes and a single list of C nodes. Insertions and deletions for each structure take constant time, and updating the *end* field for each stack node can be done without revisiting any C nodes in the list. Thus, the algorithm requires $O(|AC| + |B|)$ time.

The InterJoinML algorithm maintains a stack of A nodes and each stack node has a list of C nodes. Insertions and deletions take constant time, giving an optimal runtime of $O(|\,AC\,| + |B|)$.

The InterJoinPQ algorithm adds (and removes) each AC pair at most once to the priority queue. In the worst case, the priority queue can have size $O(|\,AC\,|)$, shown in the following example document X:

```
<A>
  <A>
    ...
      <A>
        <B>
          <C/>
          <C/>
          ...
          <C/>
```

```
        </B>
      </A>
    ...
  </A>
</A>
```

There are at most h nested `A` nodes, where h is the height of the XML tree; thus, there can be at most $h \cdot C_{dist}$ items in the priority queue, where C_{dist} is the number of distinct C nodes (i.e. nodes that matched with at least one A node) in the input pair list. The maximum size of the priority queue will be $PQ_{max} = \min(|\,AC\,|, h \cdot C_{dist})$, so the worst-case runtime is $O(|\,AC\,|\lg(PQ_{max}) + |B|)$. The space required is $O(PQ_{max})$.

The space usage for InterJoinPQ and InterJoinML is $O(|\,AC\,| + h \cdot C_{dist})$ since we may need to buffer all `(A,C)` tuples (as in the worst-case example shown above). The InterJoinSL algorithm requires only $O(h + C_{dist})$ since we only buffer one copy of each A and C node.

Each of the algorithms can be modified to buffer matches in order to output the results in the orders described in Section 3.2. Using a similar argument to the one given by Al-Khalifa et al. [9], we can show that the asymptotic runtime and space for each of the algorithms remains the same after these modifications.

4.2 Analytic Cost Models

A cost-based optimizer needs a cost model for each physical operator. In this section, we present analytic cost models for the new physical operators based on path statistics. Statistics maintained by the database can be used to estimate the costs in order to facilitate query plan optimization and pruning. We break down the cost of an operator into its I/O cost and CPU cost. In relational databases, the dominant factor is the I/O cost; however, since the XML operators are more complex than relational operators, it has been shown that CPU cost can be upwards of 30% of the total cost [8].

The cost of the InterJoinSL algorithm is determined by the input size, the output size, the stack size, and the $CList$ size. Define $T(X_1X_2...X_k)$ as the number of tuples satisfying $//X_1//X_2//...//X_k$. If `A` is non-recursive, then $T(AC) = |//A//C|$; if not, then we can use path statistics to get:

$$\begin{aligned} T(AC) &= |//A//C| + |//A//A//C| + \cdots \\ &\quad + |\underbrace{//A \cdots //A}_{MaxRec}//C| \\ &= \sum_{i=1}^{MaxRec} |(//A)^i//C|, \end{aligned}$$

where $MaxRec$ is the maximum recursion depth of `A`, (i.e., the maximum number of `A`s that appear in a root-to-leaf path) and $(//A)^i$ denotes i concatenations of the string `//A`. The input and output sizes $|AC|$ and $|ABC|$, will be $T(AC)$ and $T(ABC)$, respectively.

Thus, the CPU and I/O costs of InterJoinSL are:

$$CPU \approx \alpha_1 \cdot T(AC) + \alpha_2 \cdot |B| + \alpha_3 \cdot T(ABC) + \alpha_4 \cdot |//A//C|$$

$$IO \approx \alpha_5 \cdot T(AC) + \alpha_6 \cdot |B| + \alpha_7 \cdot T(ABC)$$

where the α_i's represent implementation-dependent costs for a single operation.

The cost of InterJoinML is determined by the input size, the output size, the stack size, and the sum of the $CList$ sizes. For each input `(A,C)` tuple, the A node will be put on the stack S, if it is not already there, and the C node will be put into the $CList$. Therefore, the number of pushes and pops of S is at most $|//A|$, and the number of insertions and deletions in the $CList$s is at most $|AC|$. Therefore, the number of update operations on S and the $CList$s is at most $2 * (|//A| + |AC|)$.

