Optimizing Cursor Movement in Holistic Twig Joins

Marcus Fontoura, Vanja Josifovski, Eugene Shekita, and Beverly Yang
IBM Almaden Research Center
San Jose, CA, USA
trevi@almaden.ibm.com

ABSTRACT

Holistic twig join algorithms represent the state of the art for evaluating path expressions in XML queries. Using inverted indexes on XML elements, holistic twig joins move a set of index cursors in a coordinated way to quickly find structural matches. Because each cursor move can trigger I/O, the performance of a holistic twig join is largely determined by how many cursor moves it makes, yet, surprisingly, existing join algorithms have not been optimized along these lines. In this paper, we describe *TwigOptimal*, a new holistic twig join algorithm with optimal cursor movement. We sketch the proof of *TwigOptimal*'s optimality, and describe how *TwigOptimal* can use information in the return clause of XQuery to boost its performance. Finally, experimental results are presented, showing *TwigOptimal*'s superiority over existing holistic twig join algorithms.

Categories and Subject Descriptors: H.3[Information Systems]: Information Storage and Retrieval; E.1[Data]: Data Structures

General Terms: Algorithms, Performance, Experimentation

Keywords: XML, Twig Joins, Indexing, Evaluation

1. INTRODUCTION

Both XPath and XQuery allow users to specify value and structural constraints in XML queries using path expressions. A path expression can be represented as a *query tree*, which is structurally matched against XML data. Performing this structural matching as efficiently as possible is one of the key issues in building an XML query engine.

There are two common approaches to perform structural matching efficiently. One approach is to use a *structural join* [1, 5, 19], where a query tree is decomposed into a set of binary ancestor-descendant or parent-child relationships. The relationships are then evaluated using a binary merge join. Another approach is to use a *holistic twig join* [3, 10], which processes a query tree with a single n-ary join.

Holistic twig joins represent the state of the art for evaluating path expressions in XML queries. Not only do they perform better [10], but they are self-tuning and do not require a query optimizer. Holistic twig joins are *index-based*, typically relying on an *inverted index* for positional information about XML elements. *Cursors* are used to access the inverted index and moved in a coordinated way to efficiently find structural matches.

Several variations on holistic twig joins have been proposed in the literature [3, 4, 9, 10, 16, 18]. Because each cursor move in an inverted index can trigger I/O, the performance of a twig join[1] is largely determined by how many cursor moves it makes. Despite this observation, existing twig join algorithms have not been optimized along these lines. There has been more focus on minimizing the memory requirements of intermediate results than on minimizing the number of cursors moves. The problem with existing twig join algorithms is that they make a local, and hence sub-optimal, decision in choosing which cursor to move next and how far to move it.

In this paper, we describe *TwigOptimal*, a new holistic twig join algorithm with optimal cursor movement. This is accomplished by looking more globally at the query's state to determine which cursor to move next. We provide a sketch of *TwigOptimal*'s optimality, and present experimental results, showing its superiority over existing twig join algorithms.

Another shortcoming of existing twig join algorithms is that they assume all nodes in a query tree need to be output. However, an XQuery return clause often requires only a subset of the nodes being matched to be output. We refer to the nodes that need to be output in the return clause as *extraction points*. By being aware of extraction points, we show how *TwigOptimal* can dramatically reduce the number of cursor moves it makes. This is accomplished by skipping over nodes that do not need to be output.

In summary, the main contributions of this paper are:

- A new holistic twig join algorithm called *TwigOptimal* that handles *and-or* branches and outperforms existing algorithms by optimizing its cursor movement.
- A description of how XQuery extraction points can be used to further improve *TwigOptimal*'s performance.
- Experimental results showing *TwigOptimal*'s superiority over existing holistic twig join algorithms.

[1]Note that, for readability, we will often use "twig join" as a shorthand for "holistic twig join".

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

2. RELATED WORK

Structural joins [1, 5, 19] were the first method proposed for evaluating path expressions. In a structural join, complex path expressions are broken down into a set of ancestor-descendant or parent-child relationships and then evaluated using binary merge joins.

Holistic twig joins were first described in [3], which showed how performance could be improved by considering all nodes in the query tree holistically. Optimizations to the original twig join algorithm were later described in [10] and then extended to handle OR predicates in [9]. Several papers have also investigated new index types to speed up twig joins [5, 8, 10, 12, 18]). *TwigOptimal* provides the same functionality offered by existing twig joins, including OR predicates. Although a standard inverted index is assumed in this paper, *TwigOptimal* can also be used with many of the new index types that have been proposed to speed up twig joins.

Reference [10] is the only other paper that really addresses the problem of choosing which index cursor to move next in a holistic twig join. In [10], when the cursor positions do not form a structural match, a "broken edge" in the query tree is chosen using various heuristics and then "fixed". Fixing a broken edge consists of repeatedly moving the two cursors forming the edge until they structurally match the query. The problem with this approach is that, once a broken edge is chosen, only the two cursors forming the edge can be moved. In contrast, using the concept of virtual cursor moves, *TwigOptimal* is able to consider the whole query's cursor state every time a cursor needs to be moved.

