XML Query Optimization Using the Binding Hash Method

Michael J. Brant Karen C. Davis
Electrical & Computer Engineering Department
University of Cincinnati
Cincinnati, OH USA 45221-0030
mikebr@gmail.com karen.davis@uc.edu

Abstract

XML query processing involves searching an XML document and returning results that satisfy a query while choosing the best method for the particular query and document. In this paper we develop a method called Binding Hash (BH) that performs a subset of XML query operations focusing on structural join. The BH method answers Xpath expressions directly or generates witness trees that can be used as intermediate results in a query processor. It uses pointer-based traversal and index-based access simultaneously to speed-up structural joins. A performance study indicates that the BH method performs better than existing competitive techniques for queries that are deeply nested and non-bushy. The BH method can be integrated into a larger system to be selected for an execution plan when it achieves the best performance.

1. Introduction

XML is a semi-structured markup language that allows the exchange and storage of information across the web [4]. Typically the volume of data in an XML file is too large to be human readable, therefore query processing needs to be automated. The most important operations include structural joins and value-based selection. Finding element tags within a given element and combining these element tags is performed by the structural join operation. Value-based selection involves matching text values found inside element tags. Because the structural join operation has the most overhead, we focus on this operation in our research.

We introduce a new method called Binding Hash (BH) that focuses on processing Xpath expressions. The Binding Hash method has four general steps:

1. Load the XML tree into an XML indexing structure.
2. Parse an Xpath expression into a tree structure.
3. Perform structural joins and value-based selections that result in retrieving XML data that comprises part of the final answer.
4. Combine XML data found in the previous step to output the query answer.

Section 2 describes the concepts and foundations of the BH method. Section 3 describes the major steps of the BH method including data structures and algorithms. Section 4 gives results of a preliminary performance study investigating scalability of the BH method compared with similar competitive techniques. Section 5 summarizes our contributions, discusses our conclusions, and defines future research issues.

2. Binding Hash Foundation

The most common representation for an XML database is an ordered tree. Each node corresponds to either an element or an attribute value while the edges correspond to the element tag nesting relationship. The BH method uses a slightly modified ordered tree model to represent the XML data. The main difference is that an element and its text data are contained within one node, while in the ordered tree model the two are contained within separate nodes. This was done for ease of implementation (so all nodes are the same type), but the algorithm could be modified to support mixed content (i.e., multiple text data items.)

Graphically, attributes are differentiated by the hexagon shape from element tags which are shown by an oval. The numbers next to the nodes represent a unique identifier for the node. An example is given in Figure 1.

Our supported query language is a subset of Xpath that allows us to test structural join and value-based selection operations using the BH method. The Xpath language contains an extensive number of keywords and features [1, 21] that are not supported here because they are outside the scope of testing our structural join algorithm. The features we support are path expressions and filters that allow the matching of attributes and XMLelement tags. Filters are sub-queries that are evaluated as Boolean expressions.

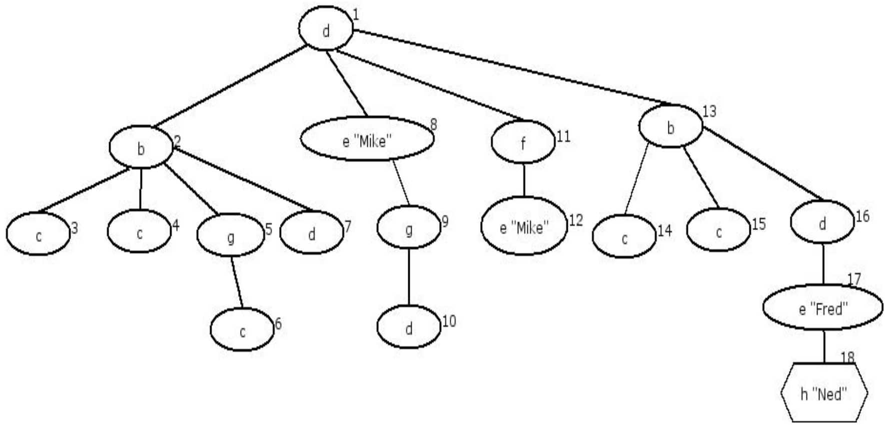

Figure 1: Sample Ordered Tree

Our approach to XML query optimization consists of the following: loading an XML document using an index, parsing the query tree into a logical plan (query tree), performing logical optimization by re-arranging the logical plan, then converting the logical plan into a physical plan. Lore [16] and the Timber XML framework [7] are two XML DBMSs that address these issues. The Timber DBMS framework [7] is based on the Tree Algebra for XML (TAX) [8]. A more recent system, Saxon [10], is an XML engine that has the ability to process Xquery [22], XSLT [23], and Xpath [21] expressions. Saxon offers competitive performance and we use it, as well as Timber, to investigate scalability with some preliminary experimentation in Section 4.

Our work focuses on indexing used for logical and physical optimization. The BH method inputs a pattern tree that supports a subset of the TAX algebra along with an XML document and outputs a set of witness trees. It provides an alternative implementation strategy than that of Timber. In this paper, we define and illustrate the approach and begin to investigate the impact of the BH Method. The novelty of the approach is that we decompose the complex structural join operation into well known simple operations such as tuple intersection (implemented with a hash table here) and pre/post traversal tree operations.

