APPROXIMATE XPATH

By

LIN XU

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2004

Acknowledgments

I would like to begin by thanking Dr. Curtis Dyreson, my thesis advisor and mentor for the past years. Dr. Dyreson has been a wonderful advisor, providing me with invaluable advice, support. His enthusiasm for research, breadth of knowledge inspires me. His encouragement gives me confidence. I thank him for all the time and energy invested into my research.

I would also like to thank Dr. Kevin Tomsovic who is also my doctoral dissertation advisor, and Dr. David Bakken for agreeing to be on my committee. I thank them for reading my thesis and providing me helpful comments. I was fortunate enough to work with them.

I would like to thank the graduate secretary Ms. Ruby Young, for all her help.

I would like to thank my family. I am forever indebted to my parents for everything that they have given me. Their unconditional support and encouragement give me strengths to finish this work. I dedicate this work to them; to all the people who love me and whom I love.

Last, but far from the least, I thank my wife, Jin Ding from bottom of my heart. I thank her for all her patience and her never-ending encouragement when I frustrated and stressed. She makes my life full of joy and love. I cannot imagine that I could have completed this work without her support.

APPROXIMATE XPATH

Abstract

by Lin Xu, M.S.
Washington State University
MAY 2004

Chair: Curtis Dyreson

As XML has been developed over the past few years, its role has expanded beyond its original domain as a semantics-preserving markup language for online document, and it is now the *de facto* format for data interchanging and integration among distributed, heterogeneous sources. Several query languages have been proposed that are based on path expressions. Because of the inherent data heterogeneity in XML data, exact path expressions may not locate desired data. It is more appropriate to have an approximate query system that can return relevant results when exact path expressions fail to locate the data

This thesis proposes an approximate query language, ApproxXPath, that can cope with data heterogeneity. It extends the popular XPath language by relaxing its semantics. ApproXPath allows both content mismatch and structure mismatch. ApproXPath queries can locate data that is within some number of errors away from the original XML data. The distance away from the exact data is measured by counting how many string edit and tree edit operations are needed to find the data.

Our approach can be categorized as query relaxation. ApproXPath redefines the semantics of axes, node test and predicates based on string/tree edit distance. The algorithms we present use navigation-based query evaluation. We also sketch an index-based solution, which is useful for searching in a XML database. We show that the complexity of ApproXPath is reasonable. The thesis also presents an empirical evaluation.

ApproXPath is implemented in Java. It combines the front end of Apache Xalan with our own approximate query engine as its back end. The thesis reports the performance of ApppproXPath, both exact matching with respect to Xalan and inexact matching varying number of errors allowed. For many queries, the inexact matching (with no errors) is as fast as exact matching and increases linearly with the number of errors.

Contents

List of Tables

List of Figures

Dedication

To my parents, *Jianqiu Xu* and *Jimin Wang*

Chapter 1

Introduction

This thesis focuses on the problem of approximate retrieval in XPath. The problem is to find the resulting node set of a XPath expression while allowing "errors." First we give some background and motivation for this problem. Then we discuss some possible ways to address the problem as well as our approach. Finally there is a summary on our contributions.

1.1 XML

XML–the eXtensible Markup Language–has recently emerged as a medium for data representation and exchange. At its most basic level, XML is a document markup language permitting tagged text (element), element nesting, and element reference. Yet the potential impact is significant: Web servers and applications encoding their data in XML can quickly make their information available in a simple and usable format, and such information providers can interoperate easily. Information content is separated from information rendering, which makes it easy to provide multiple views of the same data.

XML has gained much attention in both the information retrieval community and in the field of database research. One reason is that XML offers a uniform and standardized way to represent and exchange data. The other reason is that XML is very flexible and has great expressiveness power. On one end of the spectrum of XML usage are text-centric documents with only a few, interspersed markups. On the other end of the spectrum are data-centric documents that are solely created and interpreted by applications.

There are several characteristics that make XML an ideal medium of data representation and exchange:

1. *Simple and well-structured.* It is easy to learn and implement. Also XML can be easily parsed

2. *Self-describing.* Straightforward presentation makes it easy for human and machine consumption.

3. *Flexible.* With user defined tags and separation of the content and rendering, it allows users to encode a wide variety of information.

4. *Standard.* XML is vendor-neutral. A lot products from different vendors are available that implement tools for XML.

5. *Semi-structured.* It can support structured, semi-structured and unstructured data, making it an ideal choice for data exchange.

1.2 Data Heterogeneity

Some XML documents have a well defined structure and are stored in homogeneous collections. However, there is a lot of XML data that is inherently heterogeneous. The heterogeneity could be the result of integration of a wide range of data or data that has evolved over time. One example is a federated digital library that combines data from several repositories that have different schemas. Another example is a data warehouse that stores all company-wide XML-formatted documents such as messages and database reports. Such document collections are data-centric but do not have a common schema. Figure 1.1 is a simple XML fragment that integrates data from two different libraries' catalogs. The example shows that data heterogeneity exists in data integration.

Since the data encoding in the XML document could range from data that has a rigid schema to data with sparse markup, it is important for a query language to be versatile.

1.3 Query in XML

With the advent of XML, querying tree-structured data has become a subject of interest in the database community. However, there are several fundamental differences between well researched queries on relational tables and XML.

As we mentioned above, XML is inherently heterogeneous. Lacking of rigid schema poses the biggest difference between a traditional relational database and a XML document. We believe that approximate matching of queries and return a result within a certain error bound is appropriate.

We summarize the necessity for approximate query as following:

```
<publication>
    <books>
        <author name ="Knuth, Donald Ervin">
            <title>The Art of Computer Programming</title>
            <title>Mathematics for the Analysis of Algorithms</title>
        </author>

        <author name="Jim Gray">
            <title>
                Transaction Processing: Concepts and Techniques
            </title>
        </author>
    </books>

    <book>
        <Author>
            <name>Knuth, Donald Ervin</name>
            <title>The TEXbook</title>
        </Author>
    </book>

    <paper>
        <author name="E.F.Codd">
            <date year = "1970">
                <title>
                    A Relational Model of Data for Large
                    Shared Data Banks
                </title>
            </date>
        </author>
        <author>
            <name>Jim Gray</name>
            <date year="2003">
                <title>Consensus on Transaction Commit</title>
            </date>
            <date year="1990">
                <title>
                    An Adaptive Hash Join Algorithm for
                    Multiuser Environments
                </title>
                <title>
                    Parallel Database Systems: The Future
                    of Database Processing
                </title>
                <title>
                    Parity Striping of Disk Arrays: Low-Cost
                    Reliable Storage with Acceptable Throughput
                </title>
            </date>
        </author>
    </paper>
</publication>
```

Figure 1.1: An XML document

- Schema is heterogeneous, or may not be known. This makes it harder to write queries for exact match.

- Schema is complex, which also makes it harder for many users to formulate exact queries.

- There are few or no exact matches, but one may be interested in those similar to the exact matches

- The data and schema evolve frequently, which means that queries need to be flexible. Exact queries tend to be brittle.

Therefore, in order to deal with the above situations, the query must be relaxed. We believe that an ideal approximate XML query engine should have the following characteristics.

- A set of scoring mechanisms should be provided for users to specify the error bound.

- The set of approximate matches to a query is a super set of the exact matches.

- Exact matches for a query must have the highest scoring.

- The result sets can be ranked by relevance order using scoring.

- The query language should be easy to use.

There are two possible ways to relax a query on XML data.

1. Rewrite the query to several possible ways and apply multiple query techniques to those queries to find out common subqueries. This method quickly becomes impractical since the number of relaxed queries increases exponentially as the total number errors allowed increases.

2. Use some techniques to encoding the relax queries into one query and apply post-pruning methods on the result set. This one usually results in a lot of irrelevant results so it is sub-optimal.

1.4 XPath

XPath is a language for addressing parts of an XML document that was designed for use by both the XSL Transformations(XSLT) and XPointer languages. XPath provides a flexible way to specify path expressions. It treats XML document as a tree of nodes. XPath expressions are patterns that can be matched in the XML tree.

Since almost all XML query language model an XML document as a tree, the scoring system in ApproXPath is measured by the tree edit distance. Essentially, there are four kinds of errors/tree edit operations we allowed:

1. Node insertion
2. Node deletion
3. Subtree swapping
4. Node renaming

The first three errors/tree edit operations change the shape of the tree. We say they are *structure errors*. The fourth one, node renaming, does not change the shape of the tree. It changes the content of the XML document node. We call it a *content error*.

We present an approximate XML query system, *ApproXPath*, by allowing content and structure errors. ApproXPath finds the matches that exist in some XML data tree that are within a given edit distance from the original XML data tree. It returns the result grouped by different number of errors. The result with zero errors is identical to traditional XPath, at least semantically.

In order to get the result, ApproXPath relaxes the semantics of the XPath location step. Through proper relaxation, we find the result in the proximity of the current position. Relaxation on an XPath expression is equivalent to applying edit operations to the original XML document tree. More precisely, our query engine returns nodes that would exist in some XML document that is within the given error bound from the original document and we could find an exact XPath match on that modified document.

We treat structure error and content error a little different. We use a different weight for structure errors and content errors. Each structure error counts as one error while all content errors in a single successful inexact string matching count as one error regardless the number of mismatched characters. A Finite State Machine(FSM) is used internally in ApproXPath to trace the number of errors introduced during each evaluation step and to calculate current errors for a node. Nodes with more errors than allowed are eliminated from the result to achieve optimal performance. For structure errors, we relax the semantics meaning of the axis in a location step. But we do not change the order of the axis, that is, forward or backward. Each of our axes under new semantic meaning corresponds to one or several traditional XPath axes. We built a layer on top of traditional XPath operators. This kind of approach could let us take advantage of the current evaluation techniques in the XPath evaluation. Overall, we present two different approaches. One is a navigation-based method using a Finite State Machine(FSM) approach to keep track of error introduced; the other one is a index-based method that takes advantage of an existing structure join algorithm.