To evaluate a path expression like `/x/y/z`, a holistic twig join will typically open three index cursors, one for each path step. However, if the position of data nodes are encoded with a Dewey value [15], the cursors for x and y can be derived from the cursor for z. In [18], these derived cursors were called *virtual cursors*. The choice of names is perhaps unfortunate, but the virtual cursors described in [18] should not be confused with the virtual cursor moves described in this paper. The former describes a derived cursor, whereas the latter describes a virtual *move* of a physical cursor.

References [2] and [11] described methods for minimizing the memory requirements of intermediate results when processing XQuery fragments over XML streams. Similarly, [3, 10] described methods for minimizing the memory requirements of intermediate results in twig joins. However, the performance of a twig join is largely determined by how many cursor moves it makes. Therefore, the focus in this paper is on minimizing the number of cursor moves in a twig join rather than on its memory requirements.

3. BACKGROUND

In this section, we establish some background that will be used in the remainder of the paper.

3.1 Query Trees

As in the previous work on holistic twig joins [3, 10], we will focus on XPath expressions or XQuery fragments that can be represented by a single *query tree* and structurally matched against XML data in one pass over an index. These are basically single-document path expressions containing child ('/') axis, descendant ('//') axis, and equality predicates, all of which can be combined using Boolean *AND* and *OR* operators.

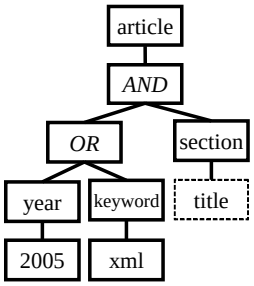

Figure 1: Example query tree

As an example, consider the following XQuery fragment, which returns the titles of XML articles in 2005:

```
for $a in //article[year = "2005" or
                    keyword = "xml"]
  for $s in $a//section
    return $s/title;
```

This query has three XPath expressions, one in each of the two *for* clauses, and one in the *return* clause. Figure 1 shows the resulting query tree. Each XPath step in Figure 1 is represented by a *path node*. There are also *AND* and *OR branching nodes*, which specify that all or at least one of the subpatterns below the node must be matched, respectively. Finally, dotted lines are used to indicate that a path node corresponds to an *extraction point*, i.e., it is returned by the query. In general a query tree can have more than one extraction point since the *for-let-where* block of XQuery returns tuples of bindings.

3.2 Inverted Indexes

XML data is also commonly represented as a tree, with nodes corresponding to text values, elements, or attributes, and edges capturing the nesting of elements. The *position* of XML data nodes are used by holistic twig joins to do structural matching. The position of an XML data node can be encoded in different ways. One alternative is to use the well-known BEL encoding, where a node's *begin, end, level* forms its position. Another alternative is the *Dewey* encoding [15]. In this paper, we assume BEL encoding, but the techniques we describe are orthogonal to the choice of the encoding.

Holistic twig joins have been implemented over several indexing structures [8, 13, 16]. Here, we assume a standard inverted index [7]. Inverted indexes have stood the test of time and are frequently used in information retrieval and XML systems alike [1, 3, 5]). Briefly, an inverted index consists of one *posting list* per distinct token in the dataset, where a token can be a text value, attribute, or element tag. A posting list contains one *posting* for each occurrence of its token in the dataset and is sorted by position (in our case using BEL). Stepping through the posting list for a token T will enumerate the positions of every occurrence of T in order, by document and then within document. We assume each posting list is itself indexed, typically with a B-tree, so that searching for a particular position within a posting list is efficient.

3.3 Query Evaluation

To find documents that structurally match a query tree Q, a holistic twig join associates each path node q with the posting list in the inverted index whose token matches q.

A *cursor* is opened for each of these posting lists, and then moved in a coordinated way to find documents that structurally match Q. Parent-child and ancestor-descendant constraints imposed by the query are checked by looking at the current positions of cursors. The output of a holistic twig join is a stream of tuples, where each tuple corresponds to a *solution*, that is, a set of data nodes that structurally match the path nodes in Q. In existing twig join algorithms, all possible solutions are output [3, 10].

4. THE TWIGOPTIMAL ALGORITHM

In this section we describe *TwigOptimal*. For simplicity and to make the comparison to existing algorithms clearer, *TwigOptimal* is initially described without taking extraction points into account. Extraction points are addressed in Section 7.

4.1 Data Structures

As in existing holistic twig join algorithms [3, 10], the evaluation state of *TwigOptimal* is a triplet $<Q, C, S>$, where Q is the query tree being evaluated, C is the set of cursors for accessing the inverted index, and S is a set of stacks for constructing solutions. Data nodes that are part of a solution are stacked on S and output when a full solution is found.

An index cursor Cq and a stack Sq are associated with each path node q in Q. Cq points to the current posting for q, while Sq is used to remember the data nodes for q that are part of a solution. Each stack entry also has a pointer to an entry in an ancestor stack, which is used by *TwigOptimal* to output a solution, much like in [3, 10, 11].

