I, Michael J. Brant, hereby submit this work as part of the requirements for the degree of:

Master’s

in:

Computer Science

It is entitled:

Binding Hash Technique for XML Query Processing

This work and its defense approved by:

Chair: Karen C. Davis

Raj Bhatnagar

Chia-Yung Han
Binding Hash Technique for XML Query Optimization

A thesis submitted to the

Division of Graduate Studies and Research
of the University Cincinnati

in partial fulfillment of the requirement for the degree of

MASTER OF SCIENCE

In the Department of Electrical and Computer Engineering
and Computer Science

May 19, 2006

By

Michael Brant

B.S., University of Cincinnati, Cincinnati, OH 2004

Thesis Advisor and Committee Chair: Dr. Karen C. Davis
Abstract

XML is a format that allows the storage and exchange of information across the World Wide Web. XML is a semi-structured markup language containing recursively-nested elements. Typically the volume of data in an XML file is too large to be human readable, therefore XML query processing (retrieving and combining subtrees) needs to be automated. An XML query processor has to choose an efficient method for a particular query and XML document. In this thesis we develop a method called Binding Hash (BH) that performs a subset of XML query operations. The BH Method focuses on improving performance of the most time-consuming XML query operation called structural join. A performance study indicates that the BH Method outperforms similar techniques for queries that are deeply nested. The BH Method is flexible since it can be integrated into a larger system and selected by an optimizer when it achieves the best performance.
Acknowledgements

This thesis would not have been possible without the direction, support and dedication of my advisor Prof. Karen C. Davis. Special thanks go to my committee members Prof. Raj Bhatnagar and Prof. Chia-Yung Han for their support and pleasant cooperation.

My sincere thanks to my parents Joseph Brant and Elena Brant, and grandparents Michael Plotkin and Galina Plotkin for their encouragement and dedicated support. To them I dedicate this thesis.
# Table of Contents

1. Introduction

1.1 General Research Objective

1.2 Specific Research Objectives

1.3 Research Plan

1.4 Expected Contributions

1.5 Overview of Contents

2. Survey of Existing XML Processing Techniques

2.1 XML Data Model

2.2 XML Query Languages

2.3 Indexing Techniques

2.4 XML Query Plans

3. Theoretical Foundations and Definitions

3.1 BH Data Model

3.2 Sorted List Data Structure

3.3 XPath Language Subset for Binding Hash

3.4 Pattern and Witness Trees

4. Binding Hash Method

4.1 Loading the XML Document

4.2 Retrieval of XML Nodes using the XML Index

4.3 Loading XPath Into a Pattern Tree

4.4 Building the BindingCollection Index using Post-order Traversal

4.5 Generating Branches Using Pre-order Traversal

4.6 Generating Witness Trees from XML Branches

4.7 Generating Nodes to Satisfy the XPath Expression

5. Comparison and Evaluation

5.1 XML Index

5.2 Witness Tree Generation Scalability

5.3 XPath Processing Scalability

6. Conclusions and Future Work

6.1 Conclusions

6.2 Future Work

Bibliography

Appendix A: XQuery and Pattern Trees for Experiment 2

Appendix B: XQuery and Pattern Trees for Experiment 3

Appendix C: Binding Hash Shakespeare Demo
Chapter 1

Introduction

The World Wide Web has become an important medium for exchange of information. Originally humans generated the documents that we see on the web, but these documents are increasingly machine generated. Therefore there is a demand for a standard exchange format that is both machine and human readable. XML is such a format that allows the exchange and storage of information across the web. XML is a semi-structured markup language containing recursively-nested elements. Typically the volume of data in an XML file is too large to be human readable, therefore query processing needs to be automated. The sample XML document in Figure 1.1 stores information about a meeting such as place and participants. In the document, the text inside the angle brackets denotes element tag names. For example, <Name> denotes the start of an element tag while </Name> denotes the end of the element tag. <Name>…</Name> constitutes an element block. The text inside the element tag block (not inside the angle brackets) is the associated text data.

```xml
<?xml version="1.0" encoding="us-ascii"?>
<Meeting ID="1">
  <NormalParticipants>
    <Participant>
      <Name>Michael Brant</Name>
      <Department>ECECS</Department>
      <EmailAddress>brantmj@email.uc.edu</EmailAddress>
      <PhoneNumber>513-111-1111</PhoneNumber>
    </Participant>
    <Participant>
      <Name>Jason Boyle</Name>
      <Department>ECECS</Department>
      <EmailAddress>Jason@email.uc.edu</EmailAddress>
      <PhoneNumber>513-222-2222</PhoneNumber>
    </Participant>
  </NormalParticipants>
  <MyLocation>Room 800 ERC Building</MyLocation>
</Meeting>
```

Figure 1.1: Sample XML document
An example query over the document in Figure 1.1 is “List names of all participants who are in the ECECS department.” The XML document’s exact structure is not known ahead of time without a schema. If the XML data is without a schema, XML data path processing is needed to address the issue. Broadly speaking, XML query processing involves two main operations: traversal of the XML document and the gathering of the results to satisfy the query. Traversal of the XML document typically involves building an index against the XML document. The most important operations involved in the gathering phase include structural joins, structural projection, value-based selection, and value-based joins. Processing the sample query against an XML document involves finding <Participant> element tags that contain a <Name> tag. Finding and combining these element tags is performed by the structural join operation. Typically value-based selection involves matching text values found inside element tags. For example, in our example a portion of the query involves finding all <Department> element tags that have “ECECS” as the text value. Structural projection involves returning only the <Name> element tags while filtering out all other nodes. Structural join and structural projection are unique to XML queries and do not exist in relational databases. One final item to note is that the structural join operation typically has a lot more complexity compared with the other operations because of the extensive scanning of the document it needs to do. Because the structural join operation has the most overhead, we focus on this operation in our research.

In any DBMS, query processing efficiency is a crucial factor in determining its success. With the added complexity of semi-structured data it becomes a challenge to develop an XML query processing method that can process a query efficiently in all cases. An XML query processor has to choose the best method for the particular query and XML document. Therefore we develop a method that attempts to perform a limited subset of XML query operations efficiently for certain types of queries and XML documents. This method can be integrated into
a larger system where it can be selected as the method of choice when it achieves the best performance.

Numerous XML query languages have been proposed: XPath, XQuery, XSLT, Lorel and XML-QL [4]. While XPath is the simplest language and may not have the expressive power of the other languages mentioned, it retains the most crucial features that all of the other XML query languages share. One such feature is path expressions. An XML document can be modeled as a set of paths from the root node. A path expression is a sequence of path labels from the root to a particular node. Simple path expressions conform to the \(l_1/l_2/l_3/\ldots/l_k\) syntax where \(l_1\) indicates the root node label, and the \(l_2\ldots l_{k-1}\) indicates the intermediate labels to traverse, and \(l_k\) indicates the ending label. For example, to reach all of the `<Name>` elements tags of the sample XML document in Figure 1.1 the path expression

\[
/\text{Meeting}/\text{NormalParticipants}/\text{Participant}/\text{Name}
\]

is used. In our research we focus on XPath queries that contain path expressions to compute both structural join and value-based selection.

1.1 General Research Objective

In the research, we introduce a new method called Binding Hash (BH) that performs XML query processing by focusing on the important structural join operation of XML query processing.

1.2 Specific Research Objectives

We address the following issues in this thesis

1. Consider the current state of the research in XML query processing to have a better understanding of what underlying issues need to be solved.

2. Develop a new method called Binding Hash (BH) that executes an XML query in the form of an XPath expression and performs structural join and value-based selection.
3. Compare performance against similar techniques to determine when to select BH as a method of choice in a complete XML processor.

1.3 Research Plan

In order to address the research objectives we conduct the tasks below.

1. Research the operations and issues involved in XML query processing. Provide an overview of existing techniques that handle the common operations in XML query processing.

2. Describe the Binding Hash method that specifically focuses on processing XPath expressions. The Binding Hash method has four general steps where each of them has associated data structures and algorithmic operations to be performed.
   a) Load the XML tree into an XML indexing structure.
   b) Parse an XPath expression into a tree structure.
   c) Perform structural and value-based selections that result in retrieving XML data that comprises part of the final answer.
   d) Combine XML data found in the previous step to output the query answer.

3. Analyze the performance of the algorithms for different query classes and XML documents.

1.4 Expected Contributions

In order to satisfy our research goals, techniques are surveyed that focus on the structural join aspect of XML query processing. Then the Binding Hash (BH) method is introduced. The BH consists of a combination of several indexing data structures and algorithms. Then we describe how this technique relates to existing methods. Finally we experimentally analyze the BH Method and compare it existing methods to determine which scenario BH Method is suited best for.

1.5 Overview of Contents

The thesis is divided into six chapters. Chapter 2 surveys existing methods that handle different aspects of XML query processing. Chapter 3 describes in detail the concepts and data models that are utilized throughout the BH Method. Chapter 4 describes the major steps of the
BH Method including their data structures and algorithms. Chapter 5 analyzes the BH Method’s performance and compares it with similar competitive techniques. In Chapter 6 we summarize our contributions, discuss our conclusions and define future research issues.
Chapter 2

Survey of Existing XML Processing Techniques

This chapter provides an overview of existing methods that handle different aspects of XML query processing. In introducing the existing methods we gain a foundation upon which to build our BH Method. Section 2.1 introduces different XML data models. Section 2.2 describes different XML query languages and why choosing XPath for BH addresses the most important aspect of XML query processing. Section 2.3 introduces existing XML indexing techniques. Section 2.4 focuses on logical and physical query plans.

2.1 XML Data Model

The XML acronym stands for eXtensible Markup Language. The goal behind XML is to describe data and its structure in one document. This is different from a traditional database system where a schema and data tables are separately stored. XML was designed as a standard cross platform exchange format between different systems. An XML document is composed of entities that contain data and markup. In the XML document the markup forms the structure and layout of the document. When the XML document is initially loaded, the markup tags are parsed while the data remains unparsed. Typically an XML tree is loaded into a Document Object Model (DOM) that is a tree where the parent-child edges represent the element tag nesting such shown in Figure 2.3.

Many markup types exist in the XML document such as elements, attributes, declaration, entity references, and comments. The most common XML markup types found in XML are element tags and attributes. Element tags are delimited by angle brackets. Typically an element has nested content surrounded by a start and end tag. For example the document in Figure 1.1 the <Participant> is the starting tag and </Participant> is the ending tag while the <Name> tag is
internally nested inside the participant element. Element tags can also contain text data. For example the first <Name> tag contains the data “Michael Brant.” An attribute always belongs to an element, contains a label, and a value. For example, the element tag name <Meeting> has an attribute named “ID” whose value is 1.

An XML document can contain entity references using a special kind of attribute called IDREF. If these IDREF relationships are modeled using relationship edges, then the XML document can be modeled by a graph.

A graph-based model can be used to model an XML document called Object Exchange Model (OEM) [2] where the vertices represent an object that may contain data while the edges represent the element tag. Every object has a unique object identifier (oid) denoted by the “&” prefix. The directed edges can represent IDREF relationships. Many of the existing XML query processing techniques that we present in this chapter use OEM to model an XML document.

Figure 2.1: OEM Model for XML

Another model for an XML database is an ordered tree. In this model each node can correspond to either an element or a value while the edges correspond to the element tag nesting relationship. While in the OEM model IDREF attributes are modeled as edges in the ordered tree.
data model IDREF attributes are simply treated as any other data. Figure 2.2 shows the XML document from Figure 1.1 as an ordered tree. Each node that is an oval represents either an element tag or attribute and the inside text is the corresponding label. The rectangular nodes represent text data and inside contain text data. The ordered tree model is the most common representation currently used to represent an XML document, therefore in our research we focus more upon it then OEM. Our BH Method uses a slightly modified ordered tree model to represent the XML data.

![Figure 2.2: Ordered Tree Model for the XML Document](image)

Having introduced a data model for XML we next introduce XML query languages. Most XML query languages enable a user to retrieve the needed XML nodes from documents. XML query languages contain many different features and we point out the most important ones.

### 2.2 XML Query Languages

A core feature of an XML query language is a path expression that allows the user to retrieve a set of nodes at arbitrary depth. Path expressions essentially treat the XML document as a UNIX directory hierarchy. Although path expressions are the most crucial portion of XML query processing they are only one feature of an XML query language. In addition to processing
path expressions, typically using structural joins, a complete language can perform value-based joins on the text data, value-based selection and projection [4]. The languages discussed below all contain path expressions with some syntactic variation. Each language provides some of the features listed. We first introduce Lore because of its historical significance. Then we discuss the most commonly used languages: XPath, XQuery and XSLT. Because XML-QL is rarely used it is not discussed here, but it has a feature set found in XQuery.

