InterJoin: Exploiting Indexes and Materialized Views in XPath Evaluation

Derek Phillips* Ning Zhang Ihab F. Ilyas† M. Tamer Özsu

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

Abstract

XML has become the standard for data exchange for a wide variety of applications, particularly in the scientific community. In order to efficiently process queries on XML representations of scientific data, we require specialized techniques for evaluating XPath expressions. Exploiting materialized views in query processing significantly enhances 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 are evaluated significantly faster and are easily costed as part of a query plan. In general, current structural joins can not 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 *allows for more join alternatives in XML query plans. 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 XML query optimizer. Experiments on real and synthetic XML data show significant speed-ups of up to 200% using* InterJoin*, and speed-ups of up to 400% using our materialized views.*

*This work was supported in part by an NSERC PGS-B scholarship.

†This work was supported in part by NSERC Discovery Grant 311671-05.

1 Introduction

In the last few years, XML has emerged as the primary format for data exchange for a wide variety of fields. The scientific community, in particular, has started to amass large quantities of XML data, making it more important to manage and query the data efficiently. Path queries play a significant role in querying data of this form since they appear ubiquitously in various XML query languages (e.g., XQuery and XSLT). Experiments suggest that efficient processing of XML queries requires specialized physical operators [17]. Several techniques have been proposed for computing the results of path queries in an XML query processor. These include navigational techniques that scan documents to find the desired output nodes (e.g., Y-Filter [6], XTrie [5], and TurboXPath [9]), join-based techniques that apply structural joins to tag indexes (e.g., binary joins [1, 17] and holistic joins [4]), and hybrid techniques that combine navigation and joins (e.g., BlossomTrees [18]).

1.1 Motivation

XPath expressions can be complex, leading to expensive query processing. In many queries, however, there are significant commonalities among the XPath expressions involved. For example, consider the following query that lists the titles of books where the editor is also an author:

```
Q: for $b in doc("bib.xml")//book
   where some $a in $b//author//name,
              $e in $b//editor//name
   satisfies $a = $b
   return <book>{$b/title}</book>
```

As in the relational context, query processing of XPath queries is accelerated by identifying common expressions and evaluating the expressions incrementally. This is done by computing materialized views on the results of some expressions and then rewriting queries to reuse the views. In order to rewrite queries to reuse views, we first need to use *view matching* to determine which views can be used to answer a query, and then we need to determine the *compensation* expression that must be applied to the view to determine the query result [2]. The query processing time depends largely on the time required to evaluate the compensation expression.

With current XPath processing techniques, the efficiency of the compensation processing varies dramatically depending on the queries and available views. For example, when we have a view `V` representing the results of the expression `//A//B`, we can answer the query `Q' = //A//B//C` by navigating from the `B` (or `C`) nodes in the document, or by using a structural join. Experiments have shown that the structural join approach is often considerably faster [1]. On the other hand, when we have a view `V'` representing the results of the expression `//A//C` and we want to answer the same query `Q'`, we have to resort to navigating from each of the `A` (or `C`) nodes. Current structural join proposals require additional processing to handle these cases. Other approaches, including string-matching on the Dewey IDs [14], are difficult to cost and require non-standard processing. We propose a new operator that extends the current structural join operator in a manner that allows it to be used to compute a larger class of compensation expressions. In this paper we do not focus on view matching, however, the reader is referred to [2, 12, 13, 16] for proposals on view matching and XPath evaluation.

We consider the structural join operator as a logical operator that operates on the results of two path queries. Current structural join implementations (such as MPMGJN [17] and the stack-based methods [1]) 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 an axis relating the two expressions in P (e.g., $P = P_1/P_2$), then we use current structural joins. 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 joins can not compute P using the intermediate results of P_1 and P_2.