We formally define a TAX pattern tree and witness tree below. A *pattern tree* contains nodes that are variables having constraints that sets of XML nodes have to satisfy. Once XML nodes are found that satisfy all of the constraints then a witness tree is generated.

Definition 1: A *pattern tree* is a pair P=(T, F), where T=(V, E) is a node-labeled and edge-labeled tree such that [8]:

1. Each node V has a distinct integer as its identifier.
2. Each edge is either a soft edge (ancestor-descendant) or a hard edge (parent-child).
3. F is a formula, a boolean combination of predicates applicable to nodes.

In order to demonstrate the BH method we introduce a restricted subset of a TAX pattern tree called the BH pattern tree that matches the BH Xpath subset.

Definition 2: The *BH pattern tree* is a pattern tree with the following restrictions:

1. Each constraint must have an attribute on the left hand side and a string constant on the right side.
2. Only the string matching equality operator is supported.
3. For a node each attribute is restricted to only have a single constraint.

Because BH pattern trees are subsets of TAX pattern trees, we simplify our visual representation of pattern trees. Each oval denotes a pattern node to match. The text inside an oval denotes the tag name and text value to match. The text outside of the brackets denotes the tag name to match while inside the brackets is the content to match proceeded by a comparison operator. Each number just outside the oval uniquely identifies the query node. An edge with "*" represents the ancestor-descendant relationship (*soft edge*) while an edge without a "*" denotes the more specific parent-child relationship (*hard edge*) between two nodes. An example appears in Figure 2.

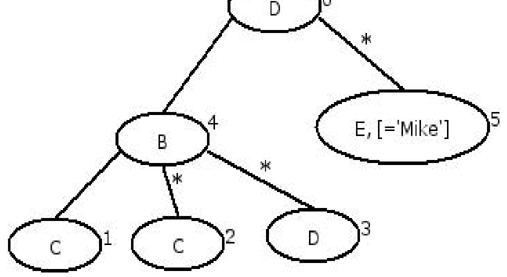

Figure 2: BH Pattern Tree

A pattern node is a variable with constraints that an XML node needs to satisfy. The set of XML nodes that satisfy the pattern tree node constraints are called *node bindings*. We now extend this definition to an *edge binding*. An XML *edge binding* contains two XML nodes whose label, text value, and structural relationship satisfy the constraints specified by a pattern tree edge. A pattern tree generates a collection of data trees that are structurally identical to each other called *witness trees*. Every witness tree contains XML nodes, also known as witness nodes, that map to a pattern tree node.

Definition 3: Let *C* be a collection of data trees, *P=(T, F)* a pattern tree. An embedding of a pattern *P* into a collection *C* induces a *witness tree*; it is a total mapping *h*: *P*→*C* from the nodes of *T* such that [8]:

1. *h* preserves the structure of *T* whenever (*u,v*) is a parent-child edge in *T, h(v)* is a child of *h(u)* (and similarly for an ancestor-descendent edge).
2. The image under the mapping *h* satisfies the formula *F*.

Witness trees are important for optimization because they are intermediate data in processing nested tree algebra operators. Examples of witness trees appear in Figure 11.

The BH method frequently utilizes a hybrid collection type, called a *sorted list*, that has features of both dynamic arrays and hash tables; it allows access to an element in constant time from a collection by either providing a key or an index.

3. Binding Hash Method

Figure 3 summarizes the BH method's data structures, actions and data flow in 7 steps. The inputs are an XML document and Xpath expression; the outputs are the XML nodes that satisfy the expression and an XML file of witness trees. In Step 1, the XML document is loaded into memory using a depth first scan of the tree, creating an XML ordered tree with the following attributes: a unique identifier based on the depth first order of the node, the name of the element/attribute, the text data pertaining to the node, the node depth, an indicator of whether this node is an element or attribute, an array of child pointers, and a pointer to the parent of this node. Figure 1 shows a sample ordered tree that is used to illustrate the BH method. In Step 2, a clustered index is created to access a set of nodes based on labels, node tree depth, and text values. The entry point to the index is a single sorted list whose keys contain label names. Each element in this entry level list points to other sorted lists that have hash keys based on tree depth. Then each of those lists has values that point to other sorted lists whose keys are based on text values. Finally the text-based sorted lists have entries that point to arrays of XML nodes. In practice, any index that returns the same data items could be used, such as one that uses more efficient memory management techniques.