The key contributions of our works are listed below:

- Create an approximate XML query engine based on XPath, a popular XML query language.
- Query results are sound in the sense that nodes in the result are with the specified edit distance.
- Eliminate irrelevant results as soon as possible to achieve the optimal speed. The analysis and experiments show that ApproXPath performs well and is highly scalable.

The remainder of the thesis is organized as follows: In chapter 2, we give a detailed introduction of XML and XPath, as well as tree/string edit distance. Related work is also discussed in chapter 2. In chapter 3, the details of our ApproXPath algorithm and analysis of it are presented. Experimental results are given in chapter 4. chapter 5 concludes the thesis and future work is discussed.

Chapter 2

Background and Related Work

Our work involves several parts. Semistructured data lays the foundation for the XML data model. XML and XPath is the start point of our approximate XPath. Edit operations on strings and trees give us the distance metrics for measuring the degree of approximation in our matching.

2.1 Semistructured Data

The semistructured data model [1–3] sits right between the well structured data such as a relational database, and totally irregular data, such as a prose. It plays a special role in database systems:

- It serves as a model suitable for integration of database, that is, for describing the data contained in two or more databases that contain similar data with different schemas.
- It serves as a document model in notations such as XML, which are being used to share information on the web.

Semistructured data model is usually represented as a labeled graph, such as the OEM developed in Stanford University. All the previous research laid a solid foundation to the XML data modeling.

2.2 XML

XML stands for Extensible Markup Language and it grew organically from the need to improve the functionality of Web technologies through the use of a more flexible and adaptable means to identify information. XML is a metalanguage. That is, it is a language that describes other languages. While it may sound circular, even Webopedia defines it as such. What this really means is that XML is more of a standard and supporting structure than a standalone programming language. It is a standard you can follow to create your own language and syntax that meets the XML criteria.

XML provides the facility to define tags and the structural relationship between them. As a result, developers can create their own customized tags (the extensible part of the puzzle) in order to define, share, and validate information between computing systems and applications. Since everything a developer creates adheres to the XML criteria and standard, it allows for customization without many of the usual perils of customization (such as a lack of interoperability and extensibility).

The extensibility and structured nature of XML allows it to be used for communication between different systems, which otherwise would be unable to communicate. While this sounds simple, the magnitude and impact of this benefit is tremendous. Consider this. With the use of XML, you can now communicate not only between internal computing systems but also external systems (vendors, customers, partners, etc.) using a common technology irregardless of the platforms and technologies used for each independent system. Phrased more simplistically, it is like having a single omniscient translator that can work between and among various nations and cultures seamlessly.

Besides the obvious benefit of information integration, sharing and system interoperability, knowledge transfer between your different computing teams becomes easier as well. Since XML has a clearly defined set of standards, people on team A can easily understand and work with information from team B. From an internal resource standpoint, this enables easier staff rotation (and coverage) with a shortened learning curve. From an external relationship standpoint (vendors, consultants, partners), knowledge transfer time is shortened and the actual understanding of the systems and information is enhanced.

From one source of XML-based information you can format and distribute it via a multitude of different channels with minimal effort. Through the use of extensible style language, XSL, developers can easily separate content from formatting instructions. In this way, XSL files act as templates, allowing a single stylesheet to be used to format multiple pages of information. Even more powerful is the ability to use several of these templates to define formatting of the same content for multiple distribution channels.

Many times with both intranet and Internet applications your audience requires data through a variety of channels such as Web, e-mail, text, handheld, wireless devices, and print. With the use of XML and the XSL technologies, you can use a separate stylesheet to distribute the same content to multiple channels. Thus, retrieve the content and data once, deliver many times and in many formats with ease.

The XML has gained much attention in both the information retrieval community and in the field of database research. Originated from different intention and baring different characteristics, there are so called data-centric and document-centric XML.

Data-centric documents are documents that use XML as a data transport. They are designed for machine consumption. Data-centric documents are characterized by fairly regular structure, fine-grained data (that is, the smallest independent unit of data is at the level of a PCDATA-only element or an attribute), and little or no mixed content. The order in which sibling elements and PCDATA occurs is generally not significant, except when validating the document. Data of the kind that is found in data-centric documents can originate both in the database (in which case you want to expose it as XML) and outside the database (in which case you want to store it in a database). An example of the former is the vast amount of legacy data stored in relational databases; an example of the latter is scientific data gathered by a measurement system and converted to XML.

Document-centric documents are (usually) documents that are designed for human consumption. The order in document is important. Examples are books, email, advertisements, and almost any hand-written XHTML document. They are characterized by less regular or irregular structure, larger grained data (that is, the smallest independent unit of data might be at the level of an element with mixed content or the entire document itself), and lots of mixed content. The order in which sibling elements and PCDATA occurs is almost always significant. Document-centric documents are usually written by hand in XML or some other format, such as RTF, PDF, or SGML, which is then converted to XML. Unlike data-centric documents, they usually do not originate in the database.

To summerize, there are several characteristics that make XML an ideal medium of data representation and exchange:

1. *Simple and well-structured.* It is easy to learn and implement. Also XML can be easily parsed.

2. *Self-describing.* Its straightforward presentation makes it easy for human and machine consumption.

3. *Flexible.* With user defined tags and separation of the content and rendering, it allows users to encode a wide variety of information.

4. *Standard.* XML is vendor-neutral. A lot of XML products are available from different vendors.

5. *Semi-structured.* It can support structured, semi-structured and unstructured data, making it an ideal choice for data exchange.

2.3 Tree Representation of XML Document

XML document can be modeled as a tree. Figure 2.1 is the tree represenation of the XML document shown in Figure 1.1

There are seven different kinds of nodes in an XML document tree:

- root node
- element nodes
- text nodes
- attribute nodes

- namespace nodes
- processing instruction nodes
- comment nodes

2.3.1 Data Model

For every type of node, there is a way of determining a string-value for a node of that type. For some types of nodes, the string-value is part of the node; for other types of nodes, the string-value is computed from the string-value of descendant nodes.

Some types of nodes also have an expanded-name, which is a pair consisting of a local part and a namespace URI. The local part is a string. The namespace URI is either null or a string. The namespace URI specified in the XML document can be a URI reference; this means it can have a fragment identifier and can be relative. A relative URI should be resolved into an absolute URI during namespace processing: the namespace URIs of expanded-names of nodes in the data model should be absolute. Two expanded-names are equal if they have the same local part, and either both have a null namespace URI or both have non-null namespace URIs that are equal.

There is an ordering, document order, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the root node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities). The

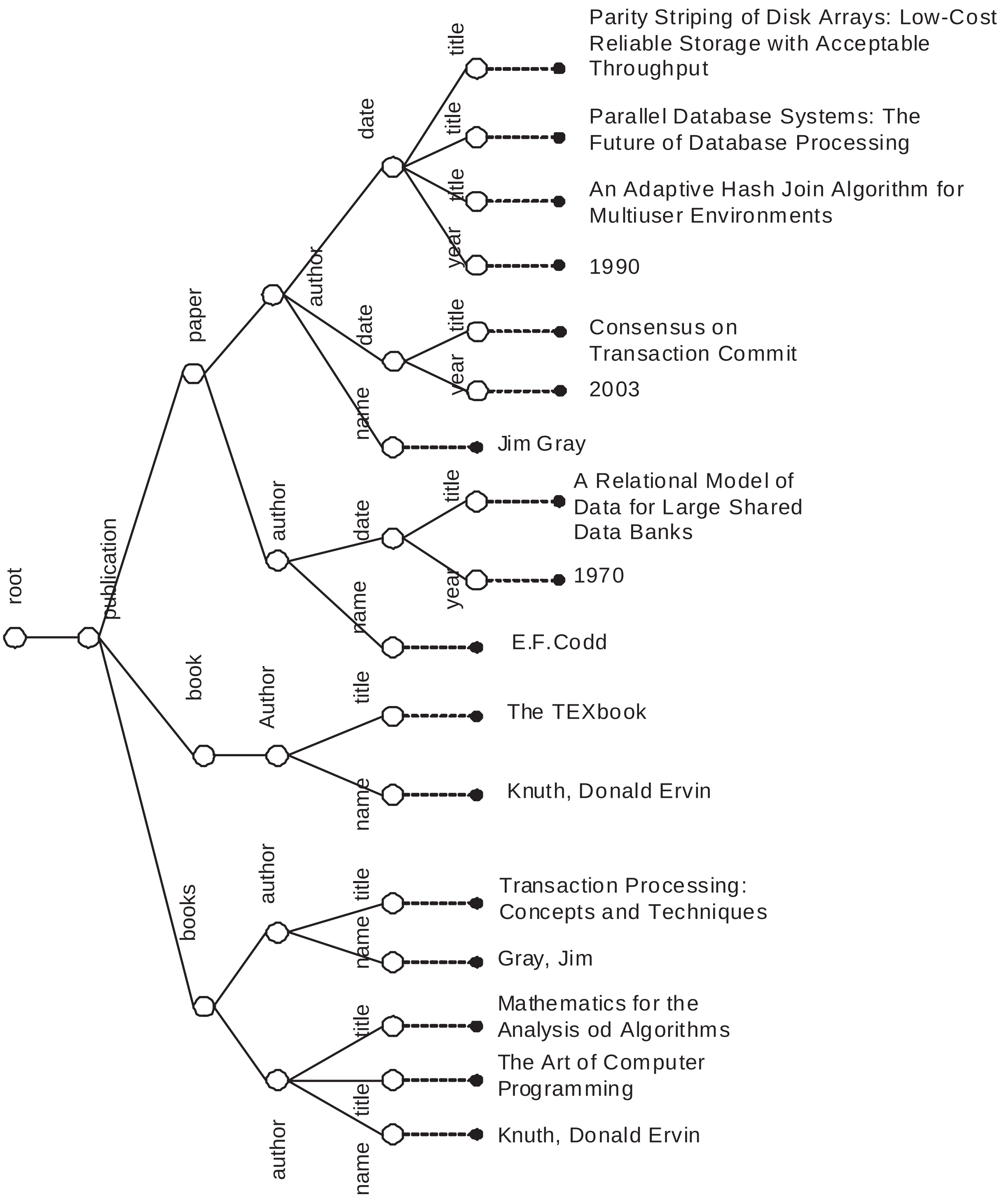

root
publication
paper
author
date
title
Parity Striping of Disk Arrays: Low-Cost Reliable Storage with Acceptable Throughput
title
Parallel Database Systems: The Future of Database Processing
title
An Adaptive Hash Join Algorithm for Multiuser Environments
year
1990
date
title
Consensus on Transaction Commit
year
2003
name
Jim Gray
author
date
title
A Relational Model of Data for Large Shared Data Banks
year
1970
name
E.F.Codd
book
Author
title
The TEXbook
name
Knuth, Donald Ervin
books
author
title
Transaction Processing: Concepts and Techniques
name
Gray, Jim
author
title
Mathematics for the Analysis od Algorithms
title
The Art of Computer Programming
name
Knuth, Donald Ervin

Figure 2.1: The tree model for the XML document in figure 1.1

attribute nodes and namespace nodes of an element occur before the children of the element. The namespace nodes are defined to occur before the attribute nodes. The relative order of namespace nodes is implementation-dependent. The relative order of attribute nodes is implementation-dependent. Reverse document order is the reverse of document order.