The position of a cursor Cq is accessed via $Cq.begin$, $Cq.end$, and $Cq.level$, and similarly for a stacked node in Sq. Parent-child or ancestor-descendant constraints are checked by looking at BEL values. We say a cursor Cp *contains* another cursor Cq iff $Cp.begin \leq Cq.begin$ and $Cp.end \geq Cq.end$. Similarly, we say that a stack Sp contains Cq if there is some entry in Sp that contains Cq.

For each cursor Cq, the method $Cq.forwardTo(pos)$ moves Cq forward from its current position to the first position greater than or equal to pos. This can trigger I/O as Cq physically seeks to pos. To optimize its cursor movements, *TwigOptimal* also uses *virtual* cursor moves that, unlike physical cursor moves, do not trigger I/O. Instead, a virtual move on Cq simply sets $Cq.begin$ without physically moving Cq. This will be made clearer shortly. $Cq.virtual$ is set to *true* whenever Cq is virtually moved and reset to *false* whenever Cq is physically moved.

Unlike path nodes, each branching node in Q is not associated with a posting list in the inverted index. However, each branching node does have a cursor, which is used to pass along the position of its parent or a child cursor when *TwigOptimal* is deciding which cursor to move next. Consequently, the cursor of a branching node is always virtual. By maintaining a cursor for each branching node, *TwigOptimal* does not need to distinguish between path nodes and branching nodes in most cases, which in turn simplifies the algorithm.

4.2 The Main Loop

ExecuteQuery(), which is shown in Figure 2, forms the entry point and main loop of *TwigOptimal*. It initializes each cursor to their first posting (line 1), then it repeatedly inspects the path node q corresponding to the *min cursor* (line 3), that is, the cursor with the smallest *begin* value, until is has found and output all solutions.

To find and output solutions, the cursors are moved until

```
ExecuteQuery()
  1. initialize all cursors and stacks;
  2. while (not done) {
  3.    q = the path node in Q
           associated with the min cursor;
  4.    while (Extension(q) == false) {
  5.       MoveCursors(q);
  6.       q = the path node in Q
              associated with the min cursor;
  7.    }
  8.    OutputAndPush(q);
  9.    Cq.forwardTo(Cq.begin + 1);
 10. }
```

Figure 2: The main loop

an *extension* [10] for q is found (lines 4–7). An extension for q is basically a partial solution rooted at q. When an extension for q has been found, the cursor positions in the subtree rooted at q are guaranteed to be a part of a solution. In addition, the stacks of q's ancestors contain the position of data nodes that, when combined with the extension for q, forms a full solution.

Once an extension has been found, *OutputAndPush()* is called (line 8) to output any new stacked solutions and push Cq onto its stack. As in [10], Cq is only stacked when it is part of a solution. Finally, Cq is advanced to its next physical location (line 9) to start the search for another solution. The main loop terminates when the end of one or more posting lists is reached, allowing the algorithm to conclude that no more solutions can be found.

4.3 Checking an Extension

Extension(), which is shown in Figure 3, checks whether the cursor positions in the subtree rooted at q form an extension. {C} is the set path of cursors in the subtree rooted at q. For the subtree rooted at q to form an extension, Cq must be contained by its parent's stack Sp, all the cursors in {C} must be real (not virtual), and all the cursors in {C} must recursively satisfy the containment constraints of Q (line 3). In the latter case, a cursor with an AND under it needs to contain all its children cursors, while a cursor with and OR under it needs to contain at least one of its children cursors.

```
Extension(q)
  1. p = the parent of q;
  2. {C} = the set of descendant
           path cursors of Cq in Q;
  3. if (Sp contains Cq and
         all the cursors in {C} are real and
         {C} satisfies Q's containment constraints) {
  4.    return true;
  5. }
  6. else {
  7.    return false;
  8. }
```

Figure 3: Checking an extension

4.4 Moving the Cursors

TwigOptimal calls *MoveCursors(q)* in its main loop as it searches for an extension, where q corresponds to the min cursor. *MoveCursors()* basically tries to move the cursors in the subtree rooted at q to the next extension for q, if any. To avoid I/O, this is done using virtual cursor moves. A physical cursor move is made only when further virtual progress becomes impossible.

MoveCursors() is shown in Figure 4. Two passes over the subtree rooted at q are made to virtually move the cursors, a bottom-up pass (line 6) and a top-down pass (line

7). The two passes over the subtree globally discover the furthest each cursor can be moved forward without missing an extension for q. This is in contrast to existing twig join algorithms, where cursors are moved in a localized way, only looking at the positions of one parent-child pair of cursors at a time.

```
MoveCursors(q)
  1. p = the parent of q;
  2. if (Sp does not contain Cq) {
  3.    Cq.begin = max(Cq.begin, Cp.begin + 1);
  4.    Cq.virtual = true;
  5. }
  6. MoveCursorsBottomUp(q);
  7. MoveCursorsTopDown(q);
  8. if (q still corresponds to the min cursor) {
  9.    Cb = the best virtual cursor to physically
            move among Cq and its descendents;
 10.    Cb.forwardTo(Cb.begin);
 11.    Cb.virtual = false;
 12. }
```

Figure 4: Moving the cursors

MoveCursors() begins by checking whether Cq is contained by its parent stack (line 2). If not, then Cq is virtually moved to the max of $Cq.begin$ or $Cp.begin + 1$, which is the most Cq can be moved forward without missing an extension for q. Next, the cursors are virtually moved bottom-up and then top-down (lines 6–7), as described earlier. Finally, a check is made to see whether q still corresponds to the min cursor (line 8). If so, then further virtual progress is impossible, at which point a physical cursor move is made (lines 9–11). Note that, on exit, *MoveCursors()* may or may not have actually found an extension for q. This is checked in the main loop of *TwigOptimal*.