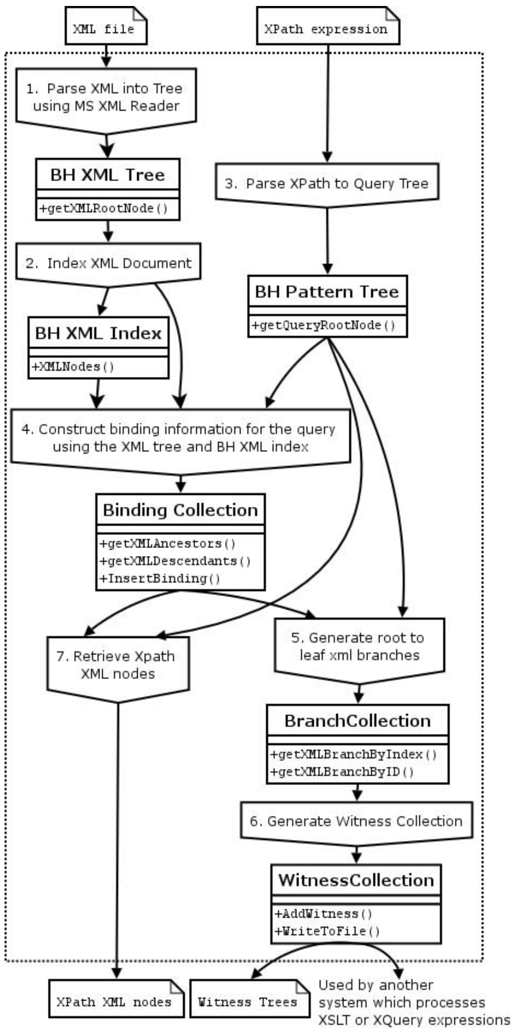

Figure 3: BH Method System Architecture

In Step 3, an Xpath expression is loaded into the BH pattern tree. The BH pattern tree consists of pattern nodes that retain all of the attributes for an XML node described above; however, two additional attributes are stored: a boolean value indicating whether a path from the root to the node has a soft edge, and an array containing edge type information (soft or hard edge) for each child of the node. For example, the Xpath expression */d[e='Mike']/b[c AND .//c AND .//d]* is translated to a pattern tree shown in Figure 2. We capitalize all of the pattern tree node labels to distinguish them from XML nodes which all use lowercase letters. For example when referring to a pattern node with ID 6, which will match all XML nodes with label "d," we refer to it by *D6*. XML nodes are referred to by their label and unique identifier, e.g., *c15* refers to node 15 with label "c." The location step "/d" in the expression is translated to the root node. The location step "/b" generates the pattern tree node *B4* and adds a hard edge

(D6, B4). The filter expression *[e='Mike']* generates the node *E5* and a soft edge *(D6, E5)*.

Finally the filter expression *[c AND .//c AND .//d]* generates three nodes *C1, C2* and *D3* and the edges connecting those nodes to the parent *B4*. The last node test inside an Xpath expression that is not inside a filter refers to the XML nodes returned from the query (for our example, *B4*). The remaining steps are explained in detail in the subsequent sections.

3.1 The *BindingCollection* Index

Step 4 of the BH method retrieves the XML nodes specified in the pattern tree (created in Step 3) from the XML tree and clustered index (created in Step 2). We process the BH pattern tree in post-order. During the traversal both pattern tree nodes and edges are visited. Nodes needed to satisfy the query are loaded into an intermediate data structure called the *BindingCollection*. During the post-order traversal, structural joins are performed and nodes that only partially satisfy the constraints are eliminated. The recursive algorithm to build a *BindingCollection* index is outlined in Figure 4.

```
1: Function PostTraversalJoin(QueryNode)
2:   For each childNode in QueryNode
3:     XMLNodeArray = PostTraversalJoin(childNode)
4:     VisitEdge(childNode, QueryNode, XMLNodeArray)
5:   End for
6:   VisitNode(QueryNode)
7: End Function
```

Figure 4: Post-order Pattern Tree Traversal

In the algorithm, *VisitNode(N)* returns a set of XML node bindings that bind to the query node *N*. If node *N* is a leaf node in the pattern tree, nodes based on the attributes of *N* (label, tree depth, and text data) are retrieved from the BH XML tree. If the pattern tree edge does not specify an attribute, such as text data, then we omit the parameter. For example, if a node is below a soft edge then it can be at any depth level and we omit the depth level parameter. The operator returns a set of nodes that bind to node *N* which are stored in the temporary variable *XMLNodeArray* that is later passed into the *VisitEdge(..)* operation described next. If *N* is a non-leaf node then a structural join operation occurs in *VisitEdge(..)*. *VisitEdge(C, P, XMLNodeArray)* visits a query edge composed of *C* the child node and *P* the parent node. Note that *VisitEdge* assumes that node *C* has been visited and all of its bindings are passed in to *VisitEdge* through the third parameter *XMLNodeArray*. If edge *(C, P)* is a hard edge then for every XML node *c* that is bound to *C* we attempt to traverse to its parent XML node *p* and check to see if *p* binds to node *P*. If *p* does bind to *P* we add edge (*c, p*) to the *BindingCollection* as a binding for edge *(C, P)*, otherwise we simply discard it. If edge (*C, P*) is a soft edge then for every XML node *c* that is bound to *C* we attempt to traverse up the parent pointers of the XML tree visiting all of the ancestor nodes until we arrive at the root node. For every ancestor node visited, denoted by *a,* we check to see if XML node *a* binds to *P*. For every ancestor node traversed that does bind to *P* we add the XML edge (*c, a*) to the *BindingCollection*.

When *N* is a non-leaf node, all of its child nodes and edges directly below *N* have been visited because of post-order traversal. For an individual query edge there could exist multiple edge bindings called an *edge binding* group. An edge binding is a parent-child tuple. We use a combination of projection and intersection operations to perform structural join shown in Figure 5. The function *getXMLAncestors(N.ID, N.Children(i).ID)* projects out only the ancestor node IDs from the edge bindings. Let *XMLNodesArray(N)* denote all of the XML node bindings found for query node *N*. We perform an intersection of ancestor node IDs from an XML edge binding in each query edge.

$$XMLNodesArray(N) = \bigcap_{i=0}^{N.ChildCount-1} getXMLAncestors(N.ID, N.Children(i).ID)$$