Root node and element nodes have an ordered list of child nodes. Nodes never share children: if one node is not the same node as another node, then none of the children of the one node will be the same node as any of the children of another node. Every node other than the root node has exactly one parent, which is either an element node or the root node. A root node or an element node is the parent of each of its child nodes. The descendants of a node are the children of the node and the descendants of the children of the node.

2.4 XPath

XPath [4] is the result of an effort to provide a common syntax and semantics for functionality shared between XSL Transformations (XSLT) and XPointer [XPointer]. The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

In addition to its use for addressing, XPath is also designed so that it has a natural subset that can be used for matching (testing whether or not a node matches a pattern); this use of XPath is described in XSLT.

XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath defines a way to compute a string-value for each type of node. Some types of nodes also have names. XPath fully supports XML Namespaces. Thus, the name of a node is modeled as a pair consisting of a local part and a possibly null namespace URI; this is called an expanded-name. The data model is described in detail in section 2.2.

The primary syntactic construct in XPath is the expression. An expression matches the production Expr. An expression is evaluated to yield an object, which has one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

Expression evaluation occurs with respect to a context. XSLT and XPointer specify how the context is determined for XPath expressions used in XSLT and XPointer respectively. The context consists of:

- a node (the context node)
- a pair of non-zero positive integers (the context position and the context size)
- a set of variable bindings

- a function library

- the set of namespace declarations in scope for the expression

The context position is always less than or equal to the context size.

2.4.1 Location Path

The most important construction of XPath expression is location path. A location path selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path. Location path can recursively contain expressions that are used to filter set of nodes.

There are two kinds of location path: relative location paths and absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by /. The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together. The set of nodes identified by the composition of the steps is this union. An absolute location path consists of / optionally followed by a relative location path. A / by itself selects the root node of the document containing the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document containing the context node.

Location Paths

```
LocationPath ::= RelativeLocationPath
               | AbsoluteLocationPath
AbsoluteLocationPath ::= ’/’ RelativeLocationPath?
                       | AbbreviatedAbsoluteLocationPath
RelativeLocationPath ::= Step
                       | RelativeLocationPath ’/’ Step
                       | AbbreviatedRelativeLocationPath
```

Location Steps [4]

A location step has three parts:

- an axis, which specifies the tree relationship between the nodes selected by the location step and the context node,
- a node test, which specifies the node type and expanded-name of the nodes selected by the location step, and
- zero or more predicates, which use arbitrary expressions to further refine the set of nodes selected by the location step.

The syntax for a location step is the axis name and node test separated by a double colon, followed by zero or more expressions each in square brackets. That is:

$$locationstep ::= axis :: nodetest[predicate]. \tag{2.1}$$

The node-set selected by the location step is the node-set that results from generating an initial node-set from the axis and node-test, and then filtering that node-set by each of the predicates in turn.

The initial node-set consists of the nodes having the relationship to the context node specified by the axis, and having the node type and expanded-name specified by the node test. The meaning of some node tests is dependent on the axis.

The initial node-set is filtered by the first predicate to generate a new node-set; this new node-set is then filtered using the second predicate, and so on. The final node-set is the node-set selected by the location step. The axis affects how the expression in each predicate is evaluated and so the semantics of a predicate is defined with respect to an axis.

Location Steps

```
Step ::= AxisSpecifier NodeTest Predicate*
             | AbbreviatedStep
AxisSpecifier ::= AxisName '::'
                          | AbbreviatedAxisSpecifier
```

Axes

The following axes are available:

- The *child* axis contains the children of the context node.
- The *descendant* axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes.
- The *parent* axis contains the parent of the context node, if there is one.

- The *ancestor* axis contains the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent's parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node.

- The *following-sibling* axis contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the following-sibling axis is empty.

- The *preceding-sibling* axis contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the preceding-sibling axis is empty.

- The *following* axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes.

- The *preceding* axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes.

- The *attribute* axis contains the attributes of the context node; the axis will be empty unless the context node is an element.

- The *namespace* axis contains the namespace nodes of the context node; the axis will be empty unless the context node is an element.

- The *self* axis contains just the context node itself.

- The *descendant-or-self* axis contains the context node and the descendants of the context node.
- The *ancestor-or-self* axis contains the context node and the ancestors of the context node; thus, the ancestor axis will always include the root node.

NOTE: The ancestor, descendant, following, preceding and self axes partition a document (ignoring attribute and namespace nodes): they do not overlap and together they contain all the nodes in the document.

Axes

```
AxisName ::= ’ancestor’
           | ’ancestor-or-self’
           | ’attribute’
           | ’child’
           | ’descendant’
           | ’descendant-or-self’
           | ’following’
           | ’following-sibling’
           | ’namespace’
           | ’parent’
           | ’preceding’
           | ’preceding-sibling’
           | ’self’
```

Node Tests

Every axis has a principal node type. If an axis can contain elements, then the principal node type is element; otherwise, it is the type of the nodes that the axis can contain. Thus,

- For the attribute axis, the principal node type is attribute.
- For the namespace axis, the principal node type is namespace.
- For other axes, the principal node type is element.

A node test that is a $QName$ is true if and only if the type of the node is the principal node type and has an expanded-name equal to the expanded-name specified by the $QName$.

A $QName$ in the node test is expanded into an expanded-name using the namespace declarations from the expression context. This is the same way expansion is done for element type names in start and end-tags except that the default namespace declared with xmlns is not used: if the $QName$ does not have a prefix, then the namespace URI is null (this is the same way attribute names are expanded). It is an error if the $QName$ has a prefix for which there is no namespace declaration in the expression context.

A node test * is true for any node of the principal node type.

A node test can have the form $NCName : *$. In this case, the prefix is expanded in the same way as with a $QName$, using the context namespace declarations. It is an error if there is no namespace declaration for the prefix in the expression context. The node test will be true for any node of the principal type whose expanded-name has the namespace URI to which the prefix expands, regardless of the local part of the name.

The node test $text()$ is true for any text node. Similarly, the node test comment() is true for any comment node, and the node test $processing-instruction()$ is true for any processing instruction. The $processing - instruction()$ test may have an argument that is $Literal$. In this case, it is true for any processing instruction that has a name equal to the value of the Literal.

A node test $node()$ is true for any node of any type whatsoever.

Node Test

NodeTest ::= NameTest
| NodeType '(' ')'
| 'processing-instruction' '(' Literal ')'

Predicates

An axis is either a forward axis or a reverse axis. An axis that only contains the context node or nodes that are after the context node in document order is a *forward axis*. An axis that only contains the context node or nodes that are before the context node in document order is a *reverse axis*. Thus, the ancestor, ancestor-or-self, preceding, and preceding-sibling axes are reverse axes; all other axes are forward axes. Since the self axis always contains at most one node, it makes no difference whether it is a forward or reverse axis. The proximity position of a member of a node-set with respect to an axis is defined to be the position of the node in the node-set ordered in document order if the axis is a forward axis and ordered in reverse document order if the axis is a reverse axis. The first position is

1.

A predicate filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the $PredicateExpr$ is evaluated with that node as the context node, with the number of nodes in the node-set as the context size, and with the proximity position of the node in the node-set with respect to the axis as the context position; if PredicateExpr evaluates to true for that node, the node is included in the new node-set; otherwise, it is not included.

A $PredicateExpr$ is evaluated by evaluating the $Expr$ and converting the result to a boolean. If the result is a number, the result will be converted to true if the number is equal to the context position and will be converted to false otherwise; if the result is not a number, then the result will be converted as if by a call to the boolean function.

Predicates

Predicate ::= ’[’ PredicateExpr ’]’

PredicateExpr ::= Expr

2.5 XPath Evaluation

Formally, for any location path $E_{xpath}()$ applied to context node c we can break it down to a sequence of smaller location step S_1, S_2, ..., S_m, that is:

$$E_{xpath}(T, \{c\}) = S_m(T, ...S_2(T, S_1(T, \{c\}))) \tag{2.2}$$

Each step S takes the initial context node-set or previous step as input, and produces a node-set as the input of the next step. Each node in the previous result set serves as the context node of the next step. The result of the last location step is the result node-set of the location path.

2.5.1 Navigation-Based

Navigation-based method is a straightforward way to evaluate XPath query. It traverses top-down from the document root to get the result. It generally follows the XPath definition, since the XPath definition itself gives a strong hint on navigation-based approach.

If there are multiple XPath queries to be processed, there are simple prefix sharing method and state-of-art $Y - Filter$ [5] that arguments the prefix tree representation and utilizes a Non-deterministic Finite Machine(NFA).