In addition to the update operations, there are lookup operations to the heads of the $CList$s (lines 4-5) that do not contribute to updates. The number of lookup operations is determined by the size of S (i.e., the recursion depth of the A nodes) and the number of A and B elements at that recursion depth. Define $W(AB)$ in the following manner:

$$\begin{aligned} W(AB) &= 1 * |//A//B| + 2 * |//A//A//B| \cdots + MaxRec * |\underbrace{//A \cdots //A}_{MaxRec}//B| \\ &= \sum_{i=1}^{MaxRec} i * |(//A)^i//B|. \end{aligned}$$

Now, if the ancestor-match property holds, then the number of lookup operations is given by $W(AB)+W(AA)$. If the property does not hold, then the number can be estimated using the proportion of the input size $|AC|$ to $T(AC)$. Similarly, the output size $|ABC|$ can be estimated using the same proportion and $T(ABC)$ in the following way:

$$T(ABC) = \sum_{i=1}^{MaxRec} \sum_{j=1}^{MaxRec} |(//A)^i(//B)^j//C|$$

Thus, the CPU and I/O costs of InterJoinML are:

$$\begin{aligned} CPU \approx \quad & \alpha_8 \cdot |//A| + \alpha_9 \cdot |AC| + \alpha_{10} \cdot |B| \\ & + \frac{|AC|}{T(AC)} \cdot \Big(\alpha_{11} \cdot \big(W(AB) + W(AA)\big) + \alpha_{12} \cdot T(ABC)\Big) \end{aligned}$$

$$IO \approx \alpha_{13} \cdot |AC| + \alpha_{14} \cdot |B| + \alpha_{15} \cdot \frac{|AC|}{T(AC)} \cdot T(ABC)$$

The CPU cost of the InterJoinPQ is determined by the size of the input lists, the output lists, and the operations on the priority queue. The cost of inserting

into or deleting from the priority queue is logarithmic in the size of the priority queue at the time that the action occurs. This number is hard to estimate using only path statistics. Assuming uniformity of matching `B` nodes with `(A,C)` pairs, the average size of the priority queue PQ_a is:

$$PQ_a = \frac{T(AC)}{max\{|//A//B[.//C]|, 1\}}$$

From this, we propose the following CPU and I/O cost estimates for InterJoinPQ:

$$CPU \approx \alpha_{16} \cdot T(AC) \cdot \log |PQ_a|$$

$$IO \approx \frac{1}{BF} \cdot (\alpha_{17} \cdot T(AC) + \alpha_{18} \cdot T(B))$$

We are currently developing an XML synopsis structure that can efficiently and accurately estimate $T(X_1X_2...X_k)$. With this capability, we can effectively cost each of the algorithms in an XML query optimizer.

The InterJoinRT algorithm is similar to an index nested-loop join in a relational processor with an R-tree index on the inner relation. Estimates for the InterJoinRT costs can be derived using existing R-tree cost models (e.g., [17]). The cost depends on the number of probes in the R-tree, the number of pages visited in an R-tree search, and the number of items in the priority queue.

5 Experimental Studies

We implemented the proposed InterJoin algorithms (including some of the result-buffering alternatives), the stack-based structural join algorithms [9], and the holistic twig join [10] in Java. B-Tree indexes were built for every distinct tag name in the input. All experiments were run using J2RE 1.5 on a 1.5GHz Intel Pentium 4 running Debian Linux 3.0 with KDE 3.3.2.

5.1 Test Set

We used both synthetic and real XML data sets to evaluate the implementations described in the paper. For synthetic data, we used the COMET data generator [18]. COMET is a tool that generates large XML datasets from DTD-style definition files. We generated XML files ranging in size from about 10 Megabytes to about 1 Gigabyte, with recursion levels varying from 0 to 50. Note that these files are actually contain far more elements than a standard XML file of this size. This is because we did not add any attributes, values, etc (since they do not affect query processing on node indexes). Furthermore, we do not add any XML elements that are not relevant to our queries since they do not affect the processing. Thus, a 10 Megabyte file, for example, consists of over one million XML elements that must be processed to answer the queries.

Some real test data was also taken from the University of Washington XML Data Repository [19]. The NASA dataset consists of 23 Megabytes of data, with average node depth of about 5.6, and no recursion. The Protein Sequence Database consists of 683 Megabytes of data, with average node depth of about 5.2, and no recursion. The TreeBank dataset consists of 82 Megabytes of data, with average node depth of 7.9 and significant recursion. The first two datasets are used to evaluate simple non-recursive data of very different sizes. The last dataset provides highly recursive data with considerable nesting that poses greater challenges for efficient query processing.