Figure 5: Structural Join Using Intersection

We implement the intersection operation using hash tables. We first attempt to find a value *k* such that the query edge *(N.Children(k),N)* has the minimal amount of edge bindings. Afterwards, we use *k* to access a list of ancestor nodes from the edge bindings that bind to *(N.Children(k), N)* storing them in the *SmallestList* sorted list. These operations are shown in Figure 6.

$$\min_k (getXMLAncestors(N.ID, N.Children(k).ID).ListSize)$$

$$SmallestList = getXMLAncestors(N.ID, N.Children(k).ID)$$

$$\text{where } 0 \le k \le N.ChildCount - 1$$

Figure 6: Selecting a Query Edge with a Minimal Amount of Edge Bindings

We loop through *SmallestList* keys and check to make sure its ID values are found in all of the other lists. The *getXMLAncestors* operation retrieves a sorted list with the XML ID as the key. For every ID key labeled in *SmallestList* we check to make sure all other lists have this key. If they all have the key then we consider that this XML node intersects across all of the lists and we add it to our XML binding array. Figure 7 shows our implementation of the intersection operator.

At the end of *PostTraversalJoin* execution our *BindingCollection* is populated with binding edges for all of the query edges. The last invocation of *PostTraversalJoin* returns a list of XML nodes called a *RootList* that stores the root node bindings. The *RootList* serves as an entry point into the *BindingCollection*. Figure 8 shows the *BindingCollection* for our example.

The *BindingCollection* index is built bottom up from the leaf nodes to the root. In Step 5, we traverse the *BindingCollection* top down. During the traversal, every node not satisfying all of the constraints specified in the pattern tree is eliminated.

```
1: Function VisitNodeNonLeaf(N)
   N is non-leaf returns a list of XML nodes
   Initialize XMLNodes to an empty list
   Find the minimum sized list SmalledList
2: For each XMLNode in SmallestList
3:   isFound = true
4:   For each child in N
5:     curAncList =
           getXMLAncestors(N.ID, child.ID)
6:     isFound = isFound AND
           curAncList.KeyExists(XMLNode.ID)
7:     if !(isFound) then
8:        exit inner for loop
9:     End if
10:  End for
11: if isFound then
12:    XMLNodes.Add(XMLNode)
13: End if
14: End for
15: Return XMLNodes
16: End Function
```

Figure 7: Visiting a Non-leaf Node N

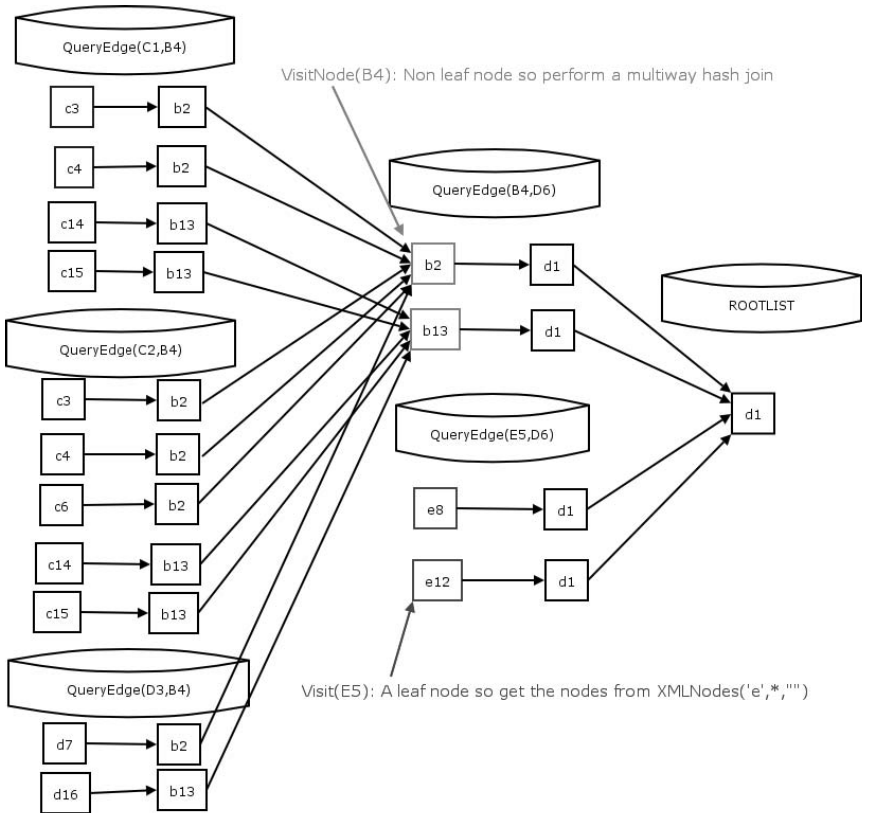

Figure 8: *BindingCollection* Example

3.2 The BranchCollection Structure

The *BindingCollection* index in the previous step allows us to retrieve bindings that are used to construct witness trees. Before generating witness trees in this step we create a data structure called *BranchCollection* containing all possible root-to-leaf XML branches built from the *BindingCollection.*

While constructing the *BranchCollection* we traverse the query using pre-order traversal. In a pattern tree, a root-to-leaf branch is a path that starts at the query root node and ends at a query leaf node. The number of root-to-leaf branches in a pattern tree is determined by the number of leaf nodes a pattern tree has. Every root-to-leaf path has a unique leaf node and we use the leaf node's ID to identify a root-to-leaf branch for a pattern tree. For example, the pattern tree in Figure 2 contains 4 leaf nodes each identifying root-to-leaf branches that are {*D6-B4-C1, D6-B4-C2, D6-B4-D3, D6-E5*}. To uniquely reference the second branch we refer to it by the leaf ID *C2*.