2.5.2 Index-Based

There are some novel indexes and join algorithms developed by mainly the database community. They generally consist of:

- ELEMENT, TEXT tuples that store XML document
- Indexes for tree structure and inverted indexes for content
- Novel join algorithms that take advantage of the indexes

There are several papers discuss this kind of approach. For our best knowledge, the novel method to evaluate XML containment query presented by [6] at SIGMOD 2001 is the first paper of this kind of method. It includes modified inverted index and join algorithm that suitable for XML query. In following paragraphs, we briefly introduce this method, since one of our ApproXPath evaluation method can built on it.

Inverted Index

Inverted index is not an alien. It is around for Information Retrieval(IR) community for quite a while, and is widely used in the web search engines. The content of the documents may stored in somewhere in disks, and the inverted index itself consists of a set of $(word, pointer)$ pairs. The $word$ is the search key for the index while the $pointer$ points to a bucket that contains a set of the physical locations of the occurrence of the $word$.

In order to process structured documents such as XML, the inverted index can be extended in a simple way: text words are indexed in a T-index similar to that used in a traditional IR system, and elements are indexed in an E-index,which maps elements to inverted lists. Each inverted list records the occurrences of a word or an element.

Structure Index [6]

Each term in a XML document is indexed by its document number ($docno$), its position ($begin : end$ or $wordno$) and its nesting depth ($level$) within the document. This is denoted as $(docno; begin : end; level)$ for an element and $(docno; wordno; level)$ for a text word. The position, begin, end or wordno, in a document can be generated by counting word numbers. Alternatively, if the document is in a parsed tree format, the position can be generated by doing a depth first traversal of the tree and sequentially assigning a number at each visit. Since each non-leaf node is always traversed twice, once before visiting all its children and once after, it has two numbers assigned, while leaf nodes have only one number. An inverted list is sorted in the increasing order of docno, and then in the increasing order of begin and end.

The E-index and T-index can be mapped into the following two relations:

name	(1, 4, 3), (1, 25, 3), (1, 39, 3) ...
author	(1, 3, 2), (1, 15, 2), (1, 24, 2)...

Figure 2.2: Fragment of T-index for XML document in figure 1.1

<books>	(1, 2:36, 1) ...
<title>	(1, 8:14, 3), (1, 15:22, 3), (1, 28:34, 3) ...
<author>	(1, 3:23, 2), (1, 24:35, 2) ...

Figure 2.3: Fragment of E-index for XML document in figure 1.1

ELEMENTS (term, docno, begin, end, level)

TEXTS (term, docno, wordno, level)

The ELEMENTS table stores occurrences of XML elements, while the TEXTS table stores occurrences of text words. Each occurrence is stored as a table row. Figure 2.2 and figure 2.3 show the T-index and E-index for the XML document in figure 1.1.

Term occurrences indexed in this way have the following properties:

1. Containment Property(Descendant). An occurrence of a term T_1, encoded as $(D_1; P_1; L_1)$, contains an occurrence of a term T_2, encoded as $(D_2; P_2; L_2)$, if and only if: (1)$D_1 = D_2$, and (2) P_1 nests P_2. For example, (1; 1 :23; 0) contains (1; 9 : 13; 2).

2. Direct Containment Property(Parent-Child). An occurrence of a term $T_1(D_1; P_1; L_1)$ direct contains $T_2(D_2; P_2; L_2)$ if and only if: (1) $D_1 = D_2$, (2) P_1 nests P_2, and (3) $L_1 + 1 = L_2$. For example, (1; 1 : 23; 0) direct contains (1; 2 : 7; 1).

3. Tight Containment Property. An occurrence of a term $T_1(D_1; P_1; L_1)$ tight contains $T_2(D_2; P_2; L_2)$ if and only if: (1) $D_1 = D_2$, and (2) P_1 nests P_2

and nothing else. For example, (1; 14 : 21; 2) tight contains (1; 15 : 20; 3). Because of the nesting structure of XML, tight containment implies direct containment but not vice versa.

4. Proximity Property. An occurrence of a term $T_1(D_1, P_1, L_1)$, is within distance k of a term $T_2(D_2, P_2, L_2)$, if and only if: (1) $D_1 = D_2$, and (2) $|P_1 - P_2| \leq k$. For example, (1,2,3) and (1,4,2) are within distance of 1(appear next to each other).

The above properties allow us to have a variety of operations on inverted lists. To process the expression "$a//b$",the inverted lists of "a" and "b" are retrieved. Occurrences from the two lists are merged if they satisfy the Containment Property. The expression "a/b" can be similarly processed by merging the inverted lists using the Direct Containment Property. The Proximity Property can be used to process string queries such as query processing with distance k = 1. Finally the Tight Containment Property can be used to process expressions such as " <month>='January' " (element '<month>' has only "January" in it and nothing else) involving a self-join on the ELEMENTS table.

Multi-Predicates Merge Join Algorithm [6]

The merge algorithm takes advantage of the index. It is a variation of the merge join that could handle multiple predicates.

This is the algorithm in [6] called Multi-Predicates Merge Join(MPMGJN).

```
procedure containment merge (list1, list2)
begin
```

```
set cursor1 at beginning of list1
set cursor2 at beginning of list2
**while** (cursor1 $\neq$ end of list1 **and**
      cursor2 $\neq$ end of list2) **do**
     **if** (cursor1.docno $<$ cursor2.docno) **then**
    cursor1++
     **else if** (cursor2.docno $<$ cursor1.docno) **then**
    cursor2++
     **else**
    mark = cursor2
    **while** (cursor2.position $<$ cursor1.position **and**
        cursor2 $\neq$ end of list2) **do**
      cursor2++
      **if** (cursor2 == end of list2) **then**
        cursor1++
        cursor2 = mark
      **else if** (cursor1.val (directly)contains cursor2.val) **then**
        mark = cursor2
        **do**
          merge cursor1 and cursor2 values
          cursor2++
        **while** (cursor1 value (directly)contains cursor2 value
              and cursor2 $\neq$ end of list2)
        cursor1++
        cursor2 = mark
```

```
26.           endif
27.         endwhile
28.          endif
29.   endwhile
end
```

This algorithm has an edge over traditional join algorithm since it can handle multiple predicate. It outperforms traditional join in most cases.

There are some other improved versions such as “Structure Join” [7]. These algorithms are all belong to this index-based category.

2.6 Edit Operation on String

A string s is a sequence of alphabets, that is $s = \Sigma^*$.

We consider only those defined in the following form: The distance $d(x, y)$ between two strings x and y is the minimal cost of a sequence of operations that transform x into y (and 1 if no such sequence exists). The cost of a sequence of operations is the sum of the costs of the individual operations. The operations are a finite set of rules of the form $\delta(z, w) = t$, where z and w are different strings and t is a nonnegative real number. Once the operation has converted a substring z into w, no further operations can be done on w.

Note especially the restriction that forbids acting many times over the same string. Freeing the definition from this condition would allow any rewriting system to be represented, and therefore determining the distance between two strings would not be computable in general.

If for each operation of the form $\delta(z, w)$ there exists the respective operation $delta(w, z)$ at the same cost, then the distance is symmetric (i.e. $d(x, y) = d(y, x)$). Note also that $d(x, y) \geq 0$ for all strings x and y, that $d(x, x) = 0$, and that it always holds $d(x; z) \leq d(x; y) + d(y; z)$. Hence, if the distance is symmetric, the space of strings forms a metric space. General substring replacement has been used to correct phonetic errors. In most applications, however, the set of possible operations is restricted to:

- Insertion: $\delta('', a)$, i.e. inserting the letter a.
- Deletion: $\delta(a, '')$, i.e. deleting the letter a.
- Replacement or Substitution: $\delta(a, b)$ for $a \neq b$, i.e. replacing a by b.

2.7 Approximate String Matching

There are numerous algorithms that can match strings with errors. We only introduce the one that based on AGrep [8] that is based on [9].

Its basic idea is to:

1. Encoding each letter in the pattern/text string into a set of bit string with 0/1 indicating if corresponding letter presents at the specified position or not
2. Matching is an AND operation with corresponding pattern and text bit string and advancing is a $Shift$ operation.
3. A set of bit strings is maintained corresponding to matching status under different number of errors.

The detail is described in [8]. Its time complexity is of $O(kn)$ where k is the number of errors allowed and n is the text length.

2.8 Edit Operations on Tree

Definition 2.8.1 (Rooted Tree) *A rooted tree is a structure* $T = (V, E, root(T))$ *that satisfies* $E \subseteq V \times V$ *and* $root(T) \in V$. V *is a nonempty finite set whose elements are called nodes or vertices, while set* E *is a set of pairs* $(u, v) \in V \times V$ *whose elements are called edges. If* $(u, v) \in E$*, then* u *is the parent of* v *and* v *is the child of* u*, that is,*

$u = parent(v)$ *and* $v = child(u)$.

A graph is called labeled when the n vertices are distinguished from one another by names such as v_1, v_2, ..., v_n.

Consider a node *x* in a rooted tree T with root $r = root(T)$. Any node y on the unique path from r to x is called an *ancestor* of x. If y is an *ancestor* of x, then x is a *descendant* of y. (Every node is both an ancestor and a descendant of itself.) If y is an ancestor of x and $x \neq y$, then y is a proper ancestor of x and x is a proper descendant of y. The *subtree* rooted at x is the tree induced by descendants of x, rooted at x.

If the last edge on the path from the root r of a tree T to a node x is (y, x), then y is the parent of x, and x is a child of y. The root is the only node in T with no parent. If two nodes have the same parent, then they are siblings.

An *ordered tree* is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, ..., and the k_{th} child.

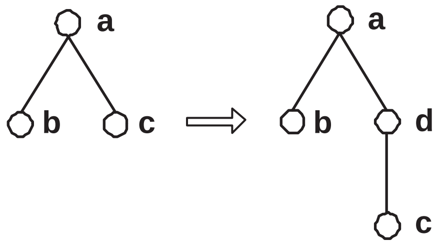


Figure 2.4: Node insertion

There is a value and type associated with each node u, namely, $value(u)$, $type(u)$.