When *MoveCursors()* is forced to make a physical cursor move, it picks the "best" virtual cursor in the subtree rooted at q to move (line 9). This is essentially the cursor that is predicted to move the furthest. How to determine the best cursor to move is beyond the scope of this papers. For various heuristics see [10].

MoveCursorsBottomUp(), which is shown in Figure 5 recursively performs a bottom-up pass over the subtree under consideration. The goal of this pass is to try and move each parent cursor forward so it contains its children cursors. If q is an AND node (line 4), then Cq must contain the cursor of its max child in order for there to be an extension. Similarly, if q is an OR node (line 8), then Cq must contain the cursor of its min child. Finally, if q is a path node with a child (line 12), then Cq must contain that child's cursor.

```
MoveCusorsBottomUp(q)
  1. for (each child c of q) {
  2.    MoveCusorsBottomUp(c);
  3. }
  4. if (q is an AND node) {
  5.    m = the child of q with the max cursor;
  6.    Cq.begin =  Cm.begin;
  7. }
  8. else if (q is an OR node) {
  9.    m = the child of q with the min cursor;
 10.    Cq.begin = Cm.begin;
 11. }
 12. else if (q has a child) {
 13.    c = the only child of q;
 14.    if (Cq.end < Cc.begin) {
 15.       Cq.begin = max(Cq.begin, Cq.end + 1);
 16.       Cq.virtual = true;
 17.    }
 18. }
```

Figure 5: The bottom-up pass to move cursors

Recall that each branching node's cursor is used to pass along the position of its parent cursor or a child cursor. Here, in the case of an AND node, Cq is used to pass up the *begin* value of its max child cursor (lines 5–6). Similar action is taken in the case of an OR node with its min child cursor (lines 9–10). Finally, if q is a path node with a child, and $Cq.end$ falls before that child's cursor, then Cq is virtually moved to the max of $Cq.begin$ or $Cq.end + 1$ (lines 15–16), which is the most Cq can be moved forward without missing an extension. The max is needed (line 15) to deal with the case where a previous call to *MoveCursors* has already virtually moved $Cq.begin$ past $Cq.end$.

After *MoveCursorsBottomUp()* finishes, each cursor will have been virtually moved as far forward as its children cursors will allow it to be moved without missing an extension. *MoveCursorsTopDown()*, which is shown in Figure 6, is then called to recursively perform the top-down pass over the subtree under consideration. The goal of this pass is to try and move each child cursor forward so it is contained by its parent cursor.

```
MoveCusorsTopDown(q)
  1. for (each c in children of q) {
  2.    if (c is an AND or an OR node) {
  3.       Cc.begin = Cq.begin;
  4.    }
  5.    else if (Cc.begin < Cq.begin and
                  Cc is not contained by Sq) {
  6.       Cc.begin = Cq.begin + 1;
  7.       Cc.virtual = true;
  8.    }
  9.    MoveCursorsTopDown(c);
 10. }
```

Figure 6: The top-down pass to move cursors

In *MoveCursorsTopDown()*, c and q correspond to the current child and parent nodes being examined, respectively. If c is a branching node, then the child cursor Cc is used to pass down the position of the parent cursor Cq (line 3). Else, if $Cc.begin$ falls before Cq and is not contained by its parent stack Sq, then Cc is virtually moved to $Cq.begin + 1$ (lines 5–8), which is the most Cc can be moved forward without missing an extension. Sq needs to be checked for containment in this case to guard against missing solutions when there is recursive data for q.

4.5 Outputting a Solution

Solutions are output and pushed onto the stacks in *OutputAndPush()*, which is shown in Figure 7. Before the cursor of node q is stacked (line 8), a check is made to see if q corresponds to the root of Q. If so, then one or more solutions are output before the new root cursor is stacked (lines 2–6).

```
OutputAndPush(q)
  1. if (q == Q.root) {
  2.    while (Sq.top() is not an ancestor of Cq) {
  3.       output solutions with Sq.top();
  4.       Sq.pop();
  5.       remove nodes from all stacks lacking a root;
  6.    }
  7. }
  8. Sq.push(Cq);
```

Figure 7: Outputting a solution

The simple stacking method use here is not optimized for space, since that is not the focus of this paper. However, note that *OutputAndPush()* can easily be changed to use the stacking method described in [10], which is optimized along those lines.