We propose the InterJoin operator, a novel physical implementation of structural joins that interleaves 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. The selectivity of two non-adjacent subpaths in an XPath expression is high. 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 a selective operation is a structural join that computes `//book//name[.='John Smith']`. We answer this query first and then use InterJoin to join the results with `author` nodes.
2. Part of a query is expensive and so it will be reused for other queries. In this case, we evaluate the subquery and materialize the results as a view. For example, in the query `Q` given earlier, it may be more efficient to compute the results for `//book//name` and use this to evaluate `//book//author//name` and `//book//editor//name`.

In order to use the InterJoin operator with materialized views, we need to extend current XML view proposals since they focus primarily 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 information about the `A` node(s) with which they matched. We propose storing the interval-encoding of all the nodes matched in an XPath expression. This allows us to compute efficient 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 [7] and TurboXPath [9]. Thus, a materialized view is a valid input to existing operators, and the output of these operators can be materialized as a view.

Note that we are *not* introducing bushy query plans for XPath evaluation (see [15]). Instead, the InterJoin operator increases the number of joinable XPath expressions.

1.2 Contributions

The main contributions of the paper are as follows:

- We propose a new definition for views that are easily integrated into a join-based XML processor to boost query performance.
- We propose efficient algorithms for a novel InterJoin physical operator. The 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 to estimate the CPU and I/O costs, allowing the InterJoin operator to be integrated into a cost-based optimizer.
- We provide experimental results on two scientific data sets (i.e., NASA and protein sequence data) and other synthetic data that demonstrate the improvements obtained using the InterJoin operator in a query plan, as well as the speed-ups that are achieved with our proposed views using appropriate algorithms and data structures.

The paper is organized as follows. Section 2 gives the necessary background. 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 operator in Section 5. Related work and conclusions are given in Sections 6 and 7.

2 Preliminaries

We consider path queries on XML databases consisting of forests of rooted, ordered, labeled trees. We make the common assumption that there is an interval-based representation of XML nodes that allows constant-time checks for ancestral or parental relationships.

Structural Joins A number of techniques have been proposed for structural joins (e.g., [1, 17]). A structural join operator outputs all node tuples that satisfy the relationship specified by a single XPath (descendant or child) axis.

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 properly 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, `(A,B)` matches correspond to the tuples output by evaluating `//A//B` (or `//A/B`). In most cases, the tuples will contain intervals for several matched nodes, however, InterJoin depends only on two intervals from the "left" input and one from the "right" (see Figure 1). To simplify the presentation, we will often refer to the `(A,C)` pairs from the left input, which represent the two intervals of interest in the input tuple. Similarly, we will refer to the `B` nodes from the right input.

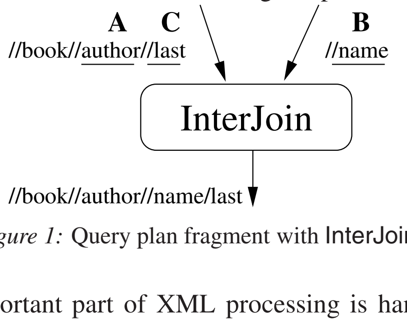

Figure 1: Query plan fragment with InterJoin.

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 [2]. 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 is easily integrated in a structural-join based processor.

3.1 Temporary Results as Materialized Views

We propose a class of views that store tuples of match encodings (i.e., intervals) corresponding to the format of the temporary results in a query processor. In the simplest case, we have a query such as $X_1 : X_2 : \dots : X_k$, where each : is some XPath axis. The materialized view will consist of $2k$ entries for each match (i.e., the open and close index for each X_i tag). We will use X_i`.open` and X_i`.close` to refer to these values. A unified format for materialized views and temporary results allows the query optimizer to seamlessly incorporate materialized views in a query plan.

For example, after evaluating the query `//book[@price>40]//author/name`, we use InterJoin and other basic operators to answer:

```
//book[@price>100]//author/name,
/mall/store/book[@price>40]//author/name
//book[@price>40]/authors/author/name
```

Given this representation, it is easy to check that an `(A,C)` pair matches with a `B` node; there is a match if the following three conditions hold: (1) $A.open < B.open$, (2) $B.open < C.open$, and (3) $C.open < B.close$.