Lore is a fully implemented DBMS system with its own query language and optimization engine [2] to handle semi-structured data. It is based on the OEM graph based data model and has been extended to handle XML. The Lore language (Lorel) is a declarative language that allows for XML document navigation and updating. Figure 2.3 Lorel query for the query “List all names of all normal participants who are in the ECECS department.” The *Meeting.NormalParticipants.Participant, P.Name* and *P.Department* portions of the query are path expressions and define a structural portion of the query to match. *Meeting* serves as the entry point into the database while the *NormalParticipants* must be a child node of *Meeting*. The *P.Department= “ECECS”* defines the value based match of the query. Note that *P* is a variable that must be bound to an actual node in the XML document.

\[
\text{SELECT P.Name} \\
\text{FROM Meeting.NormalParticipants.Participant P} \\
\text{WHERE P.Department = “ECECS”}
\]

Figure 2.3: Lorel Query Example

XPath expressions have fewer capabilities then Lorel but contain the essential features of XML query processing such as value-based matching, value-based joins and path expressions. For example, unlike Lorel, path expressions cannot construct new nodes and can only project one set of nodes. For example, the query in Figure 2.3 has the following equivalent XPath expression

/Meeting/NormalParticipant/Participant[Department='ECECS']/Name
where “/” between the two element names indicates a parent-child relationship and “//” indicates the more general ancestor-descendant relationship. The expression inside the brackets is a called a filter and is a predicate expression that needs to be satisfied. Furthermore the XPath language contains numerous numerical and string matching functions. While these features are important in real life use they are not unique to XML query processing therefore we do not focus on them in our research.

XQuery version 1.0 is being finalized to become the standard XML Query Language [6]. The XPath language is one of the major pieces embedded within XQuery. XQuery also borrows ideas from XML-QL, Quilt and SQL [4]. Like SQL which contains SELECT, FROM, WHERE, GROUP BY and ORDER BY clauses Xquery expression contains FOR, LET, WHERE and RETURN clauses. The FOR/LET clauses retrieve XML nodes, the WHERE clause prunes out the XML nodes from the FOR/LET clause, and the RETURN builds the instance of the answer.

The Extensible Stylesheet Language Template (XSLT) is a specialized XML transformation language. Its primary use is to construct a new document called a result tree from an existing XML document [7]. The result tree can be any text format such as HTML, XML or simply a delimited value text file. Because XSLT is a specialized language its capabilities are more limited then XQuery [4]. Unlike the other XML query languages discussed, an XSLT template is itself a valid XML document. In order to scan the XML document, XSLT relies heavily on the use of XPath expressions. Because XPath expressions are a core part of XSLT, our research of processing XPath expressions could be used to optimize XSLT document processing.

2.3 Indexing Techniques

In order for a query processing algorithm to work effectively, an efficient method is needed to retrieve relevant portions of the XML document to satisfy a query. The access mechanism to an XML document is called an XML index. Numerous data structures for are developed for relational databases such as B+ trees, and many of these are reused for XML trees.
XML indexing techniques load the document based on the structure and data. One challenge not found for traditional databases is that an XML document typically does not have a schema. Without a schema the indexing technique has to dynamically infer the structure of the document, increasing the complexity of query processing. In our search we assume that all of our XML documents contain no schema. We discuss the earliest indexing techniques for XML found in Lore. Then we discuss a labeling scheme that is used in conjunction with variants of B+ trees. We also describe a technique based on the Data Guide technique. Finally we discuss the more recent method of using sequencing for XML indexing.

The DataGuide was developed for semi-structured data in the Lore system. A DataGuide’s responsibility is to maintain structural summaries for OEM databases [8]. A DataGuide can be thought of as a dynamically managed structural schema generated directly from the XML document. Note that the DataGuide does not perform any indexing against values and thus can only serve as a path index (Pindex). In Figure 2.4 a DataGuide is shown for the XML document introduced in Figure 1.1. The diagram for the DataGuide also contains additional labeling inside the parentheses that will be used for a special technique based on the DataGuide explained later in this section. Note that even though we have two different participant sub-trees, since the two sub-trees are structurally identical we can summarize the structure with only one participant sub-tree. Using the Lore syntax for the path expression

\[ Meeting.Nor\text{malParticipants}.Participant \]

the DataGuide would retrieve both of the nodes without having to scan both sub-trees.
Lore has numerous other indexes that we briefly mention. The value index (Vindex) is used for comparison operations involving strings and integers. The text index (Tindex) is used for string keyword searches. Finally the link index (Lindex) allows quick retrieval of a child node’s parent.

The Toronto XML Engine (ToXin) [20] uses both path and value-based indexes. The path-based index is based on the minimal DataGuide [8] and stores structural information about the document. The instance tables labeled store this structural information. The value-based index specializes in retrieving data based on its value stored in a value table. Figure 2.5 contains the instance and value tables that correspond to a single path in the Data Guide from Figure 2.4. The instance tables labeled IT contain the parent child information while the value tables labeled VT store textual data. Each edge label inside the parentheses in Figure 2.4 corresponds to a table in Figure 2.5. For example the edge label IT3 has a table with two entries. Each entry corresponds to the two XML edges that correspond to the IT3 summary edge.
The techniques previously described use a labeling technique that assumes every node has a unique identifier. However if the nodes are labeled based on their structural arrangement, then we can infer structural information about nodes by simply looking at the labels. This can avoid the overhead of having to redundantly scan the XML tree. Interval-based schemes require each node to have a start and stop value where $\text{start} < \text{stop}$. One such proposed interval based scheme consists of labeling each node with a $(\text{PreOrder}, \text{PostOrder})$ pair [10]. Figure 2.6 shows the ordered tree model for the XML document in Figure 1.1 using this labeling scheme.
Using the following labeling scheme node A is an ancestor of node B if and only if the conditions $A.\text{Start} < B.\text{Start}$ and $A.\text{Stop} > B.\text{Stop}$ are true. For example if we choose two nodes from the XML tree from Figure 2.6 labeled (2, 21) with the tag name “NormalParticipants” and (4,2) with the tag name “Name.” Using the rule above we see that that $2 < 4$ and $21 > 2$ therefore we can conclude that “NormalParticipants” is an ancestor of “Name” without having to scan the XML tree. A disadvantage of this scheme is that during a node insertion or deletion requires re-indexing the entire tree. Therefore other interval based labeling schemes exist that process updates to the tree without having to re-index the entire tree. Such indexes are called *durable* \[11\]. The BH indexing scheme only makes the assumption that every node contains a unique identifier. The BH indexing scheme is *durable* because it doesn’t require re-indexing the entire tree when performing a node insertion or deletion for the XML tree.

We can use a B+ tree to index the XML tree using the interval labeling scheme defined earlier. A B+ tree is a variant of the B tree where the data pointers reside only at the leaf nodes. The non-leaf nodes have a different structure and only serve to guide the search. The B+ tree has the advantage of having a small update overhead while still allowing each node to be accessed in $\log(n)$ time. When indexing XML data, the B+ tree indexes against the *start* value of each node.
Such a scheme can increase efficiency by eliminating unnecessary nodes before a structural join takes place. Specifically the B+ tree eliminates the unnecessary descendant nodes.

The XML Region (XR) tree is a modified B+ tree that uses stab lists for its internal nodes [5]. The main advantage an XR tree has over the regular B+ tree is that it can eliminate unnecessary ancestor nodes in addition to eliminating descendant nodes. One disadvantage the XR tree has is it does not handle a deeply recursive tree very well because of its increased maintenance cost. Therefore another variant has been proposed called the XB tree [17]; it eliminates duplicate copies of data yielding better update and space overhead. The B+ tree and their variants can be used in place of the BH indexing scheme within the entire BH Method which is left as a future work study.

Finally a more recent but different approach to indexing involves the use of sequencing [12]. 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. In our research we do not focus on sequencing and our structural join algorithm focuses on utilizing the more traditional indexes introduced earlier.

Having discussed XML indexes that allow efficient retrieval of XML nodes, we discuss query plans in the next section. Query plans specify a sequence of retrieval and update operations against the XML tree utilizing XML indexes.

2.4 XML Query Plans

The XML tree model, query languages, and XML indexing techniques are individual pieces that all contribute to the research area of XML query optimization. In a broader picture, 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 [2] and the Timber XML framework [13] are two major XML database management systems (DBMS) that
address these underlying issues of XML query processing. We now focus our attention on physical and logical query plans.

In a relational DBMS an SQL query is converted into a logical plan called a relational algebra tree. The relational algebra tree describes a sequence of retrieval and update operations over relational data. Using the relational algebra properties a single query can be represented with multiple equivalent relational algebra expressions. Ideally the relational algebra expression should be rearranged in such a way as to minimize the size of the intermediate data and to minimize execution time. After the best tree algebra plan is chosen it is transformed into a physical plan. The physical plan deals with the data structures and algorithms to use in executing the logical operations. Note that multiple physical plans can be generated for a single logical plan.

An XML query is similarly converted into a logical plan called a twig or pattern tree. In Lore [2] the logical query plan is specifically developed to deal with the Lorel language. Before creating a physical query plan a logical query plan is constructed that can be transformed into multiple physical query plans. In Lore, one of the logical operations defined is the $\text{Discover}(x, l, y)$ operator. It is responsible for matching a single parent-child structural pattern. The variables are defined as follows: $x$ is the parent object id, $l$ is the edge (element tag) and $y$ is the object identification to bind. The $\text{Chain}(d1, d2)$ operator is responsible for choosing the best way of evaluating a path expression for joining the two $\text{Discover}(x, l, y)$ bindings. The two bindings are denoted by $d1$ and $d2$. Figure 2.7 shows the logical plan for the path expression $\text{Meeting.NormalParticipants.Participant P}$ from the sample Lore query in Figure 2.3.
To implement the logical operator **Discover**, six physical scanning operators are used in Lore. A nested loop join (NLJ) can be used to implement the final join operation. In NLJ the left subplan passes its bindings to the right side subplan and the right subplan becomes the final output.

In order to optimize a physical query plan, statistics and a cost model are defined. The statistical model involves storing information about any subpath that does not have to start from the root node of the XML document. For every subpath the following statistics are stored: number of nodes reachable, number of instances of the path, number of distinct objects reachable, and number of incoming edges. Combining such information allows Lore to estimate disk I/O cost for all of the physical operators.

Note that originally Lore’s originally was design for semi-structured data and didn’t initially support XML. In the next introduce an XML DBMS built specifically for XML and we build upon its ideas and concepts when designing the BH Method.

The Timber DBMS framework was developed from the ground up for XML at the University of Michigan [13]. The core of Timber is based on Tree Algebra for XML (TAX) [14]. Like relational algebra, which was developed for relational data, TAX is a tree algebra developed for manipulating XML data. Relational operators deal with tuple sets where each tuple has identical structure. Manipulating a set of XML trees presents additional challenge because trees
are heterogeneously structured. TAX attempts to solve this problem through the use of a pattern tree. A pattern tree is a node constraint specification that returns a set of homogenous trees called witness trees.

The generation of witness trees is a crucial logical step in Timber. An XQuery statement is decomposed into multiple pattern trees. Once the witness trees are produced there are special operators that further manipulate sets of homogenous trees before the query results are returned. For the path expression “/Person[Department="ECECS"]/Name” the pattern tree is shown in Figure 2.8 and the returned witness trees are shown in Figure 2.9. The BH Method utilizes a subset of the TAX algebra and generates a set of witness trees. Within Timber, the BH Method can be a method of choice for generating witness trees.

![Figure 2.8: Pattern Tree](image)

![Figure 2.9: Witness Trees](image)

The Timber project has been extended to use a more sophisticated algebra to generate more efficient logical query plans. Tree Logical Class (TLC) algebra [15], based on TAX algebra, can handle a set of heterogeneous data directly. In TLC the pattern tree has been
extended to an Annotated Pattern Tree that has the ability to return heterogeneous data. The increased flexibility avoids redundant lookups that can occur using TAX algebra plans.

A more recent system Saxon [18] developed by Michael Kay is an XML engine which has the ability to process Xquery, XSLT, and XPath expressions. Saxon offers competitive performance and we will use it to perform some of our experiments against in Chapter 5.

In this chapter we provide a general overview of concepts and techniques within the field of XML query processing. In the next chapter we build upon those concepts that are used by the BH Method in Ch 4.
Chapter 3

Theoretical Foundations and Definitions

In the previous chapter we introduced different concepts and aspects of XML query processing that we build our BH Method upon. In this chapter we discuss concepts and definitions in more detail that are specifically used in the BH Method. Section 3.1 introduces the BH data model. Section 3.2 discusses the sorted list data structure that is extensively used in the BH Method. Section 3.3 introduces a subset of XPath the BH Method can parse. Section 3.4 defines the TAX pattern and witness trees and discusses the subset of pattern tree features the BH Method supports.