5.2 Queries

In our experiments, we consider only the structural join parts of a query. For example, suppose we have the following query:

```
//person//info/personal[flags/@private=0]//age[. > 50]
```

We ignore the filtering steps and project the query to the following form

```
//person//info/personal[flags]//age
```

Value predicates, positional predicates, and other filters can be applied independent of the method used to verify structural relationship; thus, we will consider processing only the projected form of each query.

Random queries were generated for each of the synthetic and real data sets using COMET and the resulting queries were projected to include only the relevant parts. This resulted in queries consisting of descendant and child queries with between three and six axis steps. Some sample queries, along with their projections, for the NASA dataset are included in the following table.

Generated Query	**Projected Query**
//ds//ob[//*]//fn	//d//ob//fn
//d/ob/para//fn[//tableLink]	//d/ob/para//fn//tableLink
//fn[//para/fn/ob[//bibcode]]	//fn//para/fn/ob//bibcode

To ensure that the best query plan was chosen in all cases, we enumerated every legal query plan and ran every one. The best plan was recorded and reported in the results. In a real XML processor, a generator would prune and select plans based on cost estimates; however, cost-based pruning of XML query plans is not well studied and we risk penalizing approaches based on poor query plan choices if we do not run all plans. All experiments were run 5 times and the results were based on the average of all the results except the first run.

5.3 InterJoin and Materialized Views on Real XML Data

We conducted some initial experiments on real data sets using randomly generated queries to identify those that can be sped-up by using InterJoin with

or without materialized views. Table 1 shows the number of queries that were processed more quickly by augmenting binary structural joins with the InterJoin operator, and the average speed-up for the improved queries. The right columns show the corresponding improvements when a materialized view is added.

Runtime and I/Os on Real Data									
		Using InterJoin				Materialized Views			
		Time		I/Os		Time		I/Os	
Dataset	Queries	Num. Faster	Avg. Speedup	Num. Faster	Avg. Speedup	Num. Faster	Avg. Speedup	Num. Faster	Avg. Speedup
NASA	10	2	176%	3	169%	5	332%	9	207%
PSD	8	2	106%	2	130%	4	170%	8	183%
TreeBank	16	5	184%	7	186%	14	199%	16	225%

Table 1. Using InterJoin and materialized views to evaluate queries on real XML data.

The biggest speed-ups occur with queries on the NASA and TreeBank datasets. The NASA dataset contains several nodes that appear sporadically in the document. Joining these nodes first eliminates a significant number of temporary results, improving processing time and reducing memory requirements. For the TreeBank dataset, the data is highly recursive and so structural joins of common elements resulted in a large amount of temporary results. In cases where a selective join was performed first, InterJoin provided savings of up to 407%.

Building materialized views on query subpaths gives speedups when the result size of the query subpaths is as much as 150% of the result sizes of each of the axis step matches in the expression. For example, if we construct a view for `//A//B//C` and the result size is at most 1.5 times the size of `|//A| + |//B| + |//C|`, then further queries against this view are almost always more efficient.

Selective Non-Adjacent Node Predicates To simulate predicates relating non-adjacent nodes, we performed a structural join of two non-adjacent nodes in an XPath query and then randomly deleted some of the results before continuing the processing of the query. We used this approach to evaluate the performance of InterJoin on a real data set in the presence of (artificial) predicates. Figure 3 shows the effect of varying the selectivity of two non-adjacent nodes on the query evaluation time for five representative queries.

The processing time for some queries can be reduced to less than 10% of the original query time with highly selective predicates (filtering between 95% and 99% of the matches). This suggests that query processing can be improved dramatically by recognizing highly selective predicates relating two nodes in an XPath query and evaluating these axes first.