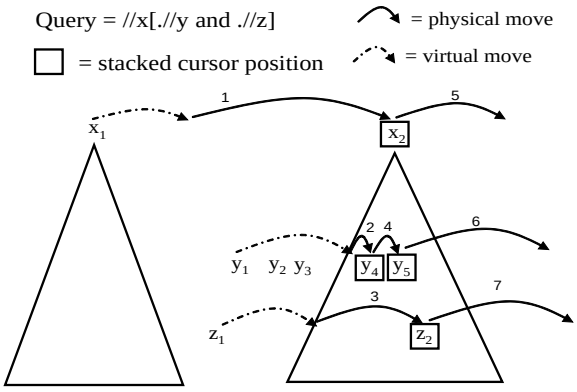

Figure 8: Cursor movement example

4.6 Cursor Movement Example

We now illustrate *TwigOptimal*'s cursor movement using the query shown in Figure 8. In this example, there are three cursors on posting lists, one per query tree node. The scope of each x element is represented by a triangle enclosing its descendant y and z elements. For clarity, the figure does not show a complete XML document, but rather only two sibling x elements. The left-to-right position of the letters in the figure indicates the relative position of the data nodes in the document. Physical cursor moves are depicted by solid curved lines, labeled with the sequence number in which they are performed, while virtual moves are depicted by dashed lines.

At the beginning, the three cursors are positioned on x_1, y_1, and z_1, which do not form an extension. Cx is the min cursor and there is no extension for it, so *MoveCursors(x)* is called. Since x_1 ends before y_1 and z_1, Cx is virtually moved just past the end of x_1. At that point, Cx is still the min cursor, so it is physically moved to x_2, after which Cz becomes the min cursor. In the subsequent calls to *MoveCursors()*, Cz and Cy are virtually moved within x_2, causing postings y_2 and y_3 to be skipped. Then Cy is physically moved to y_4. At that point Cx is the min cursor again, but the call to *Extension(x)* returns false since Cz is virtual. Cz is then physically moved to z_2, Cx remains the min cursor, and the next call to *Extension(x)* succeeds.

The call to *OutputAndPush(x)* stacks x_2 and then Cx is physically moved to the next x (not shown). At that point, Cy is the min cursor and, since *Extension(y)* returns true, y_4 is stacked and physically moved to y_5. The same steps are repeated for y_5 since it is also part of an extension. Cy is then physically moved to the next y (not shown). This ensures that all the y nodes within x_2 have been exhausted. Finally, Cz becomes the min cursor, *Extension(z)* returns true, and z_2 stacked. Since Cz is still the min cursor, it is moved again, this time beyond x_2. At that point, there are four stack entries that make up two solutions. These will be output after the next x node is stacked.

5. COMPARISON TO EXISTING ALGORITHMS

Figure 9 and Figure 10 are used to illustrate how *TwigOptimal* can generate far fewer physical cursor moves than existing twig join algorithms. Both figures trace the evaluation of the query `//w//x//y//z` on two sibling w elements. Figure 9 illustrates how the cursors would be moved using the twig join algorithm described in [10], which represents the

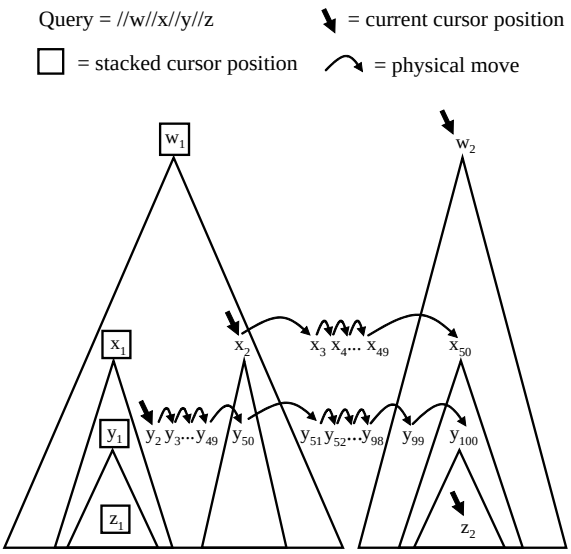

Figure 9: Counting physical cursor moves with existing algorithms

state of the art among existing algorithms, while Figure 10 illustrates how the cursors would be moved in *TwigOptimal*.

As shown, both algorithms start out in the same state, with the first solution w_1, x_1, y_1 and z_1 stacked, and with Cw, Cx, Cy, Cz positioned on w_2, x_2, y_2 and z_2, respectively. We compare the number of physical cursor moves needed for each algorithm to reach the second solution, which is w_2, x_{50}, y_{100} and z_2.

In [10], when the test for an extension on y returns false, a "broken edge" in the subtree rooted at y is identified. A broken edge is a parent-child pair, where the parent cursor does not contain its child cursor. The parent and child cursors for the broken edge are then repeatedly moved in a coordinated way until the parent cursor contains the child cursor. As shown in Figure 9, this causes Cx and Cy to be moved between $x_2 - x_{50}$ and $y_1 - y_{100}$, respectively, resulting in roughly 150 physical cursor moves to find the second solution.

In contrast, as shown in Figure 10, *TwigOptimal* performs only two physical cursor moves. Three virtual moves place Cx and Cy within w_2, and then two physical cursor moves are made to find the second solution.

6. OPTIMALITY PROOF SKETCH

In this section, we provide a sketch of *TwigOptimal*'s optimality. Our main claim is stated in Theorem 1.