3.1 BH Data Model

The BH Method XML data model is almost identical to the ordered tree model and an example is graphically represented in Figure 3.1. The main difference is that the element label and its text data are contained within one node. In the BH data model the attribute label and data information are paired with one node, denoted by a hexagon shape, allowing us to visually differentiate element tag and attribute nodes that the ordered tree model does not. Note that the numbers next to the nodes represent a unique identifier for the node.
3.2 Sorted List Data Structure

The BH Method often uses a data structure called a sorted list; we discuss the common operations and the syntax of the sorted list here. The sorted list object and the corresponding operations we define are based on the SortedList class defined in the Microsoft.NET framework. The sorted list is referred to in Chapter 4 during our discussion of the BH Method.

A sorted list is a hybrid collection type containing the features of both dynamic arrays and hash tables. Recall that to access an object (array element) arrays use an index while hash tables use a key. Using a sorted list, the user can access an object in constant time from a collection by either providing a key or an index. If we want to access the sorted list named $\text{SortedListName}$ using a key we use the $\text{SortedListName}[\text{key}]$ notation. A key can either be a numerical or string type. Also, we can easily check in constant time to see if a value with a particular key exists using the $\text{SortedListName.KeyExist(\text{key})}$ function which returns a boolean value. For accessing a value by an integer index, we use the $\text{SortedListName.ByIndex(\text{i})}$ notation. The operator $\text{SortedListName.GetKey(\text{i})}$ returns a key value for index an index $\text{i}$. 

![Figure 3.1: Modified Ordered Tree Model](image-url)
The operators associated with a sorted list are used in the pseudocode in Chapter 4 when we discuss our BH Method. Our next task is to define the XPath language subset which the BH Method supports.

3.3 XPath Language Subset for Binding Hash

Our goal is to support the minimal subset of XPath that will still allow us to test structural join and value-based selection operations features of the BH Method. The XPath language contains an extensive number of keywords and features that are not supported because they are outside the scope of testing our structural join algorithm. The features we support are path expressions and filters. The path expressions allow the matching of attributes and XML element tags. Filters are sub queries that are evaluated as Boolean expressions. We describe different syntactic features that the BH Method supports from the sample XPath expression shown below.

```
/Meeting[./@ID='1']//Name
```

The above XPath expression returns all of the names that have attended the meeting whose ID attribute is 1. The first “/” is called the axis separator and indicates the start at the root of the tree. In the XPath expression “Meeting” is called the node test that indicates to search for meeting nodes from the root. The combination of an axis separator and a node test is referred to as a location step. In our grammar a navigation step involves finding either a child indicated by “/” or the more general descendant node indicated by “//”. The expression inside the brackets, called a filter contains a branch sub-query that is evaluated as a boolean expression. An XPath expression can contain multiple filter expressions that all have to be satisfied in order for the query to return a result. For example the filter “[./@ID='1']” is a sub-query which is evaluated to true based on the boolean expression inside. The “@” prefix matches an attribute with the label name ID. The equality condition inside the filter finds ID attributes with a value of 1. Within the filter expression the “./” means at the current context (analogous to the current directory in a Unix file system). The location step “//Name” specifies finding all descendants of the “Meeting” tag that are element tags labeled “Name.”
The XPath features we described for the example directly correspond to the BNF grammar syntax we show in Figure 3.2 that defines the XPath subset the BH Method supports. Our grammar below allows us to test the structural join and value-based selection that our BH Method focuses upon.

\[
\begin{align*}
\text{<XPathExpr>} &= [<\text{LocationStep}>|<\text{LocationStep}><\text{XPathExpr}>] \\
\text{<LocationStep>} &= [<\text{Axis}>]<\text{NodeTest}>|<\text{Axis}>|<\text{NodeTest}><\text{FilterExpr}>] \\
\text{<FilterExpr>} &= ['\']['<\text{boolExpr}><\text{boolConnective}><\text{boolExpr}>]' \\
\text{<boolExpr>} &= [<\text{LocationStepOptAxis}>|<\text{LocationStepOptAxis}><\text{CompOp}><\text{Constant}>] \\
\text{<LocationStepOptAxis>} &= [<\text{NodeTest}>|<\text{NodeTest}><\text{LocationStep}>] \\
\text{<CompOp>} &= '=' \\
\text{<Axis>} &= ['[/]|//@'] ['[/]/'] \\
\text{<NodeTest>} &= [(a-z)|[(a-z) (0-9)*] \\
\text{<Constant>} &= ['""[(a-z)][A-Z]0-9""]|(0-9)] \\
\text{<boolConnective}> &= "\text{AND}" \\
\end{align*}
\]

Figure 3.2: BNF Grammar for BH XPath Subset

In the next section we discuss a BH pattern tree that is the data structure that the BH XPath expression is parsed into.

3.4 Pattern and Witness Trees

In Section 2.4 we mention that a traditional database performing bulk operations on a set of homogenous data yields efficient performance [2]. A TAX pattern tree is a query plan that generates a set of witness trees that have a homogenous structure. TAX operators can perform bulk operations on a set of witness trees. The BH Method inputs a pattern tree that supports a subset of TAX algebra with an XML document and outputs a set of witness trees. In this section we formally define a TAX pattern tree and witness tree. In our discussion we describe what features of pattern trees the BH Method supports. In our pattern tree discussion we introduce a data structure called an edge binding that will be used in the BH Method.

A pattern tree contains nodes which are variables having constraints that sets of XML nodes have to satisfy. A pattern tree node is a variable that is bound to an XML node object. Once an instance of XML nodes is found that satisfies all of the constraints then a witness tree is generated. Definition 3.1 formally defines a TAX pattern tree.
Definition 3.1: A pattern tree is a pair \( P=(T, F) \), where \( T=(V, E) \) is a node-labeled and edge-labeled tree such that [4]:

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.

The pattern tree in Figure 2.8 labels the parent-child relationships as \( pc \) and the ancestor-descendant edges as \( ad \). These are the structural constraints the XML nodes need to satisfy in order to generate the witness tree. For example the left edge from Figure 2.8 specifies that a node \( 1 \) must be parent of \( 2 \). The boolean formula \( F \) adds additional constraints that are shown to the right of the pattern tree in Figure 2.8. Boolean expressions contain a set of value-based constraints against node attributes that all need to be satisfied. In TAX, the most two important attribute types are called \( tag \) and \( content \). The \( tag \) refers to the node’s label name while the \( content \) refers to its text data. Each constraint has a binary operation against two operands. The left operand contains an attribute while the right hand attribute can be either a constant or another attribute. If the right hand attribute is a constant then a value-based selection is performed otherwise if it is an attribute then a value-based join is required. The constants can be text or numerical values. Sample operators that operate on numerical data include “\( = \)”, “\( ≠ \)” and “\( >.\)”. For example, the first constraint shown in Figure 2.8 is \( 1.tag = “Meeting.” \) The \( 1 \) is a variable that refers to the node identifier while the \( tag \) specifies the node attribute. The comparison operation “\( = \)” is an equality comparison operator, while the string constant “\( Meeting \)” is the right hand operand. An “\( \& \)” is used to separate the individual constraints.

Having described a TAX pattern tree we now focus on defining the more restricted BH pattern tree. 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 3.2: The BH Pattern tree contains 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.

In Definition 3.2 the first two restrictions are relatively straightforward but the 3rd may not be obvious to the reader therefore we demonstrate it with an example. If you have two constraints 

\[ \$1.content = "Mike" & \$1.content = "Brant" \]

are not allowed because the \$1.content attribute has more then one constraint. However the statement 

\[ \$1.tag = "Name" & \$1.content = "Mike Brant" \]

is valid because each constraint refers to a unique attribute.

One important note is that any XPath expression that conforms to our XPath grammar defined in section 3.3 can be translated to the BH pattern tree. Because BH pattern trees are subsets of TAX pattern trees, we simplify our visual representation of pattern trees. In Figure 3.3 each oval denotes a pattern node to match. The text inside the oval denotes the tag name and text value to match. The text that sits outside of the brackets denotes the tag name to match while inside the brackets shown 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 edge without a “*” denotes the more specific parent-child relationship (hard edge) between two nodes.
We mention earlier that a pattern node is a variable with constraints that an XML node needs to satisfy. The set of XML nodes that satisfy the constraints are called node bindings. We now extend this definition to an edge binding which is used in the BH Method. Assume we have two pattern nodes labeled A and B where A is a descendant of B connected directly by a single edge labeled (A, B).

Definition 3.3: An edge binding returns a pair of XML nodes labeled $a$ and $b$ from the XML tree that satisfy the following constraints

1) If the query edge $(A, B)$ is a soft edge then $a$ is a descendant of $b$. If query edge $(A, B)$ is a hard edge then $a$ is a child of $b$.

2) The XML element tag name $a$ is equal to the query tag name $A$. XML Element tag name $b$ is equal to the query tag name $B$.

3) If a value is specified for query node $A$ with a comparison operator then node $a$ must satisfy the value-based constraint. The same constraints correspond for XML node $b$ and query node $B$.

For example if the operator is “=” then the “A.value = a.value” constraint must be satisfied.

Having described a pattern tree we now define how it generates a set of witness trees from an XML document. 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. In Figure 2.9, the left witness tree has a node labeled Name with the content “Michael Brant” which is maps to node $2$ in Figure 2.8. Definition 3.4 presents a formal definition of a TAX witness tree [14].
Definition 3.4: Let $C$ be a collection of data trees, $P=(T, F)$ a pattern tree. An embedding of a pattern $P$ into a collection $C$ is a total mapping $h: P \rightarrow C$ from the nodes of $T$ such that:

1) $h$ preserves the structure of $T$ whenever $(u, v)$ is a pc edge in $T$, $h(v)$ is a child of $h(u)$ and respectively for $ad$ edge.

2) The image under the mapping $h$ satisfies the formula $F$.

In our example, the pattern tree $P$ is shown in Figure 2.8 that consists of the tree $T$ and formula $F$ shown to the right. Embedding pattern $P$ to the database denoted by $C$ shown in Figure 2.6 induces a collection of witness trees shown in Figure 2.9. Function $h$ maps pattern tree nodes to XML nodes preserving the structure of $P$. Note that function $h$ takes a single pattern tree node and maps it to an XML node. Edge $(u, v)$ refers to a pattern tree edge between nodes $u$ and $v$. In our example we will set $u$ and $v$ to be $1$ and $2$ respectively. For a single witness tree in Figure 2.9 the two witness tree nodes with tag names “Person” and “Name,” which are mapped from $1$ and $2$ by $h$, must have a child-parent relationship.

An important intermediate step in the BH Method is the generation of witness trees from BH pattern trees. Because BH pattern trees are a subset of TAX trees this definition holds for generating witness trees using the BH Method.

In this chapter we have described concepts, data structures and features specific to the BH Method applied in the next chapter. We give a working demonstration of the BH Method in Appendix C.
Chapter 4

Binding Hash Method

In this chapter we introduce the BH Method that processes a subset of XPath against an XML document. The BH Method has been implemented using Microsoft C# .NET 2003. The BH Method generates a set of witness trees and returns a set of nodes satisfying the XPath expression. The BH Method is composed of seven major steps. Each section in this chapter describes and illustrates an individual step in detail by describing the data structures and algorithms along with examples. The seven steps listed are summarized below.

<table>
<thead>
<tr>
<th>Step #</th>
<th>Step Description</th>
<th>Algorithm Description</th>
<th>Example Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load the XML file into a data structure</td>
<td>Section 4.1</td>
<td>Figure 4.2</td>
</tr>
<tr>
<td>2</td>
<td>Index the XML tree to construct the BH XML Index</td>
<td>Section 4.2</td>
<td>Figure 4.3</td>
</tr>
<tr>
<td>3</td>
<td>Convert an XPath expression to a query tree</td>
<td>Section 4.3</td>
<td>Figure 4.6</td>
</tr>
<tr>
<td>4</td>
<td>Perform Post-Order traversal against the query tree producing a hashed tree structure, BindingCollection, containing only the bindings that will become part of the answer</td>
<td>Section 4.4</td>
<td>Figure 4.13</td>
</tr>
<tr>
<td>5</td>
<td>Traverse the BindingCollection and generate all root to leaf node branches</td>
<td>Section 4.5</td>
<td>Figure 4.14</td>
</tr>
<tr>
<td>6</td>
<td>Join XML branches to form witness trees</td>
<td>Section 4.6</td>
<td>Figures 4.21a &amp; b</td>
</tr>
<tr>
<td>7</td>
<td>Returns a set of nodes satisfying the XPath expression</td>
<td>Section 4.7</td>
<td>Figure 4.23</td>
</tr>
</tbody>
</table>

Figure 4.1 summarizes the BH Method’s data structures, actions and the data flow. The items outside the rectangles represent the system inputs and outputs. The two top items (represented by the document icon) outside the dashed rectangle represent the system inputs that include the XML document and XPath expression. The items at the bottom represent the outputs of the system which include the XML nodes that satisfy the XPath expression and an XML file showing the witness trees. The items inside the dashed rectangle represent the major steps and data structures belonging to the BH Method. The pentagon shaped icon represents a step while the data structures are represented by a rectangle using UML notation. Each data structure shows
the individual operations that are executed by a step that utilizes the data structure. The parameters for each operation are not shown in the diagram and will be defined when we discuss each operation in detail.
Every step needs at least one or more inputs indicated by the incoming arrow. Note that the step’s inputs must be already built before the step can be executed.