5.4 InterJoin and Materialized Views on Synthetic XML Data

Having identified the queries where InterJoin and the tuple-based materialized views provide the greatest reductions in CPU and I/O costs, we generated

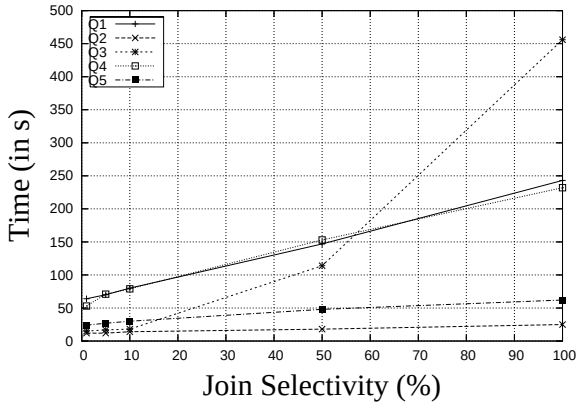

Fig. 3. Examples of query processing times for queries with predicates relating two non-adjacent nodes. The selectivity represents the number of matched tuples that are produced as output.

synthetic data to further explore these cases. The data was generated using a DTD-style schema, similar to the schema S shown below, for paths of length three up to six. By varying the parameters, we can specify approximately how many tuples will be returned by a path query for two non-adjacent nodes in the schema.

```
S:  root -> a.1*[5,15,uniform], a.2*[35,45,uniform],
           a.3*[40,60,uniform], d.1*[40,60,uniform];
    a.1  -> b.1*[5,15,uniform];
    b.1  -> c.1*[3,7,uniform];
    a.2  -> b.2*[400,600,uniform];
    d.1  -> b.3*[35,45,uniform];
    b.3  -> c.2*[5,15,uniform];
    a.3  -> c.3*[5,15,uniform];
```

Figure 4 shows the time required to answer a descendant-only query using the best plan consisting of (1) only binary structural joins, (2) a combination of InterJoin and binary structural joins, (3) the holistic twig join, and (4) InterJoin on a materialized view for `//A//C`. The numbers along the x-axis show the percentage of `(A,C)` tuples relative to the number of `(A,B)` and `(B,C)` tuples. The results are for files of about 70 Megabytes and are representative of all of the results.

A plan that uses the InterJoin operator requires between 30% and 68% of the processing time required if using only structural joins when the `(A,C)` tuple ratio is up to 50%. The holistic twig join consistently outperforms the binary operators except for the case when the selectivity of the `(A,C)` join is very high (99.5%), in which case the runtime for the InterJoin operator is similar. The discrepancy is greater for larger XPath queries as the number of I/Os saved by using the holistic

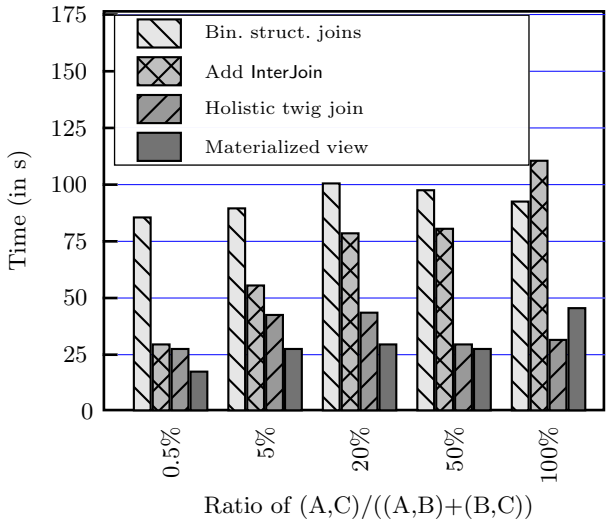

Fig. 4. Performance of InterJoin and materialized views on a query of the form `//A//B//C`. Here, the `AC` matches represent the number of matches between the two non-adjacent nodes of interest. `B` is an XPath subexpression consisting of between one and three steps.

twig join becomes increasingly significant. This can be predicted from the cost models since the CPU and I/O costs for the holistic twig join are linear in the size of the input node lists, while the CPU and I/O costs for the InterJoin operator depend on the number of `(A,C)` tuples. The materialized views outperform all other algorithms when the tuple ratio is 50% or less, requiring only 65% of the processing time of holistic twig join when the tuple ratio is 5%.

Highly Recursive Documents If an XML file is structured such that some nodes are deeply nested, then performing structural joins of adjacent nodes can cause the temporary result sizes to grow very large. For example, a document of the form `<A><B><B>...<B><C><C>...<C><D>` will result in a significant number of `(A,B)`, `(B,C)`, `(C,D)` pairs. If a predicate restricts the `D` nodes to match with a single `A` or `B` node, then the query can be evaluated faster using an InterJoin.