THEOREM 1. *Given an evaluation state <Q, C, S>, whenever a cursor is physically moved in TwigOptimal:*

1. *It is necessary to move one of the cursors considered by TwigOptimal.*
2. *The cursor that is physically moved is moved as far forward as possible without missing a solution.*

For the first part of Theorem 1, we examine the two places in *TwigOptimal* where *forwardTo()* is called to physically move a cursor. In *ExecuteQuery()*, a cursor Cq is physically moved after q has been found to be a part of an extension (and thus, a solution). This physical move is necessary to

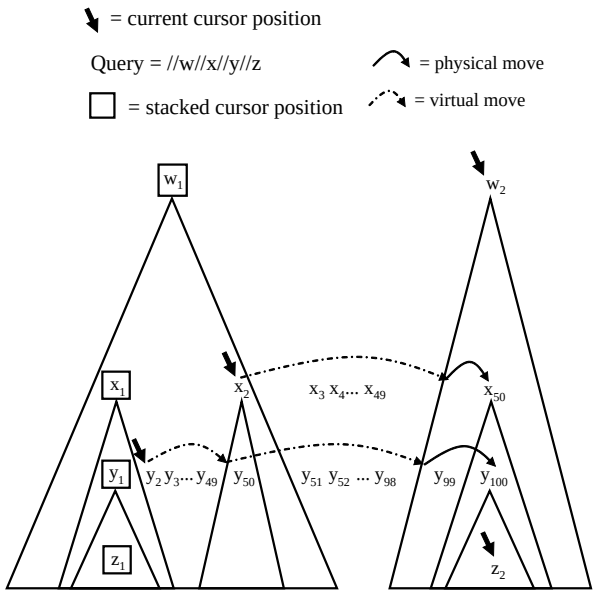

Figure 10: Counting physical cursor moves with *TwigOptimal*

check if Cq's next position is part of a solution. For example, in Figure 8, Cy has to be physically moved until it exits from the x_2's subtree. If any other cursor is moved at that point, the solution x_2, y_5 and z_2 would be missed.

The second call to *forwardTo()* is performed in *MoveCursors()* when the min cursor is virtual. In that case, the two passes over the query tree have virtually moved cursors in the subtree of the min cursor. If some of the cursors are virtual, *TwigOptimal* cannot confirm that it has found an extension. Therefore, *TwigOptimal* has to physically move one of the virtual cursors to find out if it has found a solution. This is illustrated in Figure 8 when Cz is moved to check if there is a z posting within x_2.

For the second part of Theorem 1, it can be shown by induction that the two-pass algorithm used in *MoveCursors()* passes along restrictions on the maximum possible location of the next physical cursor move from descendants to ancestors (in the bottom-up pass) and then from ancestors to descendants (in the top-down pass). Therefore, after *MoveCursors()* has been called, the cursors will have been moved as far forward as possible without missing a solution.

Note that if more than one virtual cursor is available to be physically moved, *TwigOptimal* chooses the "best" cursor using heuristics based on the available statistics [10]. This means that *TwigOptimal* is not instance optimal [6], as instance optimality can only be achieved if there was a way to always choose the best cursor to move. However, *MoveCursors()* does guarantee that when a cursor is physically moved, it is moved as far forward as possible without missing a solution.

Also note that Theorem 1 assumes forward-only cursor moves on a standard inverted index. Unfortunately, this means that when an ancestor cursor is moved to contain a descendant cursor in *MoveCursorsBottomUp()*, *TwigOptimal* can behave inefficiently. Special indexes and the cursor method *forwardToAncestor()* have been proposed to deal with this situation [8, 13]. Instead of using a special index, we propose an implementation of *forwardToAncestor()* that simply moves the ancestor cursor forward to the descendant cursor, then backs up the ancestor cursor to the first position before the descendant cursor. In the (rare) case of recursion, the backward step may have to be repeated. If *forwardTo()* is modified to detect when to call *forwardToAncestor()* under the covers, then nothing in *TwigOptimal* has to be changed to take advantage of this optimization.

THEOREM 2. *Assuming forward-only cursor moves, TwigOptimal performs the minimum number of physical cursor moves to evaluate Q.*

The proof follows from Theorem 1, where we conclude that every physical cursor move performed by *TwigOptimal* is necessary and goes as far forward as possible without missing a solution.

7. EXTRACTION POINTS

Existing twig join algorithms assume that all path nodes in a query tree need to be output. However, an XQuery return clause often requires only a subset of the path nodes being matched to be output. For example, in Figure 1, only the *title* node needs to be output. By being aware of these *extraction points* and "skipping" over data nodes that do not need to be output, the number of cursor moves performed by a twig join can be dramatically reduced. This section describes how extraction points can be used to improve *TwigOptimal*'s performance by making only minor changes to *ExecuteQuery()*, as shown in Figure 11. Lines 9a–9j in Figure 11 simply replace line 9 in Figure 2.

```
9a. if (q is not an extraction point and
        no descendent of q is an extraction point) {
9b.    p = parent of q;
9c.    virtually forward q and all its
       descendant path cursors to Cp.begin + 1;
9d. }
9e. else {
9f.    Cq.forwardTo(Cq.begin + 1);
9g.    if (no descendent of q is an extraction point) {
9h.       virtually forward q's descendant
          path cursors to Cq.begin + 1;
9i.    }
9j. }
```

Figure 11: Changes to *ExecuteQuery()* for extraction points

To understand the code in Figure 11, let q be a path node in the query tree with parent p and consider the four possibilities for q and its descendants:

1. Neither q nor any of its descendants in the query tree are extraction points.
2. Node q is an extraction point but none of its descendants are extraction points.
3. Both q and some descendant of q are extraction points.
4. Node q is not an extraction point but some descendant of q is an extraction point.