A query root-to-leaf branch contains structural constraints that a set of XML root-to-leaf branches needs to satisfy. Note that multiple XML branches can exist for a single root-to-leaf pattern branch. A *BranchCollection* is a sorted list and uses the query leaf ID key to retrieve a set of XML branches for a specific root-to-leaf pattern branch. Each individual root-to-leaf XML branch is itself a sorted list where either key (based on the query ID or index value) can be used to retrieve an individual XML node. Figure 9 shows a visual representation of the *BranchCollection* for the pattern tree in Figure 2 generated from the *BindingCollection* in Figure 8. Each oval represents a pattern tree node. Below each pattern leaf node, a set of XML branches are shown vertically. For example under leaf node *D3* that has the root-to-leaf path *D6-B4-D3*, there are two XML branches *d1-b2-d7* and *d1-b13-d16*.

3.3 Witness Trees from XML Branches

In the previous section we build *BranchCollection* that contains XML branches that bind to a root-to-leaf pattern branch. In this section we describe an algorithm that generates witness trees by stitching together XML branches from *BranchCollection*. XML root-to-leaf branches that bind to a single pattern tree branch are called a leaf group. In order to generate a valid witness tree a single branch must be chosen from each leaf group called a *witness branch*. If all of these witness branches satisfy a set of constraints (i.e., are compatible) then we can join (or stitch) them together to form a witness tree. If the candidate branch is found to be incompatible with at least one of the branches, then this branch is discarded. If we have successfully chosen one branch from every leaf group, then this set of witness branches can be stitched together to form a valid witness tree.

Definition 4: Two branches A and B are said to be *compatible* if they satisfy the following condition: $QueryID(A[i]) = QueryID(B[i]) \Rightarrow A[i].\mathrm{ID} = B[i].ID$, where i is an integer: $i \geq 0$, $i \leq A.Count\text{-}1$, and $i \leq B.Count\text{-}1$.

In our example, we choose an XML branch from every leaf group: *d1-b2-c3, d1-b2-c4, d1-b2-d7*, and *d1-e8*. These branches are stitched together during Step 5 in Figure 10. On the right side of the diagram we show root-to-leaf XML branches that are to be stitched together. The nodes inside the same round-edged rectangle all belong to the same query node ID labeled in boldface. On the right side the resulting witness tree is shown. The rectangles with rounded corners show the individual query node while the rectangles inside represent the XML node that binds to the query node. For our example, 10 witness trees are generated; we show three of them in Figure 11.

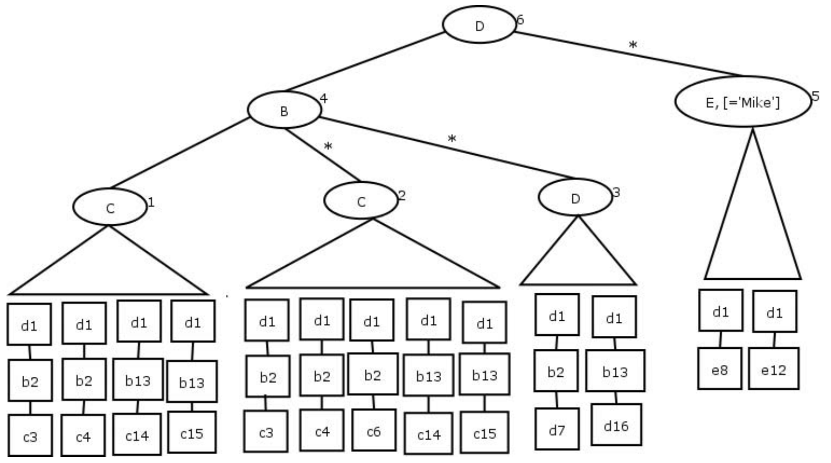

Figure 9: *BranchCollection* Example

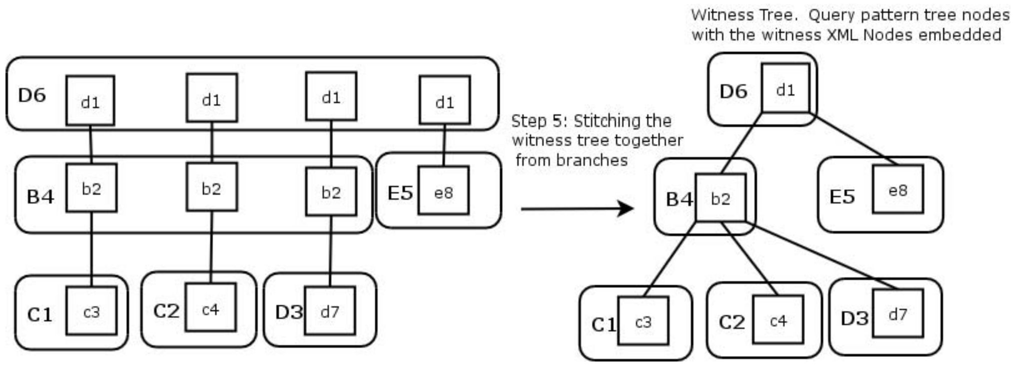
Witness Tree. Query pattern tree nodes with the witness XML Nodes embedded
Step 5: Stitching the witness tree together from branches
D6
d1
B4
b2
E5
e8
C1
c3
C2
c4
D3
d7