We first define four types of edit operations [10–12] for a rooted tree: context change(E_{change}), node delete(E_{delete}), node insert(E_{insert}) and subtree swap(E_{swap}).

Node Insertion

Insert operation on edge (a, c) means inserting a node d as a child of node a and parent of node c , as shown in figure 2.4.

Definition 2.8.2 (Node Insertion) *Let $T = (V, E, L, root(T))$ be a rooted label tree. the insertion of node $v \notin V$ in edge $(u, w) \in E$ is a function of T such that*

$insert(T, v, (u, w)) = T' = (V', E', L, root(T))$

where $V' = V \cup \{v\}$,

$E' = E - \{(u, w)\} \cup \{(u, v), (v, w)\}$

Node Deletion

The delete operation on node c means letting the children of c becoming the children of parent of c and removing c, as shown in figure 2.5.

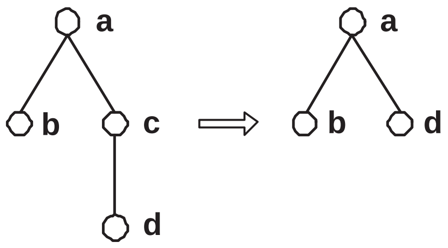


Figure 2.5: Node deletion

Definition 2.8.3 (Node Deletion) *Let $T = (V, E, L, root(T))$ be a rooted label tree. the deletion of node $v \in V$ from T is a function of T such that $delete(T) = (V', E', L, root(T))$*

where $V' = V - \{v\}$,

$E' = E - (\{(x, y)|(x, v) \in E \vee (v, y) \in E\} \cup \{(u, w)|(u, v) \in E \wedge (v, w) \in E\})$

Node Label Change: Change the Context of the Node

The context change operation on node c changes the label of the node, as shown in figure 2.6.

Definition 2.8.4 (Node Relabeling) *Let $T = (V, E, root(T))$ be a XML tree and $v \in T$ be a node. A relabeling of v is the change its label of its label from l_i to l_j. that is,*

relabel(T, v) = (V, E, L', root(t))

where if $\forall u \in V$:

$$label'(u) = \begin{cases} l_j & \textit{if } u = v \\ label(u) & \textit{otherwise} \end{cases} \tag{2.3}$$

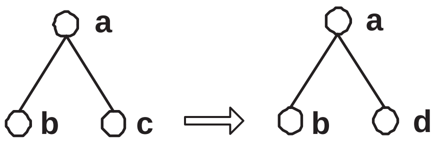


Figure 2.6: Node relabeling

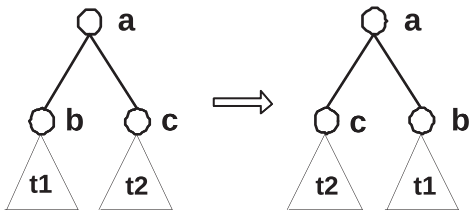


Figure 2.7: Subtree swapping

Subtree Swapping

Subtree swap is to swap the subtree rooted at siblings of same parent, as shown in figure 2.7.

Definition 2.8.5 (Subtree Swapping) *Let* $T = (V, E, L, root(T))$ *be a rooted label tree. the subtree swap of node* $u, v \in V$ *is a function of T such that* $swap(T, u, v) = T' = (V, E', L, root(T))$

where $subtree'(u) = subtree(v), subtree'(v) = subtree(u)$

2.8.1 Cost Function

An arbitrary nonnegative cost function $cost()$ is associated with each type of the edit operation, node context change, node insertion, node deletion and subtree swap. Without loss of generality, assume:

1. $cost() \geq 0$

2. $cost(T \rightarrow T) = 0$

3. $cost(T \rightarrow T') = cost(T' \rightarrow T)$

4. $cost(T \rightarrow T') + cost(T' \rightarrow T'') \geq cost(T \rightarrow T'')$

The above is easily satisfied if we assign each kind of basic operation with same nonnegative cost.

Definition 2.8.6 (Cost of Edit Operation) *The cost of a sequence of edit operations $S = s_1, s_2, ..., s_n$ is defined as $\sum_{i=1}^{n} cost(s_i)$*

Definition 2.8.7 *Definition: The edit distance between two trees T, T' is defined by the minimum edit operations that transform one tree to another.*

$dist(T, T') = min\{\sum_{i=0}^{m} cost(S_i)|S_i$ *is a sequence of edit operations that bring tree T to tree T', i.e. $T' = S_m(...(S_1(S_0(T)))\}$*

As a distance metric, $dist()$ satisfies the triangle inequality:

Theorem 2.8.1 $dist(T_1, T_3) \leq dist(T_1, T_2) + dist(T_2, T_3)$

Proof According to the definition of the distance, there exists a sequence of edit operations $s_1, s_2, ..., s_n$, that bring tree T_1 to T_2, and a sequence of edit operations $s_{n+1}, s_{n+2}, ..., s_m$, that bring tree T_2 to T_3. Thus $s_1 s_2 ... s_n s_{n+1} s_{n+2} ... s_m$ bring tree T_1 to T_3. Again, using the definition of the distance, since dist(T1, T3) is the minimum of all possible sequence of edit operations. Thus $dist(T_1, T_3) \leq cost(s_1 s_2 ... s_n s_{n+1} s_{n+2} s_m)$, that is $dist(T_1, T_3) \leq dist(T_1, T_2) + dist(T_2, T_3)$ QED.

2.9 Related Works

Our work has two related areas: distance metrics in string/trees, query language for XML.

2.9.1 Approximate String Matching

The idea of edit distance and matching with errors were first introduced to the strings and were investigated in [8,13–15]. It is still a very hot topic since its great application in DNA matching and Web searching. The [16] is a very good survey on approximate string matching. Our idea is inspired by Wu and Manber's work on Agrep [8,15], but fundamental difference exists between these two approaches. First, Wu and Manber's work is dealing with string, and our work is on trees. Second, in Wu and Manber's work, in each step, there is only one error can be introduced due to the characteristics of the string, but in trees, each step could introduce more than one error.

2.9.2 Approximate Tree Matching

The idea of approximate string matching was extended to the tree case. For our best knowledge, Tai's paper [10] is the first one to address the tree distance problem. Tree matching problem is useful for compiler code gen and recently it is relatively well researched due to the advent of tree-structure document such as XML, etc. Several papers on this ares are [11,12,17]. In [17], Kilpelainen gave an prove that unordered tree embedding is NP-Hard.

2.9.3 Language Proposals for Approximate Matching

There exist many language proposals for approximate query matching. These proposals can be classified into two main categories: content-based approaches and approaches based on hierarchical structure. In the first category, we find text search and extensions to it for querying position of text (using predicates such as near) in documents (e.g, [18–22]). In the second category, we find [21]. In [21], the author proposes a pattern matching language called approXQL, an extension to XQL [23]. In [19], the authors describe XIRQL, an extension to XQL [23] that integrates IR features. XIRQL's features are weighting and ranking, relevance-oriented search (where only the requested content is specified and not the type of elements to be retrieved) and datatypes with vague predicates (e.g., search for measurements that were taken at about 30 feet). In [22], the authors develop XXL, a language inspired by XMLQL [24] that extends it for ranked retrieval. This extension consists of similarity conditions expressed using a binary operator that expresses the similarity between a value of a node of the XML data tree and a constant or an element variable given by a query. This operator can also be used for approximate matching of element and attribute names. Our work is different from them in that we consider both hierarchical structure and content approximation and our result are grouped according to edit distance metrics. Instead of based on XQL, our work is based on XPath. Form our point of view, it has more expressiveness ability and is more widely used than XQL.

2.9.4 Specification and Semantics

A query can be relaxed in several ways. In [25], the authors describe querying XML documents in a mediated environment. The query language is similar to our tree patterns. The authors are interested in relaxing queries whose result is empty. They propose three kinds of relaxations: unfolding a node (replicating a node by creating a separate path to one of its children), deleting a node and propagating a condition at a node to its parent node. Unfortunately, this work does not consider any weighting and does not discuss evaluation techniques for relaxed queries. Another interesting work is the one presented in [21] where the author considers three relaxations of an XQL [23] query: deleting nodes, inserting intermediate nodes and renaming nodes. By allowing only stylized sequences of deleting nodes (in a bottomup fashion), [21] avoids the combinatorial effects of permitting arbitrary combinations of deletions.

Recently, Kanza and Sagiv [26] proposed two different semantics, flexible and semiflexible, for evaluating graph queries against a simplified version of the Object Exchange Model (OEM). Intuitively, under these semantics, query paths are mapped to database paths, so long as the database path includes all the labels of the query path; the inclusion need not be contiguous or in the same order; this is quite different from our notion of tree pattern relaxation. They identify cases where query evaluation is polynomial in the size of the query, the database and the result (i.e., combined complexity). However, they do not consider scoring and ranking of query answers.

In IR, there are three ways of controlling the set of relaxations that are applied to a query: threshold, top-k and boolean (e.g., [20] and [22]) approaches. Most often, query terms are assigned weights based on some variant of the tf*idf method [27] and probability independence between elementary conditions is assumed. ApproXPath does not use any post-pruning method to limits the result set of relaxation. All the irrelevant results are eliminated as soon as possible.

2.9.5 Approximate Query Matching

There exist two kinds of algorithms for approximate matching in the literature: post-runing and rewriting-based algorithms. The complexity of post-runing strategies depends on the size of query answers and a lot of effort can be spent in evaluating the total set of query answers even if only a small portion of it is relevant. Rewriting-based approaches can generate a very large number of rewritten queries. For example, in [21], the rewritten query can be quadratic in the size of the original query. [28] presents a query that based on tree pattern relaxation and Zhang's MPMGJN [6] algorithm. Her approach is close to our approach, both are query relaxation. But we take different approach to the error measurement and provide more precise measurement on "similarity". She assigned different weight on the different nodes in the query tree and allowed limited relaxation on the query tree. Our approach utilizes tree distance metrics and relatively rich transformation on the tree.