Case 1) is handled by lines 9a–9d. In this case, it should be clear that *TwigOptimal* does not need to find all the extensions rooted at q. Consequently, once an extension for q has been found, Cq and all its descendant cursors can be virtually moved within Cp (line 9c).

Case 2) is handled by lines 9e–9i. In this case, it should be clear that *TwigOptimal* does not need to find all the extensions below q. Consequently, after physically moving Cq (line 9f), all its descendant cursors can be virtually moved to within Cq (line 9h).

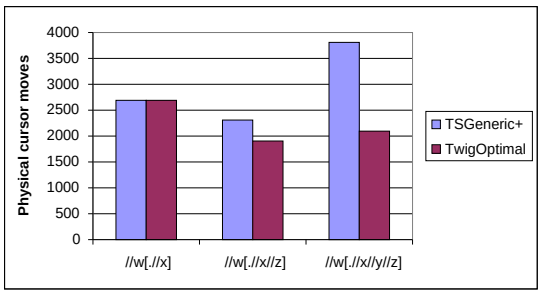

Figure 12: Physical cursor moves for the synthetic dataset

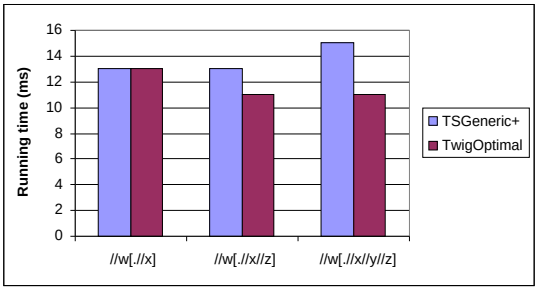

Figure 13: Running time for the synthetic dataset

Finally, case 3) and case 4) fall through the first part of the if-statement and are handled the same as in the version of *ExecuteQuery()* without extraction points, that is, just line 9f is executed. Also, although it is not shown, *OutputAndPush()* changes with extraction points. In particular, if neither q nor any of its descendants are extraction points, then Cq should not be stacked.

8. EXPERIMENTAL RESULTS

In this section, we present experimental results comparing *TwigOptimal* with *TSGeneric+* [9, 10], which represents the state of the art among existing twig join algorithms, Results are also provided to show how *TwigOptimal*'s performance can be improved by being aware of extraction points.

Our experimental testbed was built on the Berkeley DB [14] embedded database, where we implemented both *TwigOptimal* and *TSGeneric+*. All experiments were run on a Linux Red Hat 8.0 workstation, with a 2.2 GHz Intel Pentium 4 processor and 2 GB of main memory. Although, the amount of main memory was large, both twig join algorithms only access their posting lists in the forward direction (with skips). Consequently, our running times are representative of any memory buffer large enough to hold at least the internal pages of the B-tree and one leaf page per posting list.

For data, we used the XMark [17] dataset and a synthetic dataset [18]. For both datasets, the difference in the relative performance of *TwigOptimal* and *TSGeneric+* was similar.

8.1 Results without Extraction Points

In all our experiments, *TwigOptimal* and *TSGeneric+* both used the same top-down strategy [10] for picking which cursor to physically move next. As noted earlier, if both algorithms use the same strategy for picking which cursor to move next, *TwigOptimal* should never generate more physical cursor moves than *TSGeneric+*. Moreover, by looking more globally at the query's evaluation state to determine which cursor to move next, *TwigOptimal* should be able to generate fewer physical cursor moves in many cases.

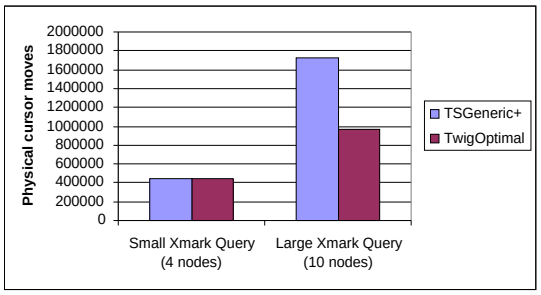

Figure 14: Physical cursor moves for the XMark dataset

Figure 12 shows the number of physical cursor moves for three queries on the synthetic dataset, while Figure 13 shows their running time using cold buffers. As shown, both algorithms performed exactly the same on the first query. This is because the query tree in that case only had two levels (i.e., two cursors), providing little room for *TwigOptimal* to optimize its cursor movement. However, as the queries got more complex and the number of levels in their query tree grew, *TwigOptimal* generated 40% fewer physical cursor moves than *TSGeneric+* and its running time improved proportionally.