When a view represents a query containing branching or predicates, we include all the nodes in the view. For example, for the query `//author//name[first]/last` there would be eight entries in the tuple (two for each tag). The materialized view (or temporary result) maintains information mapping items in the XPath expression to tuple entries. The case is similar for predicates, where we keep entries for the relevant nodes so that more selective predicates can be applied later.

3.2 InterJoin: Physical Structural Join Operators

We have implemented the InterJoin operator in a variety of ways. Each implementation has its own set of preconditions on the inputs and its own set of options for the output. The most general implementations require only that the inputs are ordered appropriately; the first general algorithm, InterJoinML, requires that the left input be ordered by `[A,C]` and the right input be ordered by `[B]`.

Algorithm 1 InterJoin with multiple temporary lists.

```
INTERJOINML(M1:Iterator, A,C:int, M2:Iterator, B:int)
Initialize empty stack $S$
▷ Loop until no more matches are possible
while ($M1.hasNext()$ or $S.size() > 0$) and $M2.hasNext()$
  do ▷ Get the open tag location of the next
     ▷ unprocessed (A,C) match or B node
     $m \leftarrow \min(M1.peek()[A].open, M2.peek()[B].open)$
     ▷ Pop (A,C) pairs that can no longer match
     for each $s \in S$
       do while $s.list[0].open < m$
            do $s.list.removeFirst()$
     if $M1.peek()[A].open < M2.peek()[B].open$
          ▷ If the next item is an A tag, add it to the
          ▷ stack with a list of all its (A,C) matches
          $s \leftarrow$ new stack node
          $s.list.append(M1.next())$
          while $M1.peek()[A].open = s.list[0][A].open$
            do $s.list.append(M1.next())$
          $S.push(s)$
     else
          ▷ Otherwise, try to match the B node
          ▷ with (A,C) pairs from the stack
          $b \leftarrow M2.next()$
          for each $s \in S$
            do for $i = 0$ to $s.list.size() - 1$
                 do until $s[i][C].open > b[B].close$
                      OutputMatch ($s[i]$, $b$)
```

Algorithm 2 InterJoin with a priority queue.

```
INTERJOINPQ(M1:Iterator, A,C:int, M2:Iterator, B:int)
Initialize empty priority queue $PQ$
▷ Loop until no more matches are possible
while ($M1.hasNext()$ or $PQ.size() > 0$) and $M2.hasNext()$
  do $m \leftarrow \min(M1.peek()[A].open, M2.peek()[B].open)$
     ▷ Remove items from PQ that can no longer match
     while $PQ_{top}[C].open < m$
       do $PQ.removeTop()$
     if $M1.peek()[A].open < M2.peek()[B].open$
     ▷ If it is an (A,C) pair, add it to PQ
          $PQ.add(\ M1.next()\ )$
     else
          ▷ Match the B node with (A,C) pairs from PQ
          $pq \leftarrow PQ_{top}$; $b \leftarrow M2.next()$
          while $pq[C].open < b[B].close$
            do OutputMatch ($pq$, $b$)
                 $pq \leftarrow PQ.next()$
```

InterJoinML: InterJoin using Multiple Lists This algorithm scans the inputs, keeping a stack of every `A` node that can still match a `B`, along with all of its corresponding `(A,C)` matches. The `B` nodes are scanned, outputting any successful matches. The output matches are ordered by `[B,A]`, however, using a buffering technique similar to that employed by Al-Khalifa et al. [1], we can also output the matches ordered by `[A,B]`. The pseudocode is given in Algorithm 1.

In the case that the left input is only guaranteed to be ordered by `[A]`, with the right input also ordered, we use InterJoinPQ, a priority-queue based implementation. In addition to having fewer constraints on the input, the InterJoinPQ algorithm provides additional output orderings. However, the InterJoinML algorithm is guaranteed to be linear time, while the InterJoinPQ algorithm is not (see Section 4).