Figure 10: Constructing a Witness Tree from XML Branches

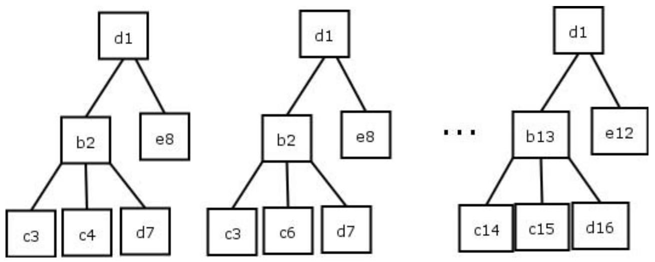

Figure 11: Witness Trees Generated by *FindWitnessBranches*

3.4 Direct Xpath Expression Evaluation

Xpath expressions specify a single pattern node *Q* for which to return a set of XML bindings. Step 7 in Figure 3 returns a set of XML nodes that bind to a single pattern node *Q* by traversing the *BindingCollection*. We construct a path from the query root *R* to *Q*. For every edge visited we retrieve a set of XML bindings from the *BindingCollection*. For our sample query in Figure 2 node *Q* is *B4*. We retrieve the nodes {*b2*, *b13*} that bind to *B4* and these are the nodes that satisfy the Xpath expression. We omit the details of this step here due to space limitations.

4. Preliminary Scalability Studies

We implemented the BH method in C# using the Microsoft .NET 1.1 framework. We conduct experiments here that give an idea how well the most computationally intensive steps scale compared to other implementations. Throughout our experiments we use the xmlgen utility developed for the XMark benchmark [20] to generate XML documents that model an auction e-commerce website. In our timing experiments we conduct each experiment three times and in our results show the average. We describe two series of experiments briefly and then focus on experiments for witness tree generation in more detail.

We use a small document size so that our results are reproducible using only RAM (disk swapping is not a factor.) Memory consumption is not studied here but it is a topic for future research.

4.1 Loading/Indexing XML

In order to study the runtime scalability of Steps 1 and 2 (loading an XML document and indexing it with our cluster index) we compared our implementation to Microsoft's XpathNavigator on an Intel Pentium 4 3.6 Ghz CPU with 3GB of memory. We varied the size of documents by adjusting the xmlgen scaling factor from 0.05 to 1. Based on 8 different document sizes, the BH load time increases linearly with the document size at a rate of 0.0003 s/KB while XpathNavigator's load time increases at a linear rate of 0.0001 s/KB. While both systems scale linearly, Microsoft's indexing method has a lower rate of increase as the document size grows. From these results we conclude that the implementation of our BH index requires further optimization or replacement with a more efficient technique that offers the same functionality.

4.2 Direct Xpath Expression Evaluation

In order to study the direct processing of an Xpath expression that returns a set of nodes (Steps 4 and 7), we perform experiments with various pattern tree shapes; we vary the number of leaf nodes (Figure 12) and the depth of the pattern tree (Figure 14). We compare the BH method with Saxon's Xpath processor version 8.7.1 open source [10] running on the Java platform. Both experiments are run on a 1.333Ghz Athlon Thunderbird CPU and 640MB DDR RAM with Windows XP SP1. As the number of leaf nodes increases, the BH Xpath processing time increases at a constant proportional rate; Saxon scales better because its rate of increase decreases logarithmically. When we measure the performance by varying the pattern tree depth level from 1 to 5, the linear regression for Saxon has an overhead of 0.0175 seconds, while the BH method has an overhead of 0.0011 seconds per node for every depth level increased by 1. We conclude that the BH method scales better as pattern tree depth level increases.

4.3 Witness Tree Generation Scalability

Steps 4, 5 and 6 of the BH method return a set of witness trees that bind to a pattern tree. We compare its scalability against Timber [7] with the implementation that uses the more recent annotated pattern tree [8, 17]. Our approach is to see how well the BH method and Timber performance scales based on the pattern tree depth level and leaf node count. We conduct two experiments both of which incrementally add nodes to a particular pattern tree. The hardware setup for these two experiments is identical to the one described for direct Xpath evaluation in Section 4.2. In both of these experiments the XMark data generator was used to generate an XML document using a size scaling factor of 0.03 (3485 KB).

In our first experiment, we use pattern trees with 1 to 6 leaves; Figure 12 shows a BH pattern tree with all 6 leaves. We plot the witness tree generation times of the two systems on the same graph, shown in Figure 13. We perform linear regression to calculate the runtime overhead associated with adding a new leaf node. The additional runtime overhead per query leaf node in BH is about 1.63 seconds while for Timber its 0.4177 seconds. We conclude that Timber has better scalability as the number of leaf nodes in a pattern tree increases.

In the second experiment we incrementally increase the depth level of the query while keeping the leaf node count constant at 1. We generate witness trees for the five pattern trees shown in Figure 14. We plot the BH and Timber witness tree generation runtime in Figure 15 and perform linear regression. Increasing the depth level by 1 increases the runtime of the BH method by 0.0151 seconds while Timber has a 0.7138 second overhead. Based on these results we conclude that the BH method has better scalability as the query depth level increases.