Figures 5 and 6 show the time and memory usage for each of the InterJoin algorithms with increasingly recursive documents. We fixed the number of XML elements at around 20,000 and varied the recursive depth of elements in the document. The runtime scales well with increasing recursive depth, allowing highly recursive documents to be processed efficiently. The memory usage becomes considerably worse for the InterJoinPQ and InterJoinML algorithms since the highly recursive documents require buffering many `(A,C)` pairs during processing. The InterJoinSL algorithm requires only 320k of memory since it retains only one copy of each `C` node.

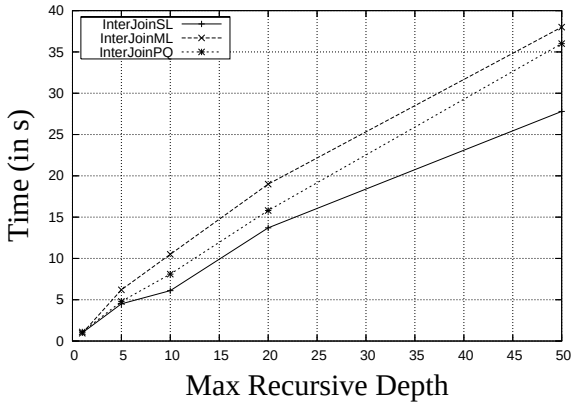

Fig. 5. Impact of recursion on runtime.

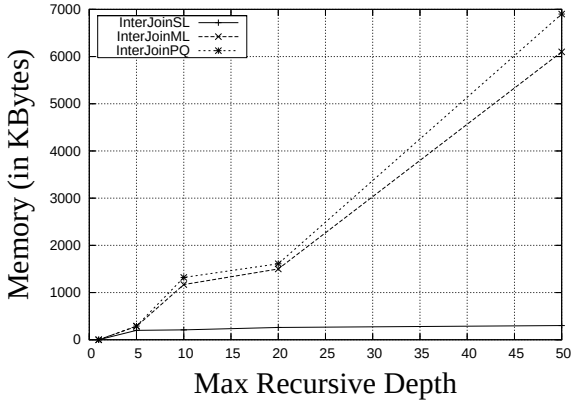

Fig. 6. Impact of recursion on memory.

Large XML Documents To test the scalability of InterJoin, we processed queries over increasingly large XML documents. Figure 7 shows the time required to answer five representative queries on the documents, using binary structural joins and InterJoinSL. We fixed the match ratio for the queries to be 50% to test the effect of the larger file sizes. The cost models for algorithms predict the linear trend shown in the picture.

Complex Interleaving with InterJoin We conducted a few preliminary experiments for answering XPath expression queries by combining the results of interleaving subexpressions. At this point, we have only considered the impact of adding additional interleaving "filters" to check for the required relationships among output nodes.

In order to test the impact of adding additional filters to the InterJoin operator, we conducted experiments on the TreeBank dataset by interleaving the results of some of the queries used in Section 5.3. We evaluated five different

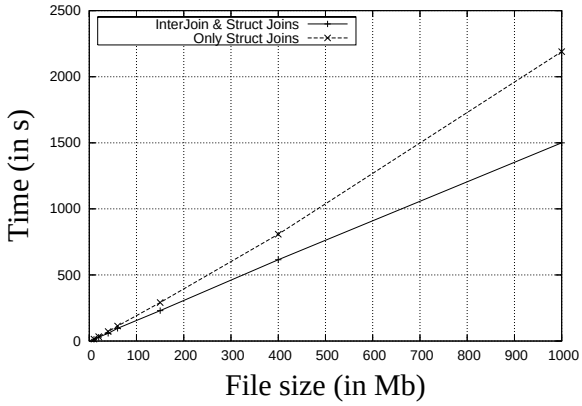

Fig. 7. Scalability of InterJoin.

queries consisting of four axis steps (for example, `//EMPTY//S//NP//PP`) and then interleaved the results. Figure 8 shows the average CPU time required to interleave the results for one up to six interleaving conditions.

When the number of interleaving conditions increase, we expect the number of successful matches to decrease. Despite the extra work involved in filtering the unsuccessful matches according to additional interleaving join conditions, the savings in output construction lead to more efficient processing times. This suggests that the extra work involved when applying additional interleaving filter conditions does not significantly hinder query processing.