Chapter 3

ApproXPath

3.1 Overview

In this chapter, we present our approach for approximate XML query. Our approach can be categorized as query plan relaxation. The ApproXPath we proposed is based on the popular XML query language, XPath, as we mentioned before. It uses the same syntax as the conventional XPath expression and relaxes its semantics. It apples approximate matching techniques to find the result in the XML document while exact XPath may fail. Approximate XPath widens its search range by allowing edit operation, i.e. allowing both structure and content errors, in XML tree while matching.

Given an XML document, an error bound n, and an XPath location expression e, the result of Approximate XPath expression is those nodes in both the original XML document tree T and some XML document tree T' that is within n edit distance away from tree T. Precisely, we have the following definition:

Definition 3.1.1 (Result Set of ApproXPath) *Given XML document tree T, an er-*

ror bound n, an XPath expression e, let R represent the result set of ApproXPath expression $E_{axpath}(T, e, n)$ *from context node c. Then set R satisfies,*

$$R = E_{ApproXPath}(T, e, n) = \{x | \exists T'[x \in E_{xpath}(T', e) \wedge x \in node(T) \wedge dist(T', T) \leq n]\} \tag{3.1}$$

where $E_{xpath}(T, \{c\})$ *denotes the conventional XPath expression* e *applied on document* T *from context node* c*.*

There are several things to be clarified in the definition above.

1. There is not requirement that tree T' for each element in R should be the same. That means we could have different tree T' for each element in R

2. The result should be in the original tree T, ApproXPath does not introduce additional node to the result set. That is, although we permit inserting spurious node in the original tree, but that node is not returned as a result.

3. The tree distance is measured by each individual node. That is, a result node set R with error n means each node in R has error n, and that $dist(T', T)$ for each node may be measured by different T' at each step. That is, we get x error from tree T' at step i and y error from tree T'' at step $i + 1$, $T' \neq T''$

3.2 Different Approaches

There are several possible approaches to the ApproXPath. One approach is trying to rewrite XPath expression such that applying rewritten XPath expression to the XML document can find out all the nodes that are within given distance away from the original tree; the other is relaxing the query plan for the conventional XPath expression. Our approach is based on the latter one.

3.2.1 XPath Query Rewriting

Query rewriting is used extensively in the relational database query evaluation and optimization. This approach does not modify the existing XPath evaluation engine. Instead, it tries to get the approximate result by rewriting the query and feed the rewritten query(s) to the conventional XPath query engine to get the result. A post-pruning technique may be needed to select the "most similar matching result" based on certain error metrics. The benefit of this approach is that it can take advantage of the existing XPath query engine. The query rewriter is just a layer above the existing XPath engine.

To achieve this, a naive approach is to blindly rewrite the query trying to find as many approximate matches as possible. However, the size of the rewritten query could be exponential, especially if one wants to find queries to cover all the possible structure errors and content errors. The better approach is to rewrite the query based on the schema of the XML data. This is similar to the tree embedding problem. But unfortunately, a large body of XPath query can not be modeled as a query tree. And the schema sometimes is not available due to such as administrative, security reasons.

Since the rewritten query can return a large set of result that only a small portion of it is relevant. A Top-k or post pruning methods is needed to rank/filter the result.

Our ApproXPath uses another approach, query plan relaxation, which we will introduce in section 3.2.2.

3.2.2 Query Plan Relaxation

The relaxation tries to include the relevant result not by rewriting the query, instead it relaxes the specification and semantics of the query. The relaxation for the XPath can be one of the two ways: structure relaxation and content relaxation. Structure relaxation relaxes the relationship between two nodes in the tree. For example, any exact matching result for query $/child :: book/child :: title$ should be those nodes with name $title$ that are the child of those nodes with name $book$. To include more possible result, we relax the semantic of axis $child$. So the axis $child$ contains not only the child but also the grandchild, grandgrandchild, ... of the current node. On the other hand, the content relaxation relaxes the string matching in the evaluation. We still take $/child :: book/child :: title$ for example. Relaxation on string matching let us include nodes with name “titles”, “titled”, “Books”, “Boot”, etc in the result.

The advantage of query relaxation is that it can greatly reduce the computation effort. Since it is built in the query engine, it includes more results while evaluate the XML data. This avoids blindfold rewrite all possible queries since it is confined by the tree it evaluating. It can also take advantage of some optimization technology that specifies to the approximate matching.

3.3 ApproXPath

ApproXPath takes the query plan relaxation approach. By relaxing the conventional exact $Axis$ and $NodeTest$, we introduce the $InexactAxis$ and $InexactNodeTest$ that relax the semantics of the $exactAxis$ and $exactNodeTest$ by allowing structure and content errors.

The semantics for $InexactAxis$, $InexactNodeTest$ and $InexactPredicate$ are discussed in the following sections. The design goals for them are:

1. The result of $InexactAxis$, $InexactNodeTest$ and $InexactPredicate$ should be a superset of the conventional $Axis$, $NodeTest$ and $Predicate$.

2. $InexactAxis$, $InexactNodeTest$ and $InexactPredicate$ with error 0 should be exact the same as conventional $Axis$, $NodeTest$ and $Predicate$.

3. $InexactAxis$ and $InexactNodeTest$ should map to a set of conventional $Axis$ and $NodeTest$.

3.3.1 Inexact Axes

By relaxing the search criteria, the inexact axes are as follows:

1. $InexactAxis$, $InexactNodeTest$ and $InexactPredicate$ with error 0 should be exact the same as conventional $Axis$, $NodeTest$ and $Predicate$.

2. For $(number\ of\ error\ allowed) >= 1$ ($k >= 1$)

   - *ancestor*, *ancestor-or-self*, have no inexact match.

     $ancestor_{inexact}(k) \rightarrow \emptyset$

     $ancestor\text{-}or\text{-}self_{inexact}(k) \rightarrow \emptyset$

   - an *attribute* has one error match, a child:

     $attribute_{inexact}(1) \rightarrow child.$

     This corresponds to a label change of the XML tree. The type of the node is changed from attribute to other non-attribute type. The cost of this change is 1.

- *child* has several inexact matches.

  Inexact match with k errors is:

  $child_{inexact}(k) \rightarrow \underbrace{child/.../child}_{k+1}.$

  There are k+1 $child$ axes on the right hand side. This corresponds to a series of deletions to the XML tree.

  Especially, beside $child/child$, one error inexact child axis also includes the context node itself,

  $child_{inexact}(1) \rightarrow self.$

  And this corresponds to an insertion of a spurious child to context node in the XML tree.

- *descendant* and *descendant-or-self* do not have inexact match.

  $descendant_{inexact}(k) \rightarrow \emptyset.$

  $descendant\text{-}or\text{-}self_{inexact}(k) \rightarrow \emptyset.$

- *following-sibling* has one error inexact match sibling:

  $following\text{-}sibling_{inexact}(1) \rightarrow following\text{-}sibling|preceding\text{-}sibling.$

  This corresponds to the swap of subtree of the parent of the context node.

- *following* does not have inexact match.

  $following_{inexact}(k) \rightarrow \emptyset.$

- *name-space* does not have inexact match.

  $name\text{-}space_{inexact}(k) \rightarrow \emptyset.$

- *parent* has k errors inexact match:

  $parent_{inexact}(k) \rightarrow \underbrace{parent/.../parent}_{k+1}.$

There are k+1 $parent$ axes on the right hand side. This corresponds to a series of deletions in the XML tree.

Especially, beside $parent/parent$, one error inexact parent axis also includes the context node itself. This corresponds to an insertion of a spurious parent to context node in the XML tree.

- *preceding-sibling* has one error inexact match sibling:

  $preceding\text{-}sibling_{inexact}(1) \rightarrow preceding\text{-}sibling|following\text{-}sibling$

- *preceding* and *self* do not have inexact match.

  $preceding_{inexact}(k) \rightarrow \emptyset.$

  $self_{inexact}(k) \rightarrow \emptyset.$

Possible Opimizations

If we take a closer look at those axes that have inexact match, such as the most used $child$ axis, we are actually doing some redundant work. When we try to get result with error$n$, in fact, we already get the result with $n - 1$ errors as a byproduct. Actually, in navigation-based approach, we need travel nodes with $n - 1$ errors in order to get those nodes with n errors. Thus, we can return all those nodes within given error in one shot. Therefore, the result of each $InexactAxis$ is an array of node set. Each node set in the array represents a result that corresponds to a specified error.

3.3.2 Inexact Node Test

A name test can be one of the following three: name test, node type test or processing-instruction test. We only allow error in the name test. The name test in a node test is essentially a string matching. So an error in node test can be categorized as a label change for that node.

An error in name test is a content error. We can apply inexact string matching here. Inexact string matching is a hot topic, since it closely related to the area such as web searching, DNA and protein matching. There are already several good existing inexact string matching algorithms. We take advantage of one of them. The algorithm used here is the *agrep* algorithm developed in University of Arizona.

There is a threshold for the content errors (mismatched characters) in the name test. If there is no matching within the given amount of errors then we consider the name test failed. It is more reasonable to define the threshold as a percentage of the string length of the QName of that node, since the length of the QName can vary greatly. This threshold is adjustable. We choose 40% in our ApproXPath.

From another point of view, as we mentioned above, we can model the error in node test as a label change to that node. So, regardless how many mismatched characters in the inexact string matching, if there is a match within given bound , we count them as one error (a label change).

In summary,

1. Inexact Node Test can have one of the three kinds of results: successful with error 0, successful with 1 error or failed.

2. For name test:

   - For name test in node test, if an exact matching is achieved, we say it is successful with 0 error.
   - Or if an inexact matching within given error bound is achieved, we say it is successful with 1 error.

- Otherwise, the name test fails.

3. There are no inexact node test for *nodetype()*, and *processing-instruction()*. So they are identical to the conventional Node Test. There are only two outcomes for the evaluation: successful with 0 erroror failed.