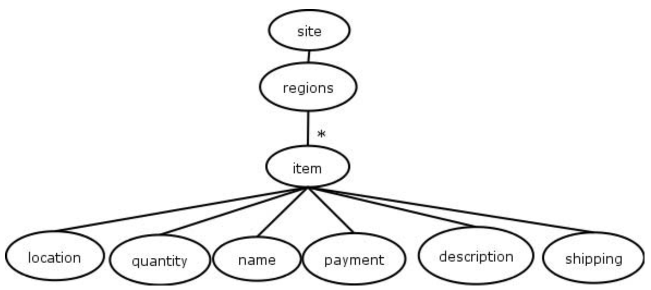

Figure 12: BH Pattern Tree with 6 Leaves

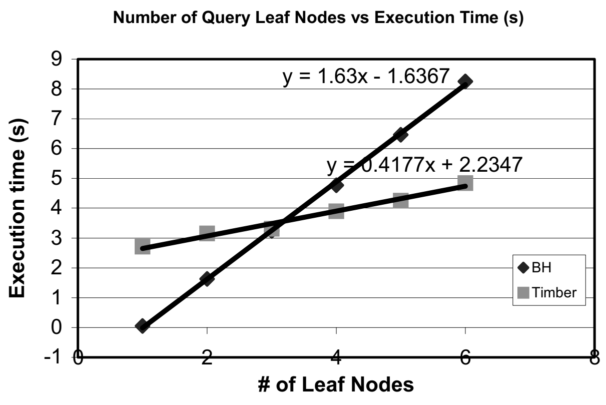

Figure 13: Witness Tree Generation Time for Varying Leaf Node Counts

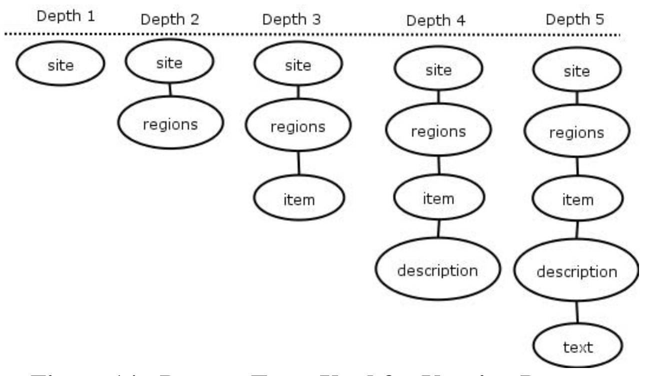

Figure 14: Pattern Trees Used for Varying Pattern Tree Depth

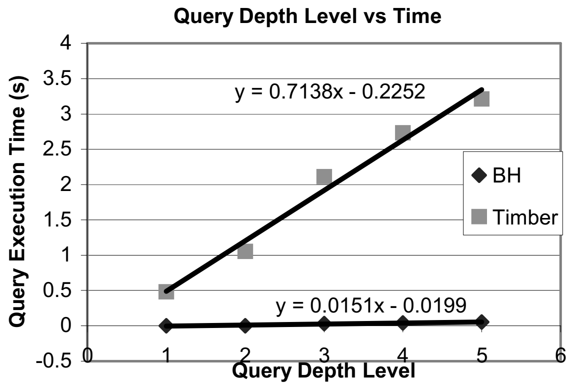

Figure 15: Witness Tree Generation Times with Varying Pattern Tree Depth

5. Conclusions

We introduce the Binding Hash method that performs structural joins and value-based selection. The BH method can generate witness trees to be used in systems with XML languages that embed Xpath expressions as well as with systems that directly process Xpath expressions.

The *BindingCollection* is flexible and can be used for XPath evaluation or witness tree generation; Xpath expression evaluation does not need to perform all of the traversal operations for the witness trees. By comparison, Timber develops its own TAX and TLC algebras [17] that are complex and require new implementation algorithms. We focus on transforming the structural join problem into well known existing problems (projection, pre/post traversal, relational tuple intersection) for which efficient techniques already exist.

For deeply nested queries (based on preliminary testing) the BH method has a performance advantage and could fit in well as an option within an XML query optimizer such as Timber. The *BranchCollection* (generated by a well known pre-traversal method) combined with a single compatibility check rule solves the complex problem of constructing valid witness trees.

Based on a comparison with Microsoft's XpathNavigator, we conclude that their indexing technique scales better than ours, and our indexing technique requires further optimization to improve processing time. We then characterize the performance of our query processor for different pattern tree shapes. We conduct experiments and compare them against competitive systems, Timber and Saxon. We observe that the BH method performs relatively better when generating witness trees and processing XPath expressions when the query is deeply nested and non-bushy.

Future work regarding the BH method includes the following:

1. Add capabilities for inequality-based selection.
2. Allow a single XML node to contain an arbitrary number of text values. Currently our implementation allows a single XML node to have a single text entry.
3. Expand the post-order traversal algorithm and the *BindingCollection* index so the algorithm can process TAX features that we currently do not support such as:
   - value-based join: allows a constraint requiring attributes across different XML nodes to have exactly the same value. For example, "return all students who belong to the same department" requires the value attribute of all department XML nodes to have the same value.
   - multiple selection: allows more then one constraint against a single XML attribute. For example, "return all students whose name starts with an M and ends with l," specifies two constraints for a *name* attribute against the text attribute for the *name* XML node.
4. Incorporate the BH method into a complete query processing system that supports query languages such as Xquery and XSLT that embed Xpath expressions.