InterJoinPQ: InterJoin using a Priority-Queue This implementation scans the list of `(A,C)` matches keeping a priority queue of matches keyed on `C.open`. We try to match `B` nodes against the priority queue nodes, knowing that when we reach a `C.open` value whose document order is after `B.close`, then no further items in the priority queue can match. The output matches will be ordered by `[B,C]` and using buffering we can output matches ordered by `[C,B]`. The pseudocode is given in Algorithm 2.

3.3 Index-Based InterJoin

Index-based InterJoin exploits 2-D indexes available on parts of the XPath expression (e.g., R-Tree indexes). In this scenario, no input order is required and large performance gains are achieved.

InterJoinRT: Index-InterJoin Operator (R-Trees)

We implement the InterJoin operator as a 2-D range query involving points and a rectangle. Given a point `(A.open,C.open)`, we define 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 (see Figure 2. Similarly, given a `B` node, a point `(A.open,C.open)` will match if it is contained in the rectangle `(0, B.close)`, `(B.open,B.close)`. The R-Tree structure (and the InterJoinRT algorithms) deals with higher dimensional data by projecting data into the dimensions of interest.

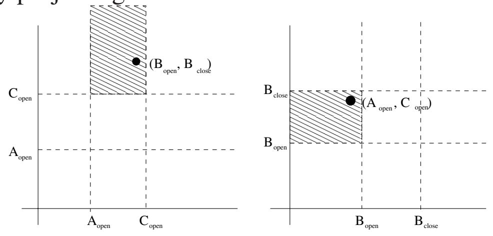

Figure 2: InterJoin conditions for the R-tree algorithms in 2-D.

We build an R-tree on points representing items from

Algorithm 3 R-tree search for B matches.

```
RTREESEARCH(RT:RTree, M:Match, B:int)
 1  $PQ \leftarrow$ empty priority queue
 2  Set extent for rectangle $R$ to
      $(0, M[B].close)$, $(M[B].open, M[B].close)$
 3  $PQ.add(RT.root)$
 4  ▷ Get next point from the priority queue
 5  while $PQ.size() > 0$
 6    do if $PQ_{top}$ is a leaf
 7             Output match of $M$ with $PQ_{top}$
 8       else
 9             ▷ Any R-tree nodes that could contain matching
10             ▷ points are added to the priority queue
11             $temp \leftarrow PQ.removeTop()$
12             for each child $ch$ of $temp$
13               do if $ch$ intersects $R$
14                    $PQ.add(ch)$
```

Algorithm 4 InterJoin with R-Tree on AC nodes.

```
INTERJOINRT(M1:Iterator, A,C:int, M2:Iterator, B:int)
1  $RT \leftarrow$ empty R-tree
2  ▷ Add a point for every (A,C) pair
3  while $M1.hasNext()$
4    do $m \leftarrow M1.next()$
5         Add point $(m[A].open, m[C].open)$ to $RT$
6  ▷ Probe with a rectangle for each B
7  while $M2.hasNext()$
8    do $m \leftarrow M2.next()$
9         $RTreeSearch(RT, m, \{B\})$
```

one input list and then probe the R-tree using rectangles built from the other input list. We enforce order on the results by modifying the R-tree search algorithm to use a priority queue. Algorithm 3 describes the new R-tree search algorithm and Algorithm 4 presents InterJoinRT, using `(A,C)` pairs in the R-tree. An analogous version that builds the R-Tree on `(B.open,B.close)` is omitted. With the search algorithm, we can output results ordered by `[B,A]`, `[B,C]`, and `[A,B]`, regardless of the input orders.

3.4 A Special Case for InterJoin

In the presence of schema information, or with sufficient analysis of the data, we obtain properties of the inputs that help to speed-up processing. For example, for non-recursive `B` nodes (i.e., no `B` node is a descendant of another `B` node), we propose the InterJoinNR algorithm. This algorithm requires that the left input be ordered by `[C]` and that the `B` nodes be ordered. It is the fastest and most space-efficient implementation we present.