3.3.3 Inexact Predicate

Predicate expression is evaluated for each node in the result of the $node\ test$. Each node from the result acts as a context node for the predicate expression. The number of nodes in the result is the context size and the proximity position of a node in the result is the context position for the predicate expression. Since it basically just another XPath expression, the inexact matching discussed previously is also applied here.

1. For the case there are more than one predicates, $InexactPredicate$s are evaluated from left to right.

2. For the case that the Predicate contains a LocationPath, $InexactPredicate$ is evaluated as described in $InexactAxis$ and $InexactNodeTest$

3. For the Equality expression in Predicate that compares a position (position(), etc) to a number one error inexact match is true for any case other than exact match(subtree swap). No inexact match for more than one error.

4. For the Equality expression in Predicate that compares between two strings, one error inexact match is true for inexact string match with specified errors. No inexact match for more than one error.

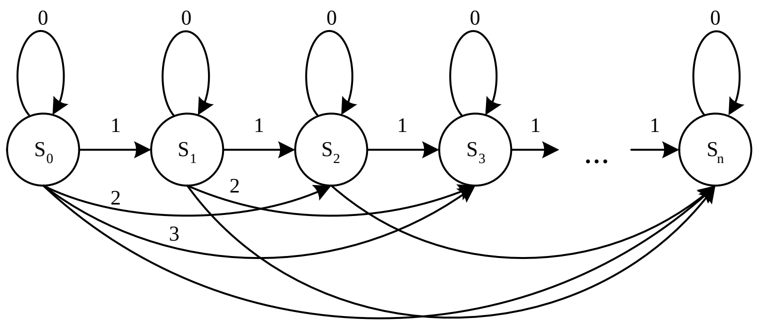


Figure 3.1: The state diagram of ApproXPath evaluation

3.3.4 Glue Together

The result of ApproXPath is the result of $InexactAxis$, $InexactNodeTest$ and $InexactPredicate$. A series of node sets is returned. Each node set corresponds to the result of a specific number of error allowed. That is, if we specify that we allow 5 errors in the evaluation, a total of 6 node sets are returned, corresponding to 0 error (exact match) to 5 errors (5 errors introduced during the evaluation). Similar to the conventional XPath evaluation, each intermediate result node set is the start point for the next step. However, the difference between the ApproXPath and conventional XPath lies in that the intermediate result is maintained in a set of the node set. Each node in those sets generate the next step of node set according to a Finite State Machine(FSM) shown in Figure 3.1.

From the FSM, one can find out that if a node in an intermediate result set is with k errors then the result of next step of evaluation starting from this node is in the intermediate result set of $k + i$ errors, if i errors is introduced during the next evaluation step. If $k + i > n$, n is the total number allowed, the result is discarded.

3.4 ApproXPath Evaluation Plan

Due to the similarity between ApproXPath and conventional XPath, the evaluation of the conventional XPath can shed some light on the evaluation of our ApproXPath.

Basically, we divide the XPath query into a sequence of $atomicstep$s.

Atomic Step

A atomic step is an indivisible step in ApproXPath evaluation that can introduce an error. The following steps are atomic step:

1. Axis
2. NodeTest
3. Equality expression with both side evaluated

There are two categories of methods on the evaluation those XPath steps. One is the navigation based technology, the other is the index based method that utilizes some novel join algorithms.

Due to the flexibility of our ApproXPath, our ApproXPath can be implemented using the methods in either category.

Before we present the method for each category, we need to discuss some method that prevents our algorithm becoming exponential.

Consider the following problem, suppose that one have m errors to distributed among n steps, there are $\begin{pmatrix} n+m-1 \\ m \end{pmatrix}$ ways to do that, that is, it is exponential to m, the number of errors .

In order to avoid the exponential cost, we introduce the $Merge$ step at proper stage in the evaluation. Merge is essentially rearranging the intermediate result set to avoid unnecessary redundant calculation. The idea for merge is that, since a node could appear in different intermediate result set with different error number, we do a set difference operation to ensure that a node is only appear in one node set (with least number of error) and the intersection of any two different intermediate result set is empty. The cost of set difference varies by the implementation of the set itself. For a sorted set or hash set, the cost is $O(n)$, while for others, the cost is $O(log(n))$, where n is the size of the set.

The following sections discuss two different implementations of ApproXPath.

3.4.1 Navigation-Based Approach

Navigation is a very natural way to evaluate the XPath query. It starts from the document root and performs a top-down traverse on the XML document tree to get the result. It is straight-forward, easy to implement, but sometimes suffer performance penalties. This approach is used in many XPath query engine such as Apache Xalan.

Here are the general steps of the navigation-based ApproXPath with error bound n:

1. Break the ApproXPath location path into atomic steps, each LocationStep may be broken into Axis, NodeTest, etc.

2. Initialize a collection of set $R_{0...n}$ corresponding to the given number of errors. Assign context node to R_0 and initialize sets $R_{1...n}$ to empty set

3. For each of the atomic step:

(a) Apply appropriate $InexactAxes$, $InexactNodeTest$ and $InexactPredicate$ to each node in sets $R_{0...n}$. Store the result in a series of set $R'_{0...n}$.

(b) Apply $Merge$ operation on sets $R'_{0...n}$. The result is put back to sets $R_{0...n}$.

Therefore, similar to the conventional XPath expression, for any approximate location path $L_{ApproXPath}()$ applying to context node c, we can break it down to a sequence of atomic steps s_1, s_2, ..., s_m, and with each step, there is a error bound e_i associate with it,

$$R_0 = (e_0, e_1, ..., e_n) = (\{c\}, \emptyset, ...\emptyset) \tag{3.2}$$

$$R_i = S_i(T, R_{i-1}) \tag{3.3}$$

$$L_{ApproXPath}(T, R_0) = S_m(T, ...(S_2(T, S_1(T, R_0))) \tag{3.4}$$

The result set of the intermediate step is: Suppose that location path expression $E_{xpath}(T, c)$ could be divided into m location step s_1, s_2,..., s_m, and totally n errors are allowed. The set R_i for location step s_i satisfies:

$$R_i = \{x | \exists T'_j [j \in (0, ..., i) \wedge x \in L_{XPath}(T'_i, r_i) \wedge x \in node(T) \wedge \sum_{j=0}^{i} dist(T'_j, T) \le n]\} \tag{3.5}$$

and the result set is set $R = R_m$

From the definition of the intermediate result set, one can find that we do not require that the tree T_i should be the same for each node. Therefore the tree uses for node v may not necessary be the same tree used for node u.

Memory or Memory-less?

In the navigation-based approach, we always start from the nodes in the original XML tree and intermediate result is also in the original XML tree. However, for each node in the intermediate or final result set, there are different ways to get to that node. Therefore, based on whether to remember the path leading to the node in those result set, there are two approaches we can use, *memory* and *memory-less*. *Memory* approach is to remember the intermediate step, that is, remember the path and the error introduced in the way that leads to the current node. The way of doing this is to associate each node with a set of traces that can lead to the current node. This could be very expensive. The cost of building these traces could be exponential, since there are potentially exponential ways to reach a certain node. For example, if the node matched is at n^{th} step of ApproXPath, and m error is allowed, for exact m error alone, there are $\begin{pmatrix} n+m-1 \\ m \end{pmatrix}$ ways to get to the current node. The advantage of memory approach is that it makes precise calculation of the introduced error possible. The other approach is *memory-less*. This approach does not record any trace of previous steps. Thus, the only thing differentiate nodes is the errors introduced to get to those nodes. This approach is very efficient, since it does not need to store the paths to get to that node. On the next matching, the choice is also very simple, since it just based on the node itself, not considering how to get to that node. The disadvantage is that it may miscalculate the error introduced. It could over count errors when backward and forward axes are mixed. An edit operation is introduced in previous step may be count again in the next match in such situation, since no history information is recorded.

In memory approach, T_i in the matching process is identical, while in the memoryless approach, T_i could be different. Comparing this to the initial goal of approximate XPath, one can find that difference between the stepwise approach and the original approximate XPath definition is that in stepwise approach, a sequence of XML trees T_i is used in matching for one node instead of a single tree T used in original approach. The question is that whether there is an XML tree T' such that $T' = T_1 = T_2 = ... = T_m$, that is, the two approach is equivalent.

If XPath location path contains only forward or backward location axes, that is, they are *one direction*, we have the following claim for memory-less approximate Xpath approach:

Claim: If the XPath location expression is of one direction, and each of its location steps does not involve nodes beyond its intermediate result set and navigation-based ApproXPath evaluation is used, then for any node in the result set, there exists a tree T' such that $dist(T, T') \leq \sum_{i=1}^{m} d_i$, and $T' \cap T_i = T_i$, where m is the total evaluation steps, and $d_1, d_2, ...d_m$ are the edit distance introduced in evaluation step 1,2, ... m.

Justification: Suppose T_i is the tree used in evaluation step i. From the way of stepwise approach works, there are some errors introduced in between the context node c_i and the result set R_i and no more. By definition, T_i should include such changes. Since both c_i and R_i are in the original tree and the edit operation is localized except swap operation, and swap operation does not change the subtree of the R_i, one can find that the edit operation is localized between c_i and R_i in the original tree. Therefore, except the region between c_i and R_i, the rest of the T_i tree could be identical to the original tree T. If not, construct a tree in this way, retaining the region between the context node c_i and the result set R_i of tree T_i, the rest of the tree is identical to the original tree T. One can find a T_i that is only different from tree T in the region between context node c_i and the result set R_i, while still satisfying stepwise approach requirement. Then, the edit operation is localized in between c_i and R_i, one could *concatenate* them to form a tree T'' which is an approximation to original error tree T'.

The equality is the nature sequence of the triangle inequality of $dist()$.

3.4.2 Algorithm for Navigation-Based Approach

The inputs of the navigation algorithm are XML document tree T, the XPath expression l and the error bound n. The return value of the algorithm is a node-set that each node in it satisfies the condition of equation.