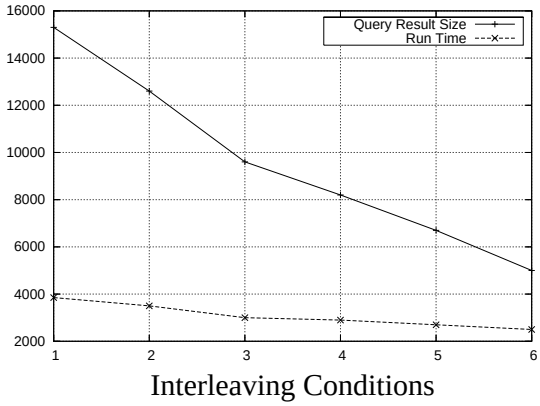

Fig. 8. Impact of interleaving.

InterJoin with R-tree Views We created an R-tree index of the form `//A//C//D//E` and used the R-tree based InterJoin operators to answer a query of the form `//A//B//C//D//E`. First, we set the number of `B` matches to be small

(i.e., about 0.01% of the view size), resulting in few probes in the R-tree. The left-most column of Figure 9 compares the number of I/Os required to answer this query. The results show that the R-tree incurs fewer than 2% of the I/Os required by the best plan using tuple-based views.

Next, we considered a case where there were many probes into the R-tree. We created R-tree indexes on `(A,C)` pairs, for the experiments described in Section 5.4 and then used InterJoinRT to evaluate the query `//A//B//C`. The middle column of Figure 9 shows the number of I/Os required to answer the query. The number of I/Os is almost twice as large since for every `B` node, at least one or two I/Os are used to probe the B-Tree. The structural join and other InterJoin algorithms use main-memory structures and so they perform better in this case.

An advantage of the R-tree is that it does not require that the `B` nodes be in document order to perform an InterJoin. The right-most column of Figure 9 shows the number of I/Os required to answer a query of the form `//A//B//C` when the input list is not ordered. The R-tree view incurs fewer than 2% of the I/Os required by the other approaches since it does not need to sort the `B` nodes.

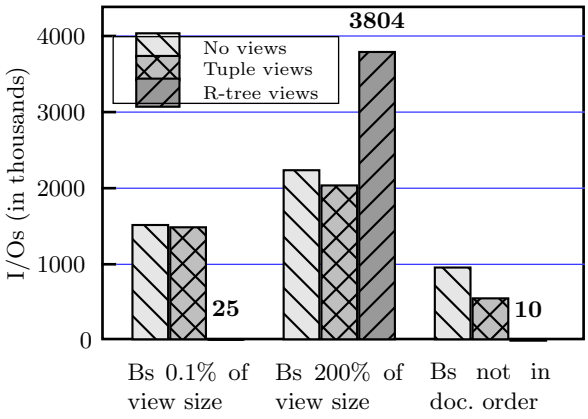

Fig. 9. Performance of R-tree views.

6 Related Work

The XML literature most relevant to our work are the proposals dealing with structural join algorithms and the use of materialized views. Zhang et al. [5] noted that relational database systems were not well suited to direct evaluation of XPath queries. They proposed the MPMGJN to perform structural joins on two node lists. Al-Khalifa et al. [9] gave a stack-based implementation for structural joins that is CPU and I/O optimal for ancestor-descendant queries. Bruno et al. [10] gave a stack-based holistic twig join algorithm that computes entire XPath expression results (for a subset of XPath queries) in a single pass of all the node

list inputs. Several other papers proposed extensions to the structural join and holistic join algorithms to take advantage of indexes on the inputs and to deal with a larger subset of XPath (for example, see [20]).

Recently, proposals have been made for generating and answering queries using materialized views. Some of the proposals are based on identifying and caching frequent XPath queries (such as [21] and [22]) and can be extended to take advantage of the InterJoin operator.

Balmin et al. [16] proposed a framework for deciding view containment and compensation for four materialization options. These views can be composed of references to the output nodes, absolute paths of each node, copies of corresponding subtrees, or the values contained in the nodes. Views composed of references can be used to answer many queries, but require navigation in the document to compute the compensation results. Path-based views can get large for some XML documents, and require pattern matching for many queries.

Our views can be used in conjunction with the views proposed by Balmin et al. [16], allowing for more options for view materialization and query processing. For example, if a view of the form `//A//Z` contains results with long paths, it is preferable to use our method, storing only the information about the matched `A` and `Z` nodes. Our views are also preferable in cases where it is expensive (or impossible) to navigate the XML document.