4.1 Loading the XML Document

The XML document is loaded into memory using a depth first scan of the tree. Afterwards, a data structure called the XML object tree is created. The XML tree is based on the modified ordered tree model discussed in section 3.1. Every XML tree node contains the following attributes:

1) ID: a unique identifier based on the depth first order of the node.
2) Label: the name of the element/attribute.
3) Data: the text data pertaining to the node.
4) TreeDepth: indicates how many traversal from the root it takes to reach this node.
5) NodeType: an enumerated value storing whether this node is an element or attribute.
6) Children(i): a collection of child pointers indexed by an integer i.
7) Parent: a pointer to the parent of this node.

Figure 4.2a shows a sample XML document that is used for the remainder of this chapter. The actual data in this document does not have meaning like in the previous example but the compact one-character label names will allow us to save space in the diagrams that demonstrate the algorithm. Also the irregular structure of the document will demonstrate how the BH Method works in a more complicated case. The XML object tree for the sample XML document based on the modified ordered label tree is shown in Figure 4.2b. Inside the node the text denotes the label name while the text inside the quotes represents the text data. The number just outside the node is based on the node’s depth first search order.
Figure 4.2a: Sample XML Tree

```xml
<?xml version="1.0" encoding="utf-8" ?>
<d>
  <b>
    <c />
    <c />
    <g>
      <c />
    </g>
    <d />
  </b>
  <e>
    <g>
      <d />
    </g>
  </e>
  <f>
    <e>Mike</e>
  </f>
  <f>
    <b>
      <c />
      <c />
      <d>
        <e h='Ned'>Fred</e>
      </d>
    </b>
  </f>
</d>
```

Figure 4.2b: Ordered Tree for Sample XML Document
In the BH Method we reference a unique node from the tree by simply referring to its ID attribute. However in our examples we also append the label name in front of the ID for the reader. For example, if we want to refer to the node with ID 15 we label the node as c15. Also in our sample document all node names have a lower case label name. Having loaded the XML document into memory in the form of a XML object tree, in the next section we build an index against the XML tree to allow fast retrieval of its nodes.

4.2 Retrieval of XML Nodes using the XML Index

The clustered index is created so we have quick access to a set of nodes based on labels, node tree depth and text values. The operator allowing access based on these attributes $XMLNodes(element, lev, text)$ returns an array of nodes that contain the element name $element$, have a tree depth $lev$, and containing text data $text$. Formally $XMLNodes$ operator returns a collection of nodes where each node $n$ in collection satisfies the following constraints

1) $n.Label = element,$
2) $n.TreeDepth = lev,$ and
3) $n.text = text.$

If the text value is an empty string this indicates that we want to return XML nodes that contain no text data. However if any one of the $XMLNode$ parameters is omitted then it means the constraint corresponding to that parameter does not need to be satisfied. We denote an omitted parameter by a “*.” For example $XMLNodes(“Person”,*,”Michael Brant”)$ returns a set of nodes that have a label name ‘Person’ and text value ‘Michael Brant’ but at any tree depth level because the $lev$ parameter is omitted.

The data structure used to implement the clustered index uses nested sorted lists. 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. A clustered index for the sample XML tree from Figure 4.1 is shown in Figure 4.3. The entry point to the index is a sorted list denoted by the rectangle at the top. Every sorted list contains other objects that may be a sorted lists themselves or an XML node list. The arrow denotes those containment relationships.

For example $\text{XMLNodes(“d”, 3, “”)}$ returns a set of nodes that have a label “d” at depth level 3 without any text data. In order to return those set of nodes we first access the entry point sorted list and locate entry d. Entry “d” points to another sorted list object that contains all of the possible tree depth levels for all nodes that contain the label “d.” We choose depth level 3 and arrive at another sorted list that contains all text entries with nodes labeled “d.” In this case there is one entry that contains a blank string key indicating that all of the nodes do not contain text data. Since we are looking for such nodes we access this entry and arrive at an array list of two XML nodes d7 and d16. These are the nodes that have a label “d” at depth level 3 which do not contain any text data. Alternatively, if we wanted to return the $\text{XMLNodes(“d”, *, ““)}$ then we would initially proceed as in the previous example by accessing entry “d” and arriving at the array list listing all of the depth levels for that label name. Since we do not have a constraint on the depth level we need to access all of the level entries 1, 2, 3 and 4. For each depth level since we have a constraint we check to make sure the text values are blank. At the end of the
navigation if we end up reaching multiple array lists then we will need to concatenate them to form one list. In this example the sets {d1}, {d7, d16} and {d10} are concatenated to return the list {d1, d7, d16, d10}. The general navigation algorithm demonstrated in the example is outlined in Figure 4.4.

Function XMLNodes(n, lev, text)
    Initialize XMLNodesArray to a blank array
    Variables DepthLists, textLists point to sorted lists

    If (n = *) then
        DepthLists = retrieve all sorted lists referenced by the label sorted list
    else if (Label.KeyExists(n))
        DepthLists = Label[n] //retrieve single list
    else
        return empty list
    End if

    for each DepthList in DepthLists
        If (lev = *)
            textLists = retrieve all text array list referenced by depthList
        else if( DepthList.KeyExists(lev) )
            textLists = depthList[lev] //retrieve single depth list
        else
            return empty list
        End if

        for each textList in textLists
            if (text = *)
                XMLNodesArray = concatenate all arraylists referenced by textList
            else if( textList.KeyExists(text) )
                XMLNodesArray = XMLNodesArray union textList[text]
        End for
    End Function

Figure 4.4: XMLNodes Psuedocode

The algorithm to add a new node to the XML Index is relatively straightforward and is outlined below. The key idea is traverse down the index tree and if an entry is missing in the sorted list for the node attribute add it to the list. Otherwise if the entry exists, traverse down it. After reaching the bottom of the index, add an XML node to an existing or newly created list.
Function AddXMLNodeToIndex(node)
    If Label.KeyExists(node.Label)
        DepthList = Label[node.Label]
    Else
        DepthList = Create new empty list
        Label.AddEntry(node.Label, DepthList)
    End If
    If DepthList.KeyExists(node.Depth)
        TextList = DepthList[node.Depth]
    Else
        TextList = Create new empty list
        DepthList.AddEntry(node.Depth, TextList)
    End If
    If TextList.KeyExists(node.Data)
        NodeArrayList = TextList[node.Data]
    Else
        NodeArrayList = Create new empty list
        TextList.AddEntry(node.Data, NodeArrayList)
    End if
End Function

Figure 4.5: Algorithm to update the XML Clustering Index

In this section and the previous section we focus on describing how the XML document is loaded and stored in memory. Our next task is to describing how an XPath is parsed and loaded into a BH pattern tree.

4.3 Loading XPath Into a Pattern Tree

In this section we describe how a BH XPath expression is loaded into the BH pattern tree. We first describe the BH pattern tree data structure and then give a high level description how a BH XPath expression is translated into one.

The BH pattern tree consists of pattern nodes that retain all of the attributes for an XML node described in Section 4.1. However, additional attributes are stored for a pattern tree outlined below.

BelowSoftEdge: a boolean value indicating whether a path from the root to the node \( N \) has a soft edge.

EdgeType(i): An array containing edge type information (soft or hard edge) indexed by an integer \( i \). The the edge type between node \( N \) and \( N . \text{Children}(i) \) the array is returned by \( N . \text{EdgeType}(i) \).
We briefly describe how a BH XPath expression restricted to the grammar in Figure 3.2 is translated into a BH pattern tree conforming to Definition 3.2. Within an XPath expression the node test described in Section 3.3 directly translates to a pattern tree node while the axis separator “/” or “//” is translated to a child query edge. Note that a “/” generates a soft edge while a “//” generates a hard edge. If the axis separator is the very first one not proceeded by a node test it is not translated to an edge. A filter expression representing a sub-query preceded by a node test is translated into edges pointing to subtrees which represent the filter expression. We briefly describe how different pieces of the XPath expression.

```
/d[.//e='Mike']/b[c AND ./c AND ./d]
```

are translated to a pattern tree shown in Figure 4.6. In our notation we will capitalize all of the pattern tree node labels to distinguish them from XML nodes which all use lower case 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.

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 generates 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.

![Figure 4.6: BH Pattern tree for a Given XPath Expression](image-url)
Note that an XPath expression specifies a single node Q which the XPath processor will return the XML nodes for. The last node test inside an XPath expression, which is not inside a filter, represents the node Q. For our XPath expression our Q node is B4.

### 4.4 Building the BindingCollection Index using Post-order Traversal

In this section we begin an important step of the BH Method that involves processing the BH pattern tree. This step involves retrieving the needed XML nodes from the XML tree and BH XML index. We process our BH pattern tree in post-order. During the traversal both pattern tree nodes and edges are visited. During these visits, nodes needed to satisfy the query are retrieved from the clustered index and XML tree and 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. We first discuss the structure BindingCollection. Then we give an example of one for a particular BH pattern tree and an XML document. Finally we discuss an algorithm that will be used to build the BindingCollection index.

A BindingCollection contains a multi-level index that allows quick retrieval of edge bindings for any pattern tree edge. There are two operators that are used in the BH Method: getXMLAncestors(…) and getXMLDescendants(…). The operator used in the post-order traversal is the getXMLAncestors(QueryAncID,QueryDecID) which returns a sorted list of ancestor nodes from the edge bindings that bind to a pattern tree edge. The QueryAncID refers to the query ancestor ID and QueryDecID refers to the query descendant ID. The combinations of these parameters are used to identify a query edge in the pattern tree. The second operator is getXMLDescendants(QueryAncID,QueryDecID,XMLAncestorID) which returns only the XML descendant nodes of the EdgeBindings that bind to the specified pattern edge and have XML ancestor with the XMLAncestorID. The operator in getXMLAncestors is utilized while visiting
query edges in the process of building the BindingCollection while getXMLDescendants operator will be used in steps 5 and 7 of the BH Method.

Having described the functionality of these operators we now proceed to describe their implementation. Both of these operators start navigating the BindingCollection to the edge bindings that bind to a pattern tree edge (QueryAncID, QueryDecID). In Figure 4.7 we show a partially built BindingCollection for three query pattern edges (B4,C1), (B4,C2) and (B4,D3). The entry access point of the index begins on the right side. We first access the entry point sorted list where the key values contain all of the query edge ancestors. We see that entry B4 is present and points to another sorted list containing three entries C1, C2 and D3. These are the available query descendants for the given the B4 query ancestor. We navigate to the C1 query node. Afterwards we have a sorted list whose keys are based on XML ancestor IDs that bind to the ancestor node of the query edge (B4, C1). At this stage we can navigate through all of these XML ancestor entries and retrieve all of the XML edges in this case {c3, b2} and {c4, b2} that bind to the query edge. However the getXMLAncestors(B4, C1) only returns the ancestor nodes of the XML edges so in this case {b2, b14} nodes would be returned. The other operator getXMLDescendants(B4, C1, b2) returns the descendant nodes {c3, c14} from edge bindings that bind to query edge (B4,C1) having the XML ancestor b2.
Having introduced the layout of the BindingCollection we are now ready to describe the algorithm that builds the BindingCollection index. We assume that the XML object tree in Section 4.1 and BH XML Tree index in Section 4.2 have already been built. The algorithm performs a post-order traversal against the query and visits both pattern nodes and edges. The traversal algorithm is recursive and is outlined in Figure 4.8.

\begin{verbatim}
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
\end{verbatim}

Figure 4.8: Post-order Pattern Tree Traversal

We now describe in more detail what happens during VisitNode(N) in line 6 of the traversal algorithm where \( N \) is the node to be visited. 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 the XMLNodes
operator described in Section 4.1 is used to retrieve nodes based on the following attributes of N: label, tree depth, text data. If the pattern tree edge does not specify an attribute, such as text data, then we omit the parameter. 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(..) edge operation described next. If N is a non-leaf then a structural join operation occurs. We postpone discussing the details of structural join until we describe VisitEdge(..).

VisitEdge(C, P, XMLNodeArray) visits a query edge composed of C the child node and P the parent node. Note that VisitEdge already assumes that node C already 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.