This algorithm maintains a series of sets, S_j^i, where $j = 0...m$ and $i = 1...n$, m is the total number of the location steps. The nodes in S_j^i are the intermediate result of step j with minimum error if i.

```
NodeSet[] InexactAxis($R$:node set, $a$:axis, $n$:max error allowed)
{
1.   initialize NodeSet array S[$n$] = $\emptyset$
2.   foreach node $c$ in $R$
3.        T = result of applying inexact axis $a$ on context node $c$
4.        for i = 0 to n
5.             S[i] = S[i] $\bigcup$ T[i]
6.   return S
}
NodeSet[] InexactNodeTest( $c$:node, $t$:nodeTest)
```

```
{
1.  NodeSet S[2]
2.  S = result of applying inexact node test $t$ on the context node $c$
3.  return S
}
```

approLocationPath (l:LoactionPath, n: maximum error permitted, R[]:input/output Node Set)

```
{
1.  Divide the XPath LocationPath $l$ into a sequence of $m$ location steps, $l = s_1, s_2, ..., s_m$
2.  Initialize temporary array $S$
3.  for i = 1 to m
4.      S = R
5.      for j = 0 to n
6.          $V = InexactAxis(R[j], axis[i], n - j)$
7.          for k = 0 to n-j
8.              $S[j + k] = S[j] \cup V[k]$
9.      for j = 0 to n
10.         $S[j] = S[j] - \bigcup_{k=0}^{j-1} S[k]$
11.     for j = 0 to n
12.         $V = InexactNodeTest(S[j], axis[i], n - j)$
13.         for k = 0 to $min$(n-j, 1)
14.             $S[j + k] = S[j] \cup V[k]$
15.     R = S
16.     foreach (predicate $p$)
17.         set S to array of empty set $\emptyset$
18.         for j = 0 to n
19.            **foreach** node $c$ in $R[j]$
20.                B = approPredicate($c$, $p$, $n-j$)
21.            **for** k = 0 to n-j
22.                if(B[k] == *true*)
23.                    $S[j+k] = S[j+k] \cup \{c\}$
24.     R = S
}
boolean[] **approPredicate** ($c$:context node-set, $p$:predicate, $n$:maximum error permitted)
{
1.  parse the Predicate $p$
    ... ...
2.  Initialize array $R$ to $\emptyset$, set $R[0] = \{c\}$
3.  **foreach** LocationPath $l$ in $p$
4.      approLocationPath ($l$, $n$, $R$)
    ... ...
}
```

3.4.3 Complexity Analysis for Navigation-Based Algorithm

Time Complexity

In $approLoactionPath$ function, statements inside loop starting from line 3 execute m (number of location steps) times. line 8 executes n^2 (n is the total number of errors allowed) times. But considering that $S[i]$ are mutual exclusive and $\bigcup S[i] = V$, all nodes in a XML tree. Thus, line 8 has time complexity of $O(nv)$.

The same thing applies to line 14. The cost for InexactAxis is $O(v^2)$. Therefore, the total time complexity for the algorithm is of $O(m * n * v + m * v^2)$.

On the other hand, we care about the extra cost of ApproXPath over the conventional navigation XPath evaluation engine. In worst case, the time will be n times of the conventional XPath evaluation time. Our experiments demonstrate that our analysis is correct.

Space Complexity

Since the maximum size of all the internal result set can not exceed the total number of the node in XML document tree, thus, the space complexity is $O(v)$.

3.4.4 Index-Based Approach

The previous section describes the navigation approach of ApproXPath implementation. The second approach of ApproXPath is based on the algorithm [6].

The works lie in following two areas:

1. Replace exact inverted index to approximate inverted indexes

2. Including error number and relax predicate in join.

A natural extension to the MPMGJN is that we include one more field called $errno$ in the ELEMENT and TEXT records we use in join. The $errno$ works in this way, if two records with $errno_1$ and $errno_2$ are joined, the result $errno_r$ is the sum of the two. i.e. $errno_r = errno_1 + errno_2$.

The above is for structure error. As for content error, if the record is an exact match for the string, the $errno$ is set to 0; if it is a match within specified error bound, it is set to 1.

Also, the inverted index needs to extend to approximate inverted index. There are several out there. They are the same as the original one. Only difference is in the $lookup$ algorithm. Exact match can take advantage of B-Tree, hash table etc, while inexact match is essentially a linear search. It compares every record in the index to find the "close" one.

So overall the algorithm is:

1. Populate the XML document in the database in the format mentioned in 2, i.e., ELEMENT and TEXT tuples, with one additional field call $errno$ indicating the error number involved with the current record. Treat attribute as child.

   ELEMENTS (term, docno, begin, end, level, errno)

   TEXTS (term, docno, wordno, level, errno)

2. Break the XPath expression into a set of select, containment join, etc. A query compiler is needed here.

3. For any query of style: "$a/b[pred]$" , do the following:

   (a) Lookup approximate inverted index to find ELEMENT/TEXT records with name matching "a" or "b" within given error bound. Mark field errno 0 for exact matching and 1 for inexact matching.

   (b) Using join algorithm indicated in the following example.

   (c) Eliminate the records with $errno$ field larger than given error bound.

There are already some implementations of the approximate inverted indexes. We can take advantage of them. So we devote effort on how to extend the MP-MGJN algorithm.

We use an example using pseudo SQL code to explain our algorithm. Query execution engine uses the following way to execute the expression: “$term$ LIKE ‘SomeString’”: uses inverted indexes to fetch any the element exactly matches $term$ ‘SomeString’ or inexactly matches ‘SomeString’ within a specified edit distance and increases $errno$ of those inexact matching tuples by 1.

$/a//b/c$

1. **Join.** Store result in temporary table/view T1 (term, docno, begin, end, errno).

   ```
   SELECT e2.term, e2.docno, e2.begin, e2.end,
          errno AS (e1.errno + e1.errno)
   FROM element e1, element e2
   WHERE e1.term LIKE ‘a’
         AND e2.term LIKE ‘b’
         AND e1.docno = e2.decno
         AND e1.begin < e2.begin
         AND e2.end < e1.end
   ```

2. **Merge.** Store result in temporary table/view T2 (term, docno, begin, end, errno).

   ```
   SELECT T1.term, T1.docno, T1.begin, T1.end,
          errno AS min(T1.errno)
   FROM T1
   GROUP BY T1.term, T1.docno, T1.begin, T1.end
   ```

3. **Join.** Store result in temporary table/view T3 (term, docno, begin, end, errno)

   ```
   (SELECT e2.term, e2.docno, e2.begin, e2.end,
           errno AS (e1.errno + e2.errno + e2.level - e1.level - 1)
   FROM T2 e1, element e2
   WHERE e2.term LIKE ‘c’
         AND e1.docno = e2.decno
         AND e1.begin < e2.begin
   ```

```
      AND e2.end < e1.end
)
UNION
(SELECT e3.term, e3.docno, e3.begin, e3.end,
        errno AS (e3.errno + 1)
FROM element e3
WHERE e3.term LIKE ‘c’
)
```

4. **Merge.** The result is the final result with schema (term, docno, begin, end, errno).

```
SELECT T3.term, T3.docno, T3.begin, T3.end,
       errno AS MIN(T3.errno)
FROM T3
GROUP BY T3.term, T3.docno, T3.begin, T3.end
```

Chapter 4

Implementation and Empirical Performance Analysis

4.1 Implementation

Our Approximate XPath is implemented in Java using Sun J2SE 1.4.2_03. Its front end, the XPath language parser, is based on Apache Xalan J-2.5.1 XPath package. The back-end is our own Approximate XPath query engine implementing the algorithm described in chapter 3. It is a relative complete implementation of XPath Location Path.

This approach is to demonstrate how adaptable of our algorithm to existing XPath implementation. This also makes the comparison with Apache Xalan reasonable because we share the same front end.

4.2 Capability Test

In this section, we run a set of queries on the example XML file, $foo.xml$ that comes with Apache Xalan J-2.5.1.

```
<?xml version="1.0"?>
<doc>
  <name first="David" last="Marston"/>
  <name first="David" last="Bertoni"/>
  <name first="Donald" last="Leslie"/>
  <name first="Emily" last="Farmer"/>
  <name first="Joseph" last="Kesselman"/>
  <name first="Myriam" last="Midy"/>
  <name first="Paul" last="Dick"/>
  <name first="Stephen" last="Auriemma"/>
  <name first="Scott" last="Boag"/>
  <name first="Shane" last="Curcuru"/>
</doc>
```

Query 1

The query tests the ability of ApproXPath to handle content errors.

Query:

$/docs/name[@name = “david''"]$

Command:

```
java ApplyAXPath foo.xml "/docs/name[@first=‘dvid’]" 2
```

comment: The query has two errors, $doc \to docs$ and $David \to dvid$ both content errors.

Result:

```
ApproXPath result with error 0:

<output0>

</output0>
```

```
ApproXPath result with error 1:

<output1>

</output1>


ApproXPath result with error 2:

<output2>
<name  first="David" last="Marston"/>
<name  first="David" last="Bertoni"/>
</output2>
```

Query 2

The query tests the ability of ApproXPath to handle the situation of mixing content errors and structure errors.

Query:

$/name[@first = `sctt']$

Command:

```
java ApplyAXPath foo.xml "/name[@first='sctt']" 2
```

comment: The query has two errors, $name$ promoted to the root child, which is a structure error and $Scott \rightarrow sctt$, which are content error.

Result:

```
ApproXPath result with error 0:

<output0>

</output0>


ApproXPath result with error 1:

<output1>
```

```
</output1>


ApproXPath result with error 2:

<output2>
<name  first="Scott" last="Boag"/>
</output2>
```

From the above two query tests, the ApproXPath demonstrate that it can handle both structure and content errors well.

4.3 Empirical Performance Analysis

The performance tests are carried out on three different trees with different shapes and relative large size. The comparison between Apache Xalan J-2.5.1 and ApproXPath are presented