7 Conclusions and Future Work

We have proposed a materialized view representation that is suitable for structural-join-based XML query processors. The views can be treated as temporary results in an XML query physical plan, making it easy to integrate them into an existing processor. To take advantage of these materialized views, we have proposed InterJoin, a new binary structural join physical operator that allows for joining interleaving fragments of a path expression. Hence, the new operator can exploit a large number of materialized views and temporary results in evaluating path queries.

We propose several efficient implementations of the InterJoin operator. In the presence of an R-tree data structure, we have shown how to build multi-dimensional indexes on temporary results in order to answer multiple queries on this data efficiently. The InterJoin operator results in lower CPU and I/O cost than traditional structural joins, for a large number of queries. Materialized views achieve additional speedups of up to a factor of four. As in the case of relational views, the view construction time is amortized by reusing the views in evaluating several path queries.

We have provided simple cost models for the algorithms based on statistics on the data. The formulas can be used to identify situations where InterJoin is a better alternative to structural joins and situations where a materialized view is favored over the holistic twig join.

We are currently looking at other temporary result formats that would allow materialized views to be used with the holistic twig join. For example, if we

represent the results of a query `//A//B` by removing all tags from the XML document except those that participate in a successful match and annotate these with a interval encoding, these results can now be considered as a *merged queue* input to the holistic twig join. The same approach does not work with tuple-based temporary results because document order is not guaranteed for all of the tags.

References

1. Fernández, M., Siméon, J.: (Galax) `http://www.galaxquery.org/`.
2. Fiebig, T., Helmer, S., Kanne, C.C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Anatomy of a native XML base management system. The VLDB Journal **11** (2002) 292–314
3. DeHaan, D., Toman, D., Consens, M., Özsu, M.: A comprehensive XQuery to SQL translation using dynamic interval encoding. In: SIGMOD. (2003) 623–634
4. Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., Naughton, J.: Relational databases for querying XML documents: Limitations and opportunities. In: VLDB. (1999) 302–314
5. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On supporting containment queries in relational database management systems. In: SIGMOD. (2001) 425–436
6. Diao, Y., Altinel, M., Franklin, M., Zhang, H., Fischer, P.: Path sharing and predicate evaluation for high-performance XML filtering. TODS **28** (2003) 267–516
7. Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. In: ICDE. (2002)
8. Josifovski, V., Fontoura, M., Barta, A.: Querying XML streams. The VLDB Journal **14** (2005) 197–210
9. Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J., Srivastava, D., Wu, Y.: Structural joins: A primitive for efficient XML query pattern matching. In: ICDE. (2002) 141–152
10. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: Optimal XML pattern matching. In: SIGMOD. (2002) 310–321
11. Zhang, N., Agrawal, S., Özsu, M.: BlossomTree: Evaluating XPaths in FLWOR expressions. Technical Report CS-2004-58, University of Waterloo (2004)
12. Jagadish, H., Al-Khalifa, S., Chapman, A., Lakshmanan, L., Nierman, A., Paparizos, S., Patel, J., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: TIMBER: A native XML database. The VLDB Journal **11** (2002) 274–291
13. et al., E.R.: (Rainbow) `http://davis.wpi.edu/~dsrg/rainbow/`.
14. Wu, Y., Patel, J., Jagadish, H.: Structural join order selection for XML query optimization. In: ICDE. (2003) 443–454
15. Grust, T.: Accelerating XPath location steps. In: SIGMOD. (2002) 109–120
16. Balmin, A., Ozcan, F., Beyer, K., Cochrane, R., Pirahesh, H.: A framework for using materialized XPath views in XML query processing. In: SIGMOD. (2004) 60–71
17. Böhm, C.: A cost model for query processing in high dimensional data spaces. TODS **25** (2000) 129–178
18. Zhang, N., Haas, P., Josifovski, V., Lohman, G., Zhang, C.: Statistical learning techniques for costing XML queries. In: VLDB. (2005)

19. Miklau, G.: (UW XML repository) http://www.cs.washington.edu/research/xmldatasets/www/repository.html.
20. Jiang, H., Lu, H., Wang, W.: Efficient processing of XML twig queries with OR-predicates. In: SIGMOD. (2004) 59–70
21. Mandhani, B., Suciu, D.: Query caching and view selection for XML databases. In: VLDB. (2005) 469–480
22. Xu, W., Özsoyoglu, Z.: Rewriting XPath queries using materialized views. In: VLDB. (2005) 121–132