Now we describe in detail what happens when invoking VisitEdge(N) when N is a non-leaf node. At a non-leaf node there is a guarantee that all of its child nodes and edges directly below N have been visited because in a post-order traversal if we have just visited node N we are guaranteed to have visited all the nodes of the subtree rooted at N. Assume that for an edge between N and its child is referred by (N.Children(i), N) where Children is an array containing N’s children individually indexed by i. For an individual query edge (N.Children(i), N) there could exist multiple edge bindings called an edge binding group. We can represent an edge binding as a tuple having two attributes; the child and parent that can be viewed as relational
tuples. Because relational algebra operations are fairly standardized and have well-established algorithms, our strategy will be to utilize relational algebra operations to perform a structural join operation. Specifically we use a combination of relational projection and intersection operations to perform structural join shown in Figure 4.9. The \texttt{getXMLAncestors(N.ID, N.Children(i).ID)} performs the projection operation by projecting out only the ancestor node IDs from the edge bindings. Let \texttt{XMLNodesArray(N)} denote all of the XML node bindings found for query node \texttt{N}. We perform an intersection of ancestor node IDs from XML edge binding in each query edge.

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

Figure 4.9: Performing a Structural Join Using Intersection.

We implement the intersection operation using hash tables. In our implementation, 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 \texttt{SmallestList} sorted list.

\[
\min_k (\text{getXMLAncestors}(N.ID, N.Children(k).ID).\text{ListSize})
\]

\[
\text{SmallestList} = \text{getXMLAncestors}(N.ID, N.Children(k).ID)
\]

where \( 0 \leq k \leq N.\text{ChildCount} - 1 \)

Figure 4.10: Selecting a Query Edge with a Minimal Amount of Edge Bindings

We loop through \texttt{SmallestList} keys and check to make sure its ID values are found in all of the other lists. The \texttt{getXMLAncestors} operation retrieves a sorted list with the XML ID as the key. For every ID key labeled in \texttt{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 4.11 shows our implementation of the intersection operator.
Function VisitNodeNonLeaf(N)
N is non leaf returns a list of XML nodes

Initialize XMLNodes to an empty list

Find the minimum sized list SmallestList

for each XMLNode in SmallestList {
  isFound = true

  for each child in N
    curAncList = getXMLAncestors(N.ID, child.ID)
    isFound = isFound AND curAncList.KeyExists(XMLNode.ID)
    if !(isFound) then
      exit inner for loop
    End if
    if isFound then
      XMLNodes.Add(XMLNode)
    End if
  End for

Return XMLNodes
End Function

Figure 4.11: Visiting a Non-leaf Node N

At the end of PostTraversalJoin execution our BindingCollection is populated with binding edges for all of the query edges. The PostTraversalJoin returns a list of XML nodes called a RootList that stores the root node bindings. The RootList is not stored in the binding collection but is saved and utilized in the next step of the algorithm as an entry point into the BindingCollection.

Having described the Post Traversal Stage of the BH Method we illustrate how the BindingCollection index is built from our sample query in Figure 4.6 and XML tree from Figure 4.2. In Figure 4.10 the BindingCollection is graphically illustrated. When performing a post-order traversal we first visit query node C1 by Visit(C1). Because query node C1 is a leaf node we will use the XMLNodes operator to retrieve XML nodes that bind to query node C1. For C1 we invoke XMLNodes (c, 2, *) where c is the label name, 3 is the depth level and * indicates to ignore matching the text constraint. XMLNodes operator returns {c3, c4, c6, c14, c15} nodes that
are temporarily stored in the `XMLNodeArray` variable from Figure 4.11. For each node in `XMLNodeArray` we attempt to retrieve its parent node using its parent pointer from the XML tree. The parent nodes \{b2, b2, g5, b13, b13\} are retrieved. We check to make sure these nodes bind to the parent node C2 of the edge. All of the parent nodes bind to C2 except g5 because the label name does not match b2 hence we discard g5. We add the edge bindings \{c3-b2, c4-b2, c13-b12, c14-b13\} to the binding collection. Figure 4.7 shows the added edges under the `QueryEdge(C1, B4)` container where the left node is the descendant and right node is the ancestor. Note that we discard binding \{c6-g5\} indicated in the diagram. Next we visit query node C2 and use `XMLNodes(c, *, *)` to retrieve nodes labeled c at any depth level without matching any text constraint and retrieve XML nodes \{c3, c4, c6, c14, c15\} which bind to C2. The reason we retrieve the XML nodes using any depth is because node C2 is below a soft edge meaning that it can be at any depth level. Afterwards, we invoke `VisitEdge(C2, B4,\{c3, c4, c6, c14, c15\})`. Because \((C2, B4)\) is a soft edge we scan all of the ancestors using parent pointer of every XML node and if the ancestor node binds to B4 we add the edge binding. For all edge bindings with descendant node c6 the only ancestor that binds to B4 is b2. As a result we will add \{c6-b2\} to the `BindingCollection`. At the end of visiting edge \((C2, B4)\) we add \{c3-b2, c4-b2, c6-b2, c13-b12, c14-b13\} to the `BindingCollection` shown under the `QueryEdge(C2, B4)` container. Similarly, we visit node D3 and query edge \((D3, B4)\) which results having the edge bindings \{d7-b2, d16-b13\} added to the `BindingCollection`. Next we invoke `VisitNode(B4)` and because it is a non-leaf node we perform structural join operation using intersection and projection outlined in Figure 4.9. For \(B4\) there are three query edges \((C1, B4), (C2, B4)\) and \((D3, B4)\) for which we will retrieve the edge bindings. Each `getXMLAncestors` operator projects the ancestor ID portion of each XML node. Then the three lists are intersected and the resulting XML nodes after the intersection are \{b2, b13\} that bind to query node B4. We summarize the mathematical operations used to perform a structural join during `VisitNode(B4)` in Figure 4.12.
XMLNodesArray(B4) = getXMLAncestors(C1, B4) ∩
getXMLAncestors(C2, B4) ∩
getXMLAncestors(D3, B4)

XMLNodesArray(B4) = \{b2, b2, b13, b13\} ∩
\{b2, b2, b2, b13, b13\} ∩
\{b2, b13\} = \{b2, b13\}

Figure 4.12: Structural Join During VisitNode(B4)

The algorithm continues the post-order traversal until it visits the root node. Since the
root node is non-leaf a structural join is performed and \{d1\} is the result. In this case the root list
RootList = \{d1\} which is saved and will be used as the entry point into the BindingCollection in
the next stage of the algorithm. Figure 4.13 shows the RootList on the right.
The Post-Traversal step of the algorithm is an important step in the BH Method that builds the *BindingCollection* index bottom up from the leaf nodes to the root. In the next step when we traverse the *BindingCollection* top down. During the traversal, every node not satisfying all of the constraints specified in the pattern tree is eliminated.

### 4.5 Generating Branches Using Pre-order Traversal

The *BindingCollection* index in the previous step allows us to retrieve bindings that are be used to construct witness trees. Before generating witness tree in this step we generate the
data structure called BranchCollection containing the entire possible root to leaf XML branches built from the BindingCollection. While constructing the BranchCollection we traverse the query using pre-order traversal. During the node visitation we access the BindingCollection allowing us to quickly retrieve the XML node and edge bindings that will be used to retrieve XML branches to be added to the BranchCollection. In this section we first describe the layout of the BranchCollection data structure and show an example of it for a sample query tree. Then we describe a pre-order traversal algorithm that is used to build the BranchCollection.

We now define the definition of a root to leaf branch within a pattern tree. A root to leaf branch is a tree 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 amount 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 4.6 contains 4 leaf nodes each identifying root to leaf branches that are {D6-B4-C1, D6-B4-C2, D6-B4-D3, D6-B5}. For example, if we wanted to uniquely reference the second branch we refer to it by the leaf ID, in this case 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. The BranchCollection structure we built in the previous allows retrieval of XML branches for a single root to leaf query 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 4.14 shows a visual representation of the BranchCollection for a pattern tree in Figure 4.6 generated from the BindingCollection in Figure 4.13. Each oval represents a pattern tree node. Below each pattern leaf node denoted by L a set of XML branches that bind from the root R to L path 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.

When we traverse the BindingCollection branches, partial matches are not accessed because the paths to reach them were eliminated using structural join operation during the construction of the BindingCollection.