InterJoinNR: InterJoin over Non-recursive Documents
Given a list of non-recursive `B` nodes, every `(A,C)` pair can match with at most one `B` node. We perform an InterJoin by keeping a single `B` node buffered and looping through all `(A,C)` pair inputs. The output will be ordered by `[C,B]`. 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 the algorithm works regardless of whether or not the `C` nodes are recursive.

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 use one of the other implementations.

3.5 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, given a materialized view for `//V//Z` and another for `//W/X/Y`, we use these views to answer the query `//V//W/X/Y/Z`. This is more difficult because we need to check for the `V//W` relationship and the `Y/Z` relationship simultaneously. Similarly, given a materialized view for `//H//J//L` and another for `//I//K`, we use these views to answer the query `//H//I//J//K//L`.

Given an R-tree implementation, we build an R-tree using the intervals for the `H` and `J` intervals, and build a second R-tree with the `J` and `L` intervals. Now, we take the `(I,K)` matches and probe with the `I` interval into the first R-tree and with the `K` interval into the second R-tree. The successful matches are joined to obtain the result matches.

To answer queries like the first one, there are several possibilities for evaluating the query using InterJoin. For example, we can 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`.

In a case where R-trees are not available, the second example is evaluated 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` interval is contained in the `J` interval and contains the `L` interval of a potential match. This situation can be generalized to any level of interleaving, however every interleaving increases the number of filters.

The InterJoin operator also allows predicates to be included in a query. For example, suppose we have a view on `//person//age[.>50]` which consists of tuples of the form `(person,age)` for all people aged over 50.

Implementation	Non-recursive Bs	Index Available	Input Orders	Output Orders	Efficiency
NR (Non-recursive)	Required	Not required	[C], [B]	[C,B]	Fastest
ML (Multiple lists)	Not required	Not required	[A,C], [B]	[B,A], [A,B]	Fast
PQ (Priority queue)	Not required	Not required	[A], [B]	[B,C], [C,B]	Reasonable
RT (R-Tree)	Not required	Required	Not required	[B,A], [B,C], [A,B]	Variable

Table 1: Comparison of InterJoin algorithms.

This view can be used to answer queries adding additional structural or value predicates.

Efficiently handling complex queries is still very much an open problem. There are many other examples of complex queries where it is beneficial to use the InterJoin operator. In this case, the applicability of the InterJoin operator rests mainly with the ability to answer the related view coverage problem.

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 InterJoinML algorithm maintains a stack of A nodes and each has a list of matches. Let $|AC|$ and $|B|$ be the number of tuples from the left and right inputs, respectively. All the operations take constant time, giving a runtime of $O(|AC| + |B|)$.

The InterJoinPQ algorithm adds (and removes) each match at most once to (from) 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 usage for InterJoinPQ and InterJoinML is $O(| AC | + h \cdot C_{dist})$ since we may need to buffer all (A,C) matches (as in the worst-case example shown above).

Each of the algorithms is easily 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. [1], we have shown that the asymptotic runtime and space for the algorithms remains the same.

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 InterJoin operator based on path statistics. Statistics maintained by the database are 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 [9].

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

In addition to the update operations, there are lookup operations to the heads of the lists (lines 4-5) that do not contribute to updates. The number of lookup operations is determined by the size of the stack (i.e., the recursion depth of the A nodes) and the number of A and B elements at that recursion depth.

Define $T(X_1 X_2 ... X_k)$ as the number of matches satisfying $//X_1//X_2//...//X_k$. If A is non-recursive, then $T(AC) = |//A//C|$; if not, then we 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 As that appear in a root-to-leaf path) and $(//A)^i$ denotes i concatenations of the string //A.