In general, a more extensive performance study for the BH method is needed, as well as an analytical study of space and performance complexity compared to other proposed XML indexing techniques such as those in Lore [6, 15, 16], the Toronto XML Engine (ToXin) [2, 18], the XML Region (XR) tree [9], and the XB tree [12]. In addition, the performance of XML query processing using sequences [13, 19] can be studied relative to the BH method. The idea is to transform the XML tree and query into sequences; then performing a subsequence match for the query is equivalent to performing a structural join against an XML tree. Other hashing-based approaches [11, 14] should be studied as well as techniques that use pipelining instead of in-memory indexing [3, 5] to avoid excessive memory consumption.

6. References

[1] Abiteboul S., Buneman P., and Suciu D., *Data on the Web: from Relations to Semi-structured Data and XML*, Morgan Kaufmann, 2000.

[2] Barbosa, D., Barta, A., Mendelzon, A.O., Mihaila, G.A., Rizzolo, F., and Rodríguez-Gianolli, P., "ToX - the Toronto XML Engine," *Proceedings of the Workshop on Information Integration on the Web,* Rio de Janeiro, Brazil, April 9-11, 2001, pp. 66-73.

[3] Brantner, M., Helmer, S., Kanne, C.C., and G. Moerkotte, "Full-fledged Algebraic XPath Processing in Natix," *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE),* Tokyo, Japan, pp. 705-716, April 5-8, 2005.

[4] Extensible Markup Language (XML) 1.0 (Third Edition), Febuary 2004, http://www.w3.org/TR/REC-xml/#sec-intro.

[5] Fernandez, M., Siméon, J., Choi, B., Marian, A., G. Sur, "Implementing XQuery 1.0: The Galax Experience," *Proceedings of the 29th Very Large Databases Conference (VLDB)*, Berlin, Germany, 2003.

[6] Goldman, R., and Widom, J., "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB),* Athens, Greece, pp. 436-445, August, 1997.

[7] Jagadish, H.V., Al-Khalifa, S., Chapman, A., Lakhsmanan, L.V.S., Nierman, A., Paprizos, S., Patel, J. M., Srivastava, D., Wiwattana, N., Wu, Y., and Yu, C., "TIMBER: A Native XML Database," *VLDB Journal*, pp. 279-291,Vol. 11, No. 4, Springer, 2002.

[8] Jagadish, H. V., Lakshmanan, L. V., Srivastava, D., and Thompson, K. "TAX: A Tree Algebra for XML," *Proceedings of the 8th International Workshop on Database*

Programming Languages, Rome, Italy, pp. 149-164, September 2001.

[9] Jiang, H., Lu H., Wang W., and Chin Ooi, B., "XR-Tree: Indexing XML Data for Efficient Structural Joins," *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE)*, Bangalore, India, pp. 253-264, March 2003.

[10] Kay, M.H., Saxon, April 2006, http://saxon.sourceforge.net/.

[11] Li, Q., and B. Moon, "Partitioned Based Path Join Algorithms for XML Data," *Proceedings of the 14th International Conference on Databases and Expert Systems (DEXA)*, Prague, Czech Republic, pp. 160-170, September 1-5, 2003; *Lecture Notes in Computer Science*, Vol. 2736, Springer-Verlag.

[12] Li, H., Lee, M. L., Hsu, W., and Chen, C., "An Evaluation of XML Indexes for Structural Join," *SIGMOD Record,* No. 33, pp. 28-33, September 2004.

[13] Manolescu I., Arion A., Bonifati A., and Pugliese, A., "Path Sequence-Based XML Query Processing," *Journées de Bases de Données Avancées (BDA),* Montpellier, France, October 2004.

[14] Mathis, C., and T. Haerder, "Hash-Based Structural Join Algorithms," *Proceedings of the Current Trends in Database Technology Workshop (EDBT)*, Munich, Germany, pp. 136-149, March 26-31, 2006; *Lecture Notes in Computer Science*, Vol. 4254, Springer-Verlag.

[15] McHugh J., Widom J., Abiteboul S., Luo, Q., and Rajaraman, A., "Indexing Semistructured Data," Technical Report, Stanford University, January 1998.

[16] McHugh J., and Widom, J. "Query Optimization for XML," *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB),* Edinburgh, Scotland, U.K., pp. 315-326, September 1999.

[17] Paparizos, S., Wu, Y., Lakshmanan, L.V., and Jagadish, H. V., "Tree Logical Classes for Efficient Evaluation of XQuery," *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data,"* Paris, France, pp. 71-82, June 2004.

[18] Rizzolo, F., and Mendelzon, A.O., "Indexing XML Data with ToXin," *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB),* Santa Barbara, California, USA, May 24-25, 2001, pp. 49-54.

[19] Wang, H. and Meng, X., "On the Sequencing of Tree Structures for XML Indexing," *Proceedings of the 21st International Conference on Data Engineering (ICDE),* Tokyo, Japan, pp. 372-383, April 2005.

[20] XMark-An XML Benchmark Project, June 2003, http://monetdb.cwi.nl/xml/.

[21] XML Path Language (XPath)Version 1.0, November 16, 1999, http://www.w3.org/TR/xpath.

[22] XQuery 1.0: An XML Query Language, November 2005, http://www.w3.org/TR/xquery.

[23] XSL Transformations (XSLT) Version 1.0, November 1999, http://www.w3.org/TR/xslt.