For the XMark dataset, we studied two queries, a small one with a 4-node query tree, and a large one with a 10-node query tree:

- *//item//description["science" and "paper"]*
- *//item//description//parlist//listitem//text ["science" and "logotype" and "benefit" and "paper" and "windsor"]*

Figure 14 shows the number of physical cursor moves for these queries. As with the synthetic dataset, on the smaller query, both algorithms generated roughly the same number of cursor moves, while for the larger query, *TwigOptimal* generated 45% fewer cursor moves. *TwigOptimal*'s running time also improved proportionally, but because of space limitations, the graph with those results has been omitted.

8.2 Results with Extraction Points

Figure 15 and Figure 16 show how *TwigOptimal*'s performance can be improved by being aware of extraction points. The graphs in those figures were generated using the synthetic dataset. Moving from left to right in the graphs, the number of extraction points in the query decreases from all 4 nodes to 1 node. The left-most query represents *TwigOptimal* without any optimization for extraction points.

As expected, the number of physical cursor moves *TwigOptimal* made decreased as the number of extraction points in the query decreased. The fact that they decreased somewhat linearly is an artifact of the dataset and the way it was generated with uniform probabilities. In general, the impact of extraction points on performance can vary dramatically, depending on the query and the dataset. The improvement in performance can be substantial anytime there is a pattern like `a[.//b]`, where there are a large number of b nodes under each a node. To illustrate this, another data point was taken from the XMark dataset. On that dataset, the query `//item//text` with 2 extraction points generated

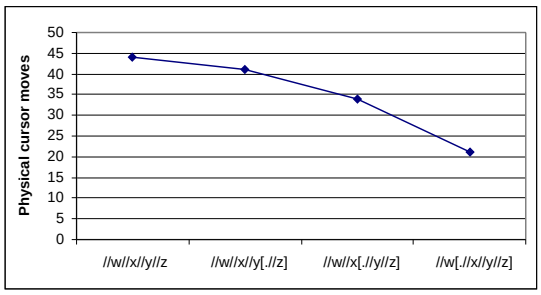

Figure 15: Physical cursor moves for queries with different number of extraction points

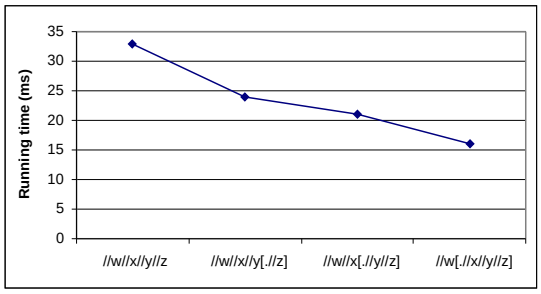

Figure 16: Running time for queries with different number of extraction points

835,740 cursor moves, whereas the same query with 1 extraction point `//item[.//text]` generated 435,000 cursor moves, roughly half as many.

9. CONCLUSION

Because each cursor move can trigger I/O, the performance of a holistic twig join is largely determined by how many cursor moves it makes. Yet, existing twig join algorithms have not been optimized along these lines. Instead, they make a local, and hence sub-optimal, decision in choosing which cursor to move next and how far to move it. In this paper, we described *TwigOptimal*, a new holistic twig join algorithm that minimizes the number of cursor moves it makes by looking more globally at the query's state to determine which cursor to move next. Experimental results were provided, showing that *TwigOptimal* can outperform existing twig joins algorithms by up to 40%.

Another shortcoming of existing twig join algorithms is that they assume all nodes in a query tree need to be output. However, an XQuery return clause often requires only a subset of the nodes being matched to be output. By being aware of these so-called extraction points, we described how *TwigOptimal* is able to further reduce the number of cursors moves it makes. Experimental results were provided, showing that this can provide a substantial improvement in performance.

10. 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] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over xml streams. In *PODS*, 2005.

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

[4] T. Chen, J Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD*, 2005.

[5] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, 2002.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[7] G.Salton and M. J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[8] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. Xr-tree: Indexing xml data for efficient structural join. In *ICDE*, 2003.

[9] H. Jiang, W. Wang, and H. Lu. Efficient processing of xml twig queries with or-predicates. In *SIGMOD*, 2004.

[10] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed xml documents. In *VLDB*, 2003.

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

[12] R. Kaushik, R. Krishnamurthy, J. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, 2002.

[13] Q. Li and B. Moon. Indexing and querying xml data for reguar path expressions. In *VLDB*, 2001.

[14] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Summer Usenix Technical Conf.*, 1999.

[15] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational dbms. In *SIGMOD*, 2002.

[16] H. Wang, S. Park, W. Fan, and P. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *SIGMOD*, 2003.

[17] Xmark: The xml benchmark project. http://monetdb.cwi.nl/xml/index.html.

[18] B. Yang, M. Fontoura, E. Shekita, S. Rajagopalan, and K. S. Beyer. Virtual cursors for xml joins. In *CIKM*, 2004.

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