The number of lookup operations is estimated using the proportion of the input size $|AC|$ to $T(AC)$. The output size $|ABC|$ is 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:

$$\begin{aligned} CPU &\approx \alpha_1|//A| + \alpha_2|AC| + \alpha_3|B| + \alpha_4\frac{|AC|T(ABC)}{T(AC)} \\ IO &\approx \alpha_5|AC| + \alpha_6|B| + \alpha_7\frac{|AC|}{T(AC)}T(ABC) \end{aligned}$$

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

The CPU cost of the InterJoinPQ is determined by the size of the input, output, and 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:

$$CPU \approx \alpha_8 T(AC) \log |PQ_a|$$
$$IO \approx \alpha_9 T(AC) + \alpha_{10} T(B)$$

We are currently developing an XML synopsis structure that efficiently and accurately estimates $T(X_1 X_2 ... X_k)$. 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., [3]). 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 [1], and the holistic twig join [4] 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.

5.1 Test Set

We used both synthetic and real XML data sets to evaluate the proposed algorithms. For synthetic data, we used the COMET data generator [19], which generates large XML datasets from DTD-style definition files. We generated XML files ranging in size from about 10 Megs to 1 Gig, with recursion levels varying from 0 to 50. Real test data was taken from the UW XML Data Repository [11]. The NASA and PSD sets consist of 23 and 683 Megs of non-recursive data, respectively. The TreeBank dataset consists of 82 Megs of data with significant recursion.

5.2 Queries

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

```
//person//info/all[flags/@p=0]//age[.>50]
```

We drop the predicates and project the query to:

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

Value predicates, positional predicates, and other filters are 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. 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 Data

We conducted some initial experiments on real data sets using randomly generated queries to identify those that are evaluated more quickly using InterJoin with or without materialized views. Table 2 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 improvements when a materialized view is added.

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%.

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.

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 2: Using InterJoin and materialized views to evaluate queries on real XML data.

Figure 3 shows the effect of varying the selectivity of two non-adjacent nodes on the query evaluation time for five representative queries.

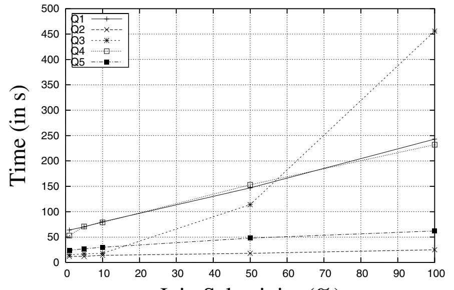

Figure 3: Predicates relating two nodes.