We now describe the algorithm that builds the BranchCollection index from the BindingCollection index. The pseudocode for the algorithm is shown in Figure 4.15b. The key idea behind the algorithm is to explore all of the branch paths from BindingCollection and add them to the BranchCollection. We do so by traversing the BranchCollection in pre-order starting at the root and traversing it until we reach a leaf node. During the traversal we store the current path from the root to the current node in the stack labeled XMLBranchPathStack. Once we reach a leaf XML node the contents of XMLBranchPathStack contains a root to leaf XML branch and we add it to the BranchCollection. During the traversal we use the getXMLDescendants operator (described in Section 4.4) to navigate down the BindingCollection index. Recall that from the previous step the QueryPostTraversal routine that builds the RootList is used as an entry point.
into the BindingCollection. For every RootList item we invoke QueryPreOrderTraversal to retrieve all of the root to leaf branch paths for each root node. The invocation of QueryPreOrderTraversal is summarized in the pseudocode in Figure 4.15a.

```java
// perform step building the BindingCollection Index
RootList = QueryPostTraversal(QueryRootNode)

// build the BranchCollection index
for each XMLRootNode in RootList
    XMLPathStack = allocate empty list
    XMLPathStack.add(XMLRootNode)
    QueryPreTraversal(QueryRootNodeID, XMLRootNode, XMLPathStack)
end for

Figure 4.15a: Invocation of QueryPreOrderTraversal

QueryPreOrderTraversal(QueryAncNode, XMLAnc, XMLBranchPathStack) {
    for each objQueryDecNode in QueryAncNode.Children
        XMLDecArray = BindCollection.getXMLDescendants(
            objQueryAncNode.ID, objQueryDecNode.ID,
            objXMLAnc.ID)
        for each XMLDecNode in XMLDecArray {
            XMLBranchPathStack.Push(XMLDecNode)
            QueryPreOrderTraversal(objQueryDecNode, objXMLDecNode,
                XMLBranchPathStack)
        }
    if objQueryAncNode is a leaf Node {
        AddLeafBranch(objQueryAncNode.ID, XMLDecNode);
    }
    XMLBranchPathStack.Pop()
}

Figure 4.15b: QueryPreOrderTraversal Routine

Having described how the BranchCollection is built we now trace through an example demonstrating how the first two root to leaf branches are added to the BranchCollection in Figure 4.14. Recall that after we invoke the post-order traversal we build the BindingCollection shown in Figure 4.13 and were left with the RootList that contains a single XML root node ID of d1. The XML root node d1 binds to the root query node D6 and serves as our starting point for navigating the BindingCollection index. Hence d1 is pushed on to the XMLBranchPathStack first and is used to keep our current path from the root. For the pattern root node at D6 we visit the
child edges. In this case $D6$ has two children $B4$ and $E5$. We first navigate to the $(B4, D6)$ edge. While visiting this edge we invoke $getXMLDescendants(B4, D6, d1)$ which will return the descendant portion of the edge bindings which bind to edge $(B4, D6)$ and have $d1$ as the XML ancestor edge. Those edge bindings are visually shown in Figure 4.13 under the $QueryEdge(B4, D6)$ container symbol. Specifically $getXMLDescendants$ returns a list of two XML nodes $\{b2, b13\}$ that bind to the query node $B4$. We now inspect the children of $B4$ and realize there are three child nodes $C1, C2$ and $D3$. Because we are using a depth first scan we pick the first child edge $(C1, B4)$ and choose $b2$ as the XML node that binds to $B4$. We push the XML node to the $XMLBranchPathStack$ that now contains the path $d1-b2$. Then we invoke $getXMLDescendants(B4, D6, b1)$ which returns all XML edges that bind to edge $(C1, B4)$ and contains $b1$ as the XML ancestor edge. Two XML nodes are returned $c3$ and $c4$. We now arrive at node $C1$ and choose $c3$ as the XML binding add push onto the XMLBranchPathStack which now contains nodes $d1-b2-c3$. Because $C1$ is a leaf node and the XMLBranchPathStack contains a root to leaf XML branch. We add this path to the $BranchCollection$ under the key index $C1$ using the AddLeafBranch function. Then we pop off the top element from the XMLBranchPathStack. We are still at node $C1$ and choose node $c4$ to visit and push to the XMLBranchPathStack. Realizing we are at a leaf node $C1$ we add the current contents of XMLBranchStack $d1-b2-c4$ to the $BranchCollection$ index and pop $c4$ off the stack.

In this section we focused on how to build the $BranchCollection$ from the $BindingCollection$ we built in section 4.4. In the next step we use the built $BranchCollection$ to generate a set of witness trees.

### 4.6 Generating Witness Trees from XML Branches

In the previous section we generate the $BranchCollection$ containing root to leaf XML branches that bind to a root-to-leaf pattern branch identified by the leaf query node ID. In this section we describe an algorithm that generates witness trees by joining together XML branches from the $BranchCollection$. We begin this section by describing how to access XML branches
from the BranchCollection. Then we describe an algorithm that chooses which XML branches are used to form a valid witness tree. Finally we describe how the XML branches are actually stitched together to form the witness trees.

XML root-to-leaf branches that bind to a single pattern tree branch are called a leaf group. A BranchCollection is a sorted list containing leaf group elements that can either be accessed by the leaf node ID or by an index. The getXMLBranchByIndex(i, j) operator accesses an individual XML branch by specifying the leaf group index i and the individual XML branch at the jth position. For example using Figure 4.14 if we invoke getXMLBranchByIndex(4,2), the 2nd XML branch is chosen from the 4th leaf group in this case d1-e12. The call to getXMLBranchByID(ID,j) accesses the leaf group by its ID and the individual XML branch at the jth position. For example getXMLBranchByIndex(E5, 2) retrieves XML branch d1-e12.

Having defined the basic operation which retrieves an XML branch using the BranchCollection index, we now focus on how to utilize these operations to choose XML branches that are part of witness trees.

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 then we can join (or stitch) them together to form a valid witness tree. The algorithm called FindWitnessBranches is responsible for choosing the witness branches to form witness trees. During the branch selection process FindWitnessBranches utilizes an important operation known as a compatibility check that can determine whether a pair of XML branches can be stitched together to become part of a valid witness tree. We begin discussing the general idea behind FindWitnessBranches. Then we describe in detail how a compatibility check works for a pair of XML branches. One important note is that any time during the branch selection process if a leaf group is empty then we cannot generate any valid witness trees.

The first step in the algorithm involves retrieving the first XML branch from the first leaf group by invoking getXMLBranchByIndex(0 ,0) and if it exists push it on the stack. We then choose an
XML branch from the next leaf group. Whenever we navigate to an XML branch that has not yet been added to XMLBranchesStack, we call it a candidate witness branch denoted by XMLBranch. We first check to see if the candidate witness branch can be used to form a valid witness tree with the current branches found in XMLBranchesStack. For every XML branch currently in XMLBranchesStack a compatibility check is performed against the candidate witness branch. If the candidate XML branch is compatible with all the current XML branches in XMLBranchesStack, we push it onto the stack and choose a witness branch from the next leaf group. If the candidate branch is found to be incompatible with at least one of the branches, then this branch is discarded and the next leaf branch is chosen from the current leaf group. 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. The FindWitnessBranches algorithm is summarized in Figure 4.16.

FindWitnessBranches(currLeafNodeIndex, XMLBranchesStack)
{
    for(XMLBranchIndex=0 to Number of branches in query group)
    {
        currLeafNodeID = IndexToQueryID(currLeafNodeIndex)
        XMLBranch = getXMLBranchByIndex(currLeafNodeID,XMLBranchIndex)
        if (XMLBranch is compatible with all other branches in XMLBranchStack)
        {
            XMLBranchesStack.Push(XMLBranch);
            if(currLeafNodeQueryIndex < getQueryLeafCount()-1) {
                FindWitnessBranches(currLeafNodeQueryIndex+1,
                XMLBranches)
            } else
                stitch all the branches together in XMLBranch
                to form a witness tree using AddWitness() operation
                XMLBranchesStack.Pop(XMLBranches)
        }
    }
}

Invoked with: FindWitnessBranches(0, XMLBranchesStack)
where XMLBranchesStack is empty

Figure 4.16: Finding Witness Branches in BranchCollection
Figure 4.17 shows a trace of steps performed in FindWitnessBranches for the BranchCollection index in Figure 4.13. Although the algorithm internally uses getXMLBranchByIndex (...) in the diagram, for reader clarity, we label each branch by using getXMLBranchByID(ID, j) operator using the shorthand syntax <ID, j>. For example <C2, 1> retrieves the 1st leaf branch from the leaf group C2. Recall that ID is a pattern leaf node ID identifying the leaf group and j is an index specifying the specific XML branch within the leaf group. The nodes within the same smooth edged rectangle all belong to the same query node ID indicated by the bold label inside. The bold vertical separators are used to separate XML branches belonging to different leaf groups. Note that this example only shows a trace of the four steps of FindWitnessBranches. Our example ends after we found the XML branches d1-b2-c3, d1-b2-c3, d1-b2-d7, and d1-e8, which are stitched together to form a witness tree. The discussion of the stitching operation appears later in this section.

Figure 4.17: FindWitnessBranches Trace

We now focus on describing how a compatibility check determines whether two XML branches labeled branch1 and branch2 can be used to construct a witness tree. In Section 4.5 we mentioned that a root to leaf XML branch is itself a sorted list that uses a query ID for its keys. During a compatibility check we need to access an XML node’s ID and the pattern node ID the
XML node binds to from a root-to-leaf XML branch. For a root-to-leaf XML branch we can access an XML node’s ID for an index $i$ by $branch[i].ID$ where $ID$ is an attribute uniquely identifying the XML node. To access the query ID which the XML node binds to indexed by $i$ we do so by the $branch.getKey(i)$ operation for a sorted list defined in Section 3.2. For compactness we represent the retrieval of the query node ID for an index by $QueryID(branch[i])$.

Definition 4.1: Two branches $A$ and $B$ are said to be compatible if they satisfy the following condition.

$$QueryID(A[i]) = QueryID(B[i]) \implies A[i].ID = B[i].ID$$

Where $i$ is an integer: $i \geq 0$, $i \leq A.Count-1$, $i \leq B.Count-1$

The operation that checks the above condition is denoted by $CompatibilityCheck(A,B)$ where $A$ and $B$ are the two branches to be checked for compatibility. The compatibility check ensures that we avoid stitching branches together which are incompatible.

The algorithm of the operation $CompatibilityCheck(A,B)$ is outlined below

1) Let $node1$ be a pointer to an XML branch $A$ and $node2$ be a pointer to an XML node in branch $B$

2) Let $node1$ start at the root of branch $A$ and $node2$ start at the root of branch $B$

3) If both XML nodes $node1$ and $node2$ belong to the same query node they must have identical XML IDs. If they do not, then the compatibility check fails and is finished.

4) Attempt to traverse to the child nodes for $node1$ and $node2$ along the corresponding branches $A$ and $B$. If such a node exists, update $node1$ and $node2$ pointers to the child nodes and then go to step 3. If either of those children do not exist, then the compatibility check is finished and considered passed.

In order to gain a better understanding of the compatibility operation we provide two examples in Figures 4.18a and 4.18b that traces its steps. Figure 4.18a shows an example where the compatibility check fails for the two XML branches $d1-b2-c3$ and $d1-b13-c3$ which means these two branches can never be used to form a witness tree. Figure 4.18b shows an example
where a compatibility check passes for the XML branches \(d1-b2-c3\) and \(d1-e5\) which means these two branches can later be used to form a witness tree. Note that the branches are drawn horizontally in the diagram. Each rectangle indicates a node, with its XML ID inside and at the upper left corner the query ID the XML node binds to.

![Compatibility Check Trace for a Failing Case](image)

**Figure 4.18a: Compatibility Check Trace for a Failing Case**

Once a set of root to leaf XML branches have been chosen from every leaf group and are found to be compatible with each other we need to stitch these branches together to form a witness tree. The found witness trees are added to an array list called the *WitnessCollection*. We now discuss the data structure that stores a witness tree and then how it is created.

![Compatibility Check Trace for a Passing Case](image)

**Figure 4.18b: Compatibility Check Trace for a Passing Case**
The data structure used to store an individual witness tree is a hash table whose keys are based on the query node ID containing XML nodes. When \textit{queryID} is used as the key, the entry returned from the witness tree is the XML node \textit{n} that binds to the pattern tree node \textit{q} identified by \textit{QueryID}. Using the terminology from the TAX definition of a witness tree from Section 3.4, the query node \textit{q} embeds the XML node \textit{n}.

The operation that builds a single witness tree hash table and adds it to the \textit{WitnessCollection} is called \textit{AddWitness} outlined in Figure 4.19 and is invoked within the \textit{FindWitnessBranches} routine. \textit{AddWitness} input is a list of XML branches called \textit{WitnessTreeBranches} that form a valid witness tree. The goal is to convert the list of XML branches to a witness tree stored in a hash table and add it to the \textit{WitnessCollection} array. This can be accomplished by traversing every node \textit{n} of all the XML witness branches. For each node \textit{n} we check to see if its query ID is in the hash table and if it is not present we add it to the hash table called \textit{WitnessTree}. Once we finish scanning the branches and build our \textit{WitnessTree} hash table, we add this witness tree hash table to an array list called \textit{WitnessCollection} that contains all of the currently found witness trees. The steps for \textit{AddWitness} are outlined in Figure 4.19.

\begin{verbatim}
Function AddWitness(WitnessTreeBranches)
    Initialize empty WitnessTree hash table
    for each XMLBranch in WitnessTreeBranches
        for i=0 to XMLBranch.Count-1
            if (WitnessTree.ContainsKey( QueryID(XMLBranch[i]) )
                WitnessCollection.Add(QueryID, XMLBranch[i])
            end if
        next
    next
End Function
\end{verbatim}

Figure 4.19: \textit{AddWitness} Operation

We now show a graphical example of the \textit{AddWitness} operation. Recall our example from Figure 4.17 where at the end of Step 4 we have chosen an XML branch from every leaf group d1-b2-c3, d1-b2-c4, d1-b2-d7, and d1-e8. We now demonstrate how these branches are stitched together as step 5 in Figure 4.20. On the right side of the diagram we show root-to-leaf XML branches that are to be stitched together. Following the conventions from Figure 4.17 the
nodes inside a same smooth edged rectangle all belong to the same query node ID labelled by the bold label inside. The QueryID function defined earlier in this section is used to retrieve this information. On the right side the resulting witness tree is shown. The smooth edged rectangles show the individual query node while the sharp edged rectangles inside represent the XML node that binds to the query node.

Figure 4.20: Constructing a Witness Tree from XML Branches

In this section we select the root-to-leaf XML branches and stitch them together to form a set of witness trees that are stored in the table-like structure WitnessCollection. For our example there are many witness trees added to the WitnessCollection and in order to save space we only show three of them in Figure 4.21a. A WitnessCollection can be viewed as a data table shown in Figure 4.21b where the rows represent the witness tree index while the query node ID’s.

XPath expressions specify a single pattern node for which to return a set of XML bindings. In the next section we perform an operation that returns a set of XML nodes that bind to a single pattern node.
Our XPath expressions implicitly specify a single pattern node for which to return a set of XML bindings. In the next section we perform an operation that returns a set of XML that bind to a single pattern node.

### 4.7 Generating Nodes to Satisfy the XPath Expression

In Section 4.6 the witness trees where added to the \textit{WitnessCollection}. If we are embedding an XPath expression within a more complex language such as Xquery and XSLT then the information stored \textit{WitnessCollection} can be used to perform more complex operations by the query processor. However, if we are processing only a single XPath expression then we can retrieve a set of XML nodes directly from the \textit{BindingCollection} that bind to a single query node specified in the XPath expression. Therefore the generation of the \textit{BranchCollection} in Section 4.5 and witness tree generation from Section 4.6 steps are not needed. In this section we introduce a method that accesses the \textit{BindingCollection} to retrieve a set of XML nodes that satisfy an XPath expression.
Recall from Section 4.3 that an XPath expression specifies a node \( Q \) for which to retrieve XML nodes. In our implementation we first construct a path from the query root \( R \) to \( Q \). One way to construct this path is to start at \( Q \) and traverse up the pattern tree until \( R \) is found. For every node we visit along the traversal we push it onto stack \( S \). Once we reach query node \( R \) we begin popping stack \( S \) that contains a path from \( R \) to \( Q \).

We now describe the algorithm that navigates the \emph{BindingCollection} data structure using the path stored in stack \( S \), and returns a set of nodes that bind to \( Q \) in the pattern tree. The algorithm starts at the root and visits all of the query edges until it reaches \( Q \). The idea behind the algorithm is for every query edge visited we retrieve a set of XML edge bindings from the \emph{BindingCollection}. If the query edge \((A, B)\) is the first one visited then node \( A \) is the root. We use the root list from the \emph{BindingCollection} to retrieve the ancestor nodes that bind to query node \( A \). To retrieve the bindings for \( B \) we invoke \texttt{getXMLDescendant}(A, B, XMLAncestor) for every XML ancestor and union the results together. Then for the next query edge \((B, C)\) we use the XML bindings we find in \( B \) to bind to \( C \). Once the final edge is visited the last set of XML nodes retrieved by the \texttt{getXMLAncestors(..)} operator are the XML nodes that bind to \( Q \) and satisfy the XPath expression. We write those nodes to an XML file. The algorithm is outlined in Figure 4.22.
GenerateXPathNodes(Q) {
    CurrNode=Q

    //construct the path and store it in S
    S.Push(CurrNode)
    while (CurrNode.IsRoot)
    {
        CurNode = CurNode.getParentNode();
        S.Push(CurNode)
    }

    //traverse down S
    XMLAncestors = BindingCollection.RootList()
    ParentQueryNode = S.Pop()
    While (S is not Empty) {
        ChildQueryNode = S.Pop()
        for each XMLAncestor in XMLAncestors {
            XMLDescendants.AddNodes(
                BindingCollection.getXMLDescendants(ParentQueryNode.ID,
                ChildQueryNode.ID,
                XMLAncestor.ID)
            )
        }
        XMLAncestors = XMLDescendants
        ChildQueryNode = ParentQueryNode
    }

    for each XMLNode in XMLAncestors
    {
        WriteNodeToFile(XMLNode)
    }
}

Figure 4.22: XPath Node Generation

Having described the algorithm, we trace an example of GenerateXPath which uses the BindingCollection from Figure 4.13 and the pattern tree from Figure 4.6. Recall from Section 4.3 that the pattern node of interest Q is B4. Our first goal is to find a path from the root to B4 and we do so by starting at B4 and then traversing up the parent pointers until we reach the root node pushing the nodes visited along the way onto the stack S. In the query tree we first visit B4 pushing it onto the stack and after traversing its parent pointer we already arrive at the root node D6. Afterwards we visit node D6 and push it onto the stack S that has the contents {D6, B4}. 

65
Having constructed the path from the root node D6 to the node of interest B4 we know traverse down visiting all edges between D6 and B4. In this example case there is only one edge to visit \( \{D6, B4\} \). We retrieve our bindings for the root query node D6 by accessing the root list that contains only one XML node d1. We then invoke \( \text{getXMLDescendants}(D6, B4, d1) \) which retrieve the descendant nodes \( \{b2, b13\} \) that bind to node B4 and contain the XML node ancestor d1. After obtaining the binding for B4 we output them to a file in XML format. Figure 4.23 shows the resulting file that contains the output of nodes b2 and b13.

```xml
<b></b>
<b></b>
```

Figure 4.23: Results Returned by the Query Processor

In this chapter we have described the entire BH Method which we implemented. In the next chapter we experimentally analyze the scalability for the the steps with the most overhead.
Chapter 5

Comparison and Evaluation

In the previous chapter we described the major steps involved in the BH Method. We implemented the major steps of the BH system using Windows in C# using the Microsoft .NET 1.1 framework. In this chapter we conduct experiments that give an idea how well these steps scale. In our experiments we focus on evaluating the performance of the most computationally intensive steps that are summarized in the table below.

<table>
<thead>
<tr>
<th>Experiment Number</th>
<th>Binding Hash Steps</th>
<th>Description</th>
<th>Compared against</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td>Measure the loading and indexing times for an XML document as the number of XML nodes increases</td>
<td>Microsoft</td>
<td>5.1</td>
</tr>
<tr>
<td>2</td>
<td>4, 5, 6</td>
<td>Vary the pattern tree leaf count to assess impact of witness tree generation</td>
<td>Timber</td>
<td>5.2</td>
</tr>
<tr>
<td>3</td>
<td>4, 5, 6</td>
<td>Vary the pattern tree depth level to assess the impact of witness tree generation</td>
<td>Timber</td>
<td>5.2</td>
</tr>
<tr>
<td>4</td>
<td>4, 7</td>
<td>Vary the pattern tree leaf count to assess the impact of processing an XPath expression</td>
<td>Saxon</td>
<td>5.3</td>
</tr>
<tr>
<td>5</td>
<td>4, 7</td>
<td>Vary the pattern tree depth level to assess the impact of processing an XPath expression</td>
<td>Saxon</td>
<td>5.3</td>
</tr>
</tbody>
</table>

Throughout our experiments we use the xmlgen utility developed for the XMark benchmark [19] to generate XML documents that model an auction e-commerce website. The xmlgen utility accepts a single parameter called the scaling factor that determines the document’s output size. In our timing experiments we conduct each experiment three times and in our results show the average of three experimental runs.
5.1 XML Index

In this experiment we seek to get an idea of the runtime scalability of loading the XML document into the BH XML Tree and BH XML Index while increasing the XML document size. We compare the scalability against Microsoft’s XML indexing engine (XPathNavigator) that is packaged with the Microsoft.NET 1.1. The computer is an Intel Pentium 4 3.6 Ghz CPU with 3GB of memory. In this experiment we use the xmlgen utility to generate our XML documents and vary their size by adjusting the size-scaling factor from .05 to 1. The runtime data we obtain when running the BH and Microsoft indexing is shown in Figure 5.1 and plotted in Figure 5.2. The number of XML nodes includes the total number of element tags, attributes and text values the XML document has.

<table>
<thead>
<tr>
<th>Number of XML nodes</th>
<th>Xmark Scaling Factor</th>
<th>File Size (KB)</th>
<th>BH XML Document Loading and Indexing Time (s) (Steps 1 and 2)</th>
<th>Microsoft XML Loading and Indexing Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>161520</td>
<td>0.05</td>
<td>5736</td>
<td>1.231</td>
<td>0.70</td>
</tr>
<tr>
<td>324273</td>
<td>0.10</td>
<td>11597</td>
<td>3.605</td>
<td>1.23</td>
</tr>
<tr>
<td>650335</td>
<td>0.20</td>
<td>23365</td>
<td>6.050</td>
<td>2.70</td>
</tr>
<tr>
<td>969617</td>
<td>0.40</td>
<td>34866</td>
<td>9.914</td>
<td>5.10</td>
</tr>
<tr>
<td>1290249</td>
<td>0.50</td>
<td>57641</td>
<td>13.480</td>
<td>6.21</td>
</tr>
<tr>
<td>1940765</td>
<td>0.60</td>
<td>69892</td>
<td>23.333</td>
<td>8.31</td>
</tr>
<tr>
<td>2585984</td>
<td>0.80</td>
<td>92975</td>
<td>30.120</td>
<td>10.50</td>
</tr>
<tr>
<td>3221925</td>
<td>1.00</td>
<td>115775</td>
<td>37.891</td>
<td>13.00</td>
</tr>
</tbody>
</table>

Figure 5.1: Loading the XML document

In this experiment we are interested in the loading and indexing time of BH and Microsoft’s XPathNavigator as the size of the XML document grows. Based on our regression in Figure 5.2 the BH load time increases linearly with the document size at a rate of .0003 s/KB while the Microsoft’s load time increases at a linear rate of 0.0001 s/KB. While both systems scale linearly, Microsoft’s indexing method scales better as the document size increases. From these results we conclude that the implementation of our BH Index requires further optimization.
5.2 Witness Tree Generation Scalability

For an XPath expression, Steps 5 and 6 of the BH Method return a set of witness trees that bind to the entire pattern tree that can be used in a system that embeds XPath expressions. In this section we focus on determining how well our system scales generating witness trees for various pattern trees. We also simultaneously compare its scalability against Timber [13] with the implementation that uses the more recent annotated pattern tree implementation [14].

Recall that Timber uses Xquery language to construct sets of pattern trees while our BH Method uses an XPath expression to construct a pattern tree. In the BH Method, Steps 5 and 6 generate a set of witness trees that bind to a pattern tree P. In order to compare how efficiently the BH Method generates witness trees compared against Timber, in Timber we construct an Xquery statement that equivalently generates a set of witness trees for a pattern tree P.

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. In this section we conduct two experiments both
of which incrementally add nodes to a particular pattern tree. For both experiments in this section the BH system and Timber are run on the same hardware setup which is a 1.333Ghz Athlon Thunderbird CPU and 640MB DDR RAM with Windows XP SP1. 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 this experiment we are interested in comparing the runtime overhead of the witness tree generation between the two systems while incrementally increasing the number of leaf nodes. In Figure 5.3 we show all of the XPath expressions used in Experiment 2 containing various leaf counts. Figure 5.4 shows a BH pattern tree P that is generated for the XPath expressions in Figure 5.3 with a leaf count of 6. In this experiment we also use 6 different Xquery expressions processed by Timber that generates witness trees. We show one of those Xquery expressions in Figure 5.5 that performs the equivalent action of generating a set of witness for the pattern tree shown in Figure 5.4. The rest of the pattern tree diagrams and Xquery expressions can be found in Appending A.

<table>
<thead>
<tr>
<th>Pattern Tree Leaf count</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/site/regions//item[location]</td>
</tr>
<tr>
<td>2</td>
<td>/site/regions//item[location and quantity]</td>
</tr>
<tr>
<td>3</td>
<td>/site/regions//item[location and quantity and name]</td>
</tr>
<tr>
<td>4</td>
<td>/site/regions//item[location and quantity and name and payment]</td>
</tr>
<tr>
<td>5</td>
<td>/site/regions//item[location and quantity and name and payment and description]</td>
</tr>
<tr>
<td>6</td>
<td>/site/regions//item[location and quantity and name and payment and description and shipping]</td>
</tr>
</tbody>
</table>

Figure 5.3: XPath Expressions Used for Experiment 2
The runtime data for the first experiment is shown in Figure 5.1 where we incrementally add leaf nodes while keeping the pattern tree depth at a constant value of 4.
Figure 5.6: Experiment 2 Witness Tree Generation Runtime (Seconds)

In order to get an idea of the overhead we plot the witness generation times of the two systems on the same graph, shown in Figure 5.6. We perform a linear regression to calculate the runtime overhead in 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 can conclude that Timber has better scalability as the number of leaf nodes in a pattern tree increases.

![Figure 5.4: Experiment 2 Witness Tree Generation Time Regression](image)

In the second experiment we incrementally increase the depth level of the query while keeping the leaf node count constant at 1. Figure 5.6 shows five XPath expressions used in the experiment which translate to the pattern trees shown in Figure 5.7. Note that in this experiment...
we generate 5 different Xquery expressions where each of them returns a set of witness trees for the five pattern trees. Figure 5.8 shows one of the Xquery expressions that generate a set of witness trees against the pattern tree in Figure 5.7 that contains a depth level of 5. All of the XQuery expressions used in this experiment can be found in Appending B.

<table>
<thead>
<tr>
<th>Depth level</th>
<th>XPath Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/site</td>
</tr>
<tr>
<td>2</td>
<td>/site/regions</td>
</tr>
<tr>
<td>3</td>
<td>/site/regions/item</td>
</tr>
<tr>
<td>4</td>
<td>/site/regions/item/description</td>
</tr>
<tr>
<td>5</td>
<td>/site/regions/item/description/text</td>
</tr>
</tbody>
</table>

Figure 5.6: XPath Expressions Used for Experiment 3

Figure 5.7: Pattern Trees Used for Experiment 3
Our goal in this experiment is to compare the runtime overhead between the two systems while incrementally adding a new node that increases the depth level of the pattern tree. We do so by plotting the BH and Timber Witness Generation Tree runtime in Figure 5.6 and perform a linear regression for the two systems.

<table>
<thead>
<tr>
<th>Pattern Tree DepthLevel</th>
<th>Generation of the BindingCollection Index (Step 4)</th>
<th>BH Witness Tree Generation (Steps 5 and 6)</th>
<th>Total Witness tree Generation Time (Steps 4, 5, and 6)</th>
<th>Timber Witness Tree Generation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.481</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.050</td>
</tr>
<tr>
<td>3</td>
<td>0.01</td>
<td>0.020</td>
<td>0.030</td>
<td>2.110</td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>0.023</td>
<td>0.043</td>
<td>2.730</td>
</tr>
<tr>
<td>5</td>
<td>0.03</td>
<td>0.024</td>
<td>0.054</td>
<td>3.210</td>
</tr>
</tbody>
</table>

Figure 5.9: Experiment 2 Witness Tree Generation Runtime (seconds)

According to the regression adding a node, which increases 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 can conclude the BH Method has better scalability as the query depth level increases.
Figure 5.10: Experiment 3 Witness Tree Generation Time Regression

Our general conclusions from these experiments is the BH Method performs well generating witness trees when the query tree is deep and non-bushy while Timber performs well when the query is bushy.

5.3 XPath Processing Scalability

In the BH Method Steps 4 and 7 return a set of nodes that satisfies an XPath expression. In the BH Method, an XPath expression is translated to a pattern tree and when processed a set of XML nodes are returned that bind to a single pattern tree node. In this section we perform experiments that focus on determining how well our system scales when processing XPath expressions that translate to various pattern tree shapes. We also simultaneously compare the XPath processing performance against Saxon’s [18] XPath processor. In our experiments we used the 8.7.1 open source version of Saxon running on the Java platform. In section 5.2, we measured the performance of Steps 4, 5, and 6 that generated witness trees by adjusting the pattern tree depth level and leaf node count. We take a similar approach in this section but instead measuring XPath expression processing time for various pattern tree depth level and leaf node counts.

In Experiment 4 we measure the XPath performance by adjusting the pattern tree depth leaf count. The XPath expressions in Figure 5.3, which are used to generate the pattern trees in
Experiment 2 are also used in this experiment. The timing results for XPath processing in BH (Steps 4 and 7) and Saxon are shown in Figure 5.11.

<table>
<thead>
<tr>
<th>Number of Leaf Branches</th>
<th>Binding Collection (Step 4)</th>
<th>XML Node Retrieval (Step 7)</th>
<th>BH Method XPath Time (Steps 4 and 7)</th>
<th>Saxon XPath Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.02</td>
<td>0</td>
<td>0.02</td>
<td>0.112</td>
</tr>
<tr>
<td>2</td>
<td>0.03</td>
<td>0.01</td>
<td>0.04</td>
<td>0.115</td>
</tr>
<tr>
<td>3</td>
<td>0.04</td>
<td>0.01</td>
<td>0.05</td>
<td>0.141</td>
</tr>
<tr>
<td>4</td>
<td>0.06</td>
<td>0.01</td>
<td>0.07</td>
<td>0.145</td>
</tr>
<tr>
<td>5</td>
<td>0.07</td>
<td>0.01</td>
<td>0.08</td>
<td>0.153</td>
</tr>
<tr>
<td>6</td>
<td>0.09</td>
<td>0.01</td>
<td>0.1</td>
<td>0.151</td>
</tr>
</tbody>
</table>

Figure 5.11: Experiment 4 Timing Data

In order to compare how well Saxon and BH Method scale when increasing the number of leaf nodes we plot the timing data for both of them in Figure 5.12. The Saxon XPath processing time increasing rate decreases as the numbers of leaf nodes increase. We can model this trend by performing a logarithmic regression. The BH XPath processing time increases at a constant proportional rate that we model with a linear regression line. As the number of leaf nodes increase the Saxon method scales better because logarithmic function’s rate of increase decreases.
In Experiment 5 we measure the XPath performance by adjusting the pattern tree depth level. The XPath expressions in Figure 5.6 generating pattern trees shown in Figure 5.7 are used in this experiment. The timing results for XPath processing timing data using the BH Method (Steps 4 and 7) and Saxon are shown in Figure 5.11.

<table>
<thead>
<tr>
<th>Number of Leaf Branches</th>
<th>Binding Collection (Step 4)</th>
<th>XML Node Retrieval (Step 7)</th>
<th>BH Method XPath Time (Steps 4 and 7)</th>
<th>Saxon XPath Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.085</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.090</td>
</tr>
<tr>
<td>3</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.101</td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>0.01</td>
<td>0.03</td>
<td>0.135</td>
</tr>
<tr>
<td>5</td>
<td>0.03</td>
<td>0.01</td>
<td>0.04</td>
<td>0.150</td>
</tr>
</tbody>
</table>

Figure 5.13: Experiment 5 Timing Data

In order to compare how well Saxon and BH System scale when increasing the pattern tree depth level we plot the timing data for both of them in Figure 5.14. In order to see the scalability of both systems as we increase the pattern tree depth level we perform regression on the data. We use linear regression to model the timing data for Saxon and BH XPath because for
both systems the processing time increases at a constant rate. From the regression Saxon has an overhead of 0.0175 seconds when increasing the depth level by 1 while the BH Method has an overhead of .0011 seconds per node for every depth level increase by 1. We conclude that the BH Method scales better as the number as pattern tree depth level increases.

![Pattern Tree Depth Level vs XPath processing Time](image)

Figure 5.14: Experiment 5 Plot

We conclude in the BH Method while the loading time is linear there is a need for further optimization. When generating Witness Trees or processing XPath expressions, the BH Method is faster then competitive techniques when the query is deep and non-bushy.
Chapter 6

Conclusions and Future Work

In this thesis we introduce the Binding Hash Method that focuses on the structural join aspect of XML query processing. The theoretical foundations that we build upon were presented in Chapter 3 and a detailed description was given in Chapter 4. We split the BH Method into seven steps for which we define the data structures and algorithms along with examples. In Chapter 4, the 4th step was our most important and involved contribution because it performed the important structural operation of XML query processing. The other steps give us the ability to test our structural join algorithm against XPath expressions and XML documents. Finally we compare the BH Method’s scalability against existing systems in Chapter 5 that perform equivalent functionality.

6.1 Conclusions

The principal conclusions can be described as follows.

We propose a new method called Binding Hash that is able to process an XPath expression by performing structural join and value-based selection.

We develop the BindingCollection index built in Step 4 which is a flexible data structure. In addition to performing structural joins, the BindingCollection index is also used to perform value-based selection and has the flexibility to utilize other XML indexing methods. The BindingCollection index gives us the ability to generate witness trees to be used in systems with XML languages that embed XPath expressions. The BH Method can also be used directly in a system that processes an XPath expression.
We develop algorithms utilizing the BindingCollection index which can process either an XPath expression directly or generate witness trees that can be used in a system whose XML query language includes XPath expressions.

We develop a durable BH XML Index that indexes an XML that the BindingCollection index uses giving us the ability to test our structural join algorithm.

We 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 competitively when generating witness trees and processing XPath expressions when the query is deeply nested and non-bushy.

We characterize the performance of our indexing method as the size of the XML document grows and compare it against Microsoft’s indexing technique. We conclude that the Microsoft’s indexing technique scales better than our and our indexing technique requires further optimization with respect to processing time.

6.2 Future Work

This thesis provides a foundation from which several research issues can be investigated including the following.

1) Add additional indexing to be able to add the feature of inequality-based selection. This will allow us to process XPath expressions which have a less than or greater than comparison operators. Indexing techniques that can be used include B+ trees, XR and XB trees.

2) Optimize the implementation of the BH Method’s indexing technique to reducing time overhead. This involves modifying the index implementation in such a way as to reduce its memory allocation.

3) Design and develop algorithms for node deletion from the XML Index and XML Data tree and analyzer its performance against time.
4) Modify the algorithm to use the ordered tree model that will allow a single XML node to contain an arbitrary amount of text values. Currently our implementation allows a single XML node to have a single text entry.

5) Expand the Post-Order traversal algorithm and the BindingCollection index so the algorithm can process TAX features that we currently do not support which include:

   Value-based joins: Allows to set a constraint requiring attributes across different XML nodes to have the exactly the same value. For example the following query “Return all students which belong to the same department” requires the value attribute of all department XML nodes to have the same value.

   Multiple selections: Allows us to specify more then one constraint against a single XML attribute. For example the query “Return all student whose name starts with an M and end with l,” specifies two constrains against the name attribute against the text attribute for the name XML node.

6) Incorporate the method into a more complex query processing system that supports complex query languages such as Xquery and XSLT that embed XPath expressions.

7) Perform a more extensive performance study for the BH Method. This includes characterizing the performance of the BH Method by adjusting the structural parameters of the XML document such as its depth level and adjusting the number of leaf nodes.
Bibliography


Appendix A: XQuery and Pattern Trees for Experiment 2

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions//item
for $location in $item/location
for $quantity in $item/quantity
for $name in $item/name
for $payment in $item/payment
for $description in $item/description
return
<result>
  <SITE>{$site/text()}</SITE>
  <REGIONS>{$regions/text()}</REGIONS>
  <ITEM>{$item/text()}</ITEM>
  <LOCATION>{$location/text()}</LOCATION>
  <QUANTITY>{$quantity/text()}</QUANTITY>
</ITEM>
</REGIONS>
</SITE>
</result>

Figure A.1: XQuery and Pattern Tree with Leaf Count 1

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions//item
for $location in $item/location
for $quantity in $item/quantity
for $name in $item/name
for $payment in $item/payment
for $description in $item/description
return
<result>
  <SITE>{$site/text()}</SITE>
  <REGIONS>{$regions/text()}</REGIONS>
  <ITEM>{$item/text()}</ITEM>
  <LOCATION>{$location/text()}</LOCATION>
  <QUANTITY>{$quantity/text()}</QUANTITY>
</ITEM>
</REGIONS>
</SITE>
</result>

Figure A.2: XQuery and Pattern Tree with Leaf Count 2
for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $location in $item/location
for $quantity in $item/quantity
for $name in $item/name
for $payment in $item/payment
for $description in $item/description
return
</result>
<br/>Figure A.3: XQuery and Pattern Tree with Leaf Count 3

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $location in $item/location
for $quantity in $item/quantity
for $name in $item/name
for $payment in $item/payment
for $description in $item/description
return
</result>
<br/>Figure A.4: XQuery and Pattern Tree with Leaf Count 4
for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $location in $item/location
for $quantity in $item/quantity
for $name in $item/name
for $payment in $item/payment
for $description in $item/description

return
<result>
  <SITE>{$site/text()}</SITE>
  <REGIONS>{$regions/text()}</REGIONS>
  <ITEM>{$item/text()}</ITEM>
  <LOCATION>{$location/text()}</LOCATION>
  <QUANTITY>{$quantity/text()}</QUANTITY>
  <NAME>{$name/text()}</NAME>
  <PAYMENT>{$payment/text()}</PAYMENT>
  <DESCRIPTION>{$description/text()}</DESCRIPTION>
</result>

Figure A.4: XQuery and Pattern Tree with Leaf Count 4

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $location in $item/location
for $quantity in $item/quantity
for $name in $item/name
for $payment in $item/payment
for $description in $item/description
for $shipping in $item/shipping

return
<result>
  <SITE>{$site/text()}</SITE>
  <REGIONS>{$regions/text()}</REGIONS>
  <ITEM>{$item/text()}</ITEM>
  <LOCATION>{$location/text()}</LOCATION>
  <QUANTITY>{$quantity/text()}</QUANTITY>
  <NAME>{$name/text()}</NAME>
  <PAYMENT>{$payment/text()}</PAYMENT>
  <DESCRIPTION>{$payment/text()}</DESCRIPTION>
  <SHIPPING>{$shipping/text()}</SHIPPING>
</result>

Figure A.5: XQuery and Pattern Tree with Leaf Count 5
Appendix B: XQuery and Pattern Trees for Experiment 3

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $description in $item/description
for $txt in $description/text
return
<result>
  <SITE>{$site/text()}</SITE>
</result>

Figure B.1: XQuery and Pattern Tree with Depth Level 1

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $description in $item/description
for $txt in $description/text
return
<result>
  <SITE>{$site/text()}</SITE>
  <REGIONS>{$regions/text()}</REGIONS>
</result>

Figure B.2: XQuery and Pattern Tree with Depth Level 2

for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions/item
for $description in $item/description
for $txt in $description/text
return
<result>
  <SITE>{$site/text()}</SITE>
  <REGIONS>{$regions/text()}</REGIONS>
  <ITEM>{$item/text()}</ITEM>
</result>

Figure B.3: XQuery and Pattern Tree with Depth Level 3
for $site in document("f.03.xml")/site
for $regions in $site/regions
for $item in $regions//item
for $description in $item/description
for $txt in $description/text
return
  <result>
    <SITE>{$site/text()}</SITE>
    <REGIONS>{$regions/text()}</REGIONS>
    <ITEM>{$item/text()}</ITEM>
    <DESCRIPTION>{$description/text()}</DESCRIPTION>
    <TEXT>{$txt/text()}</TEXT>
  </RESULT>
Appendix C: Binding Hash Shakespeare Demo

In this example we present a working demonstration of the BH Method by executing a query against a sample XML file. For this example we downloaded the Romeo and Juliet XML document partially shown in Figure C.1 from http://www.ibiblio.org/xml/examples/shakespeare/ and executed a query against it in Figure C.2.

Figure C.3 shows the XML nodes that satisfy the XPath expression generated in Step 7. Figure C.4 shows two of the witness trees generated by Steps 5 and 6 for the XPath expression.

```xml
<SPEECH>
<SPEAKER>SAMPSON</SPEAKER>
<LlNE>Fear me not.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LlNE>No, marry; I fear thee!</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>SAMPSON</SPEAKER>
<LlNE>Let us take the law of our sides; let them begin.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LlNE>I will frown as I pass by, and let them take it as</LINE>
<LlNE>they list.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>SAMPSON</SPEAKER>
<LlNE>Nay, as they dare. I will bite my thumb at them;</LINE>
<LlNE>which is a disgrace to them, if they bear it.</LINE>
</SPEECH>
```

Figure C.1: A portion of the Romeo and Juliet XML Document

/PLAY/ACT/SCENE/SPEECH[SPEAKER = 'GREGORY']/LINE

Figure C.2: XPath Expression
<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>No, for then we should be colliers.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>Ay, while you live, draw your neck out o' the collar.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>No, for then we should be colliers.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>Ay, while you live, draw your neck out o' the collar.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

Figure C.3: XPath Expression Results

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>No, for then we should be colliers.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>Ay, while you live, draw your neck out o' the collar.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>No, for then we should be colliers.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

<PLAY>
<ACT>
<SCENE>
<SPEECH>
<SPEAKER>GREGORY</SPEAKER>
<LINE>Ay, while you live, draw your neck out o' the collar.</LINE>
</SPEECH>
</SCENE>
</ACT>
</PLAY>

Figure C.4: Two of the Witness Trees in XML format
In Figure C.5 we show the GUI interface implemented in C# for the Binding Hash Method. The interface allows us to individually execute and benchmark the individual steps of the Binding Hash Method. In the GUI interface clicking on the “XPath answer output” and “Output Witness Trees” buttons generates files shown respectively in Figures C.3 and C.4.

![Test form](image)

Figure C.5: BH Method GUI Interface