The processing time for some queries is 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 is 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 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 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 specify approximately how many tuples will be returned by a path query for two non-adjacent nodes in the schema.

```
r   -> 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`.

When the `(A,C)` tuple ratio is up to 50%, a plan that uses the InterJoin operator requires between 30% and 68% of the processing time required when using only structural joins. 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

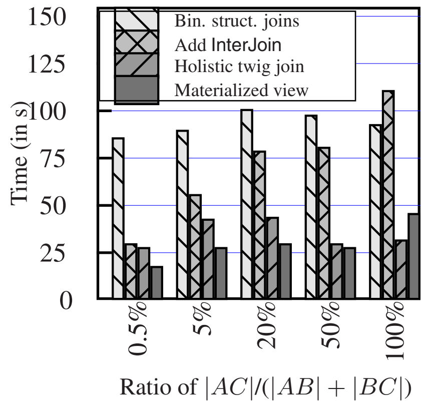

Figure 4: Synthetic Data Tests.

discrepancy is greater for larger XPath queries as the number of I/Os saved by using the holistic 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 is evaluated faster using an InterJoin.

Figure 5 shows the processing time for the two general 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.

Large XML Documents To test the scalability of InterJoin, we processed queries over increasingly large XML documents. Figure 6 shows the time required to answer five representative queries on the documents, using binary

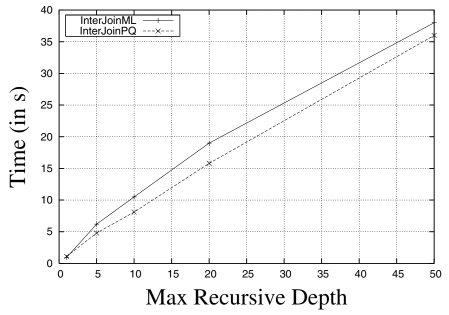

Figure 5: Impact of recursion on runtime.

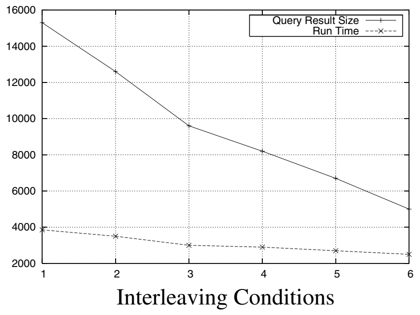

Figure 7: Impact of interleaving.

structural joins and InterJoin. 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.

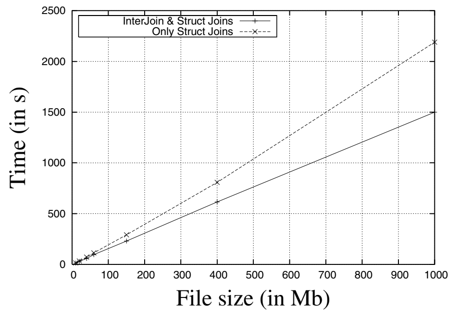

Figure 6: Scalability of InterJoin.

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 queries consisting of four axis steps (for example, `//EMPTY//S//NP//PP`) and then interleaved the results. Figure 7 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.

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 8 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 8 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.

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 8 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.

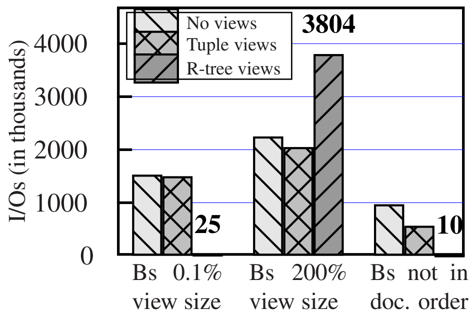

Figure 8: Performance of R-tree views.

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. [17] 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. [1] gave a stack-based implementation for structural joins that is CPU and I/O optimal for ancestor-descendant queries. Bruno et al. [4] 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 and to handle a larger subset of XPath (see [8]).

Recently, proposals have been made for generating and answering queries using materialized views. Some of the proposals involve caching frequent XPath queries (e.g, [10] and [16]) and can be extended to use the InterJoin operator.

Balmin et al. [2] proposed a framework for deciding view containment and compensation for four materialization options. These views can be composed of references to the output nodes, paths to each node, copies of the subtrees, or 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 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. [2], allowing for more options for view materialization and query processing. For example, given a view of the form `//A//Z` that 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 are 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 exploits 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 to evaluate several queries.

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

References

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[3] C. Böhm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

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

[5] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.

[6] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 28(4), 2003.

[7] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *VLDB Journal*, 11, 2002.

[8] H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with OR-predicates. In *SIGMOD*, 2004.

[9] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *VLDB Journal*, 14(2), 2005.

[10] B. Mandhani and D. Suciu. Query caching and view selection for XML. In *VLDB*, 2005.

[11] G. Miklau. UW XML repository. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[12] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[13] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. Candan. Incremental maintenance of path-expression views. In *SIGMOD*, 2005.

[14] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[15] Y. Wu, J. Patel, and H. Jagadish. Structural join order selection for XML query optimization. In *ICDE*, 2003.

[16] W. Xu and Z. Özsoyoglu. Rewriting queries using materialized XPath views. In *VLDB*, 2005.

[17] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

[18] N. Zhang, S. Agrawal, and M. Özsu. BlossomTree: Evaluating XPaths in FLWOR expressions. In *Proc. 21th Int. Conf. on Data Engineering*, 2005.

[19] N. Zhang, P. Haas, V. Josifovski, G. Lohman, and C. Zhang. Statistical learning techniques for costing XML queries. In *VLDB*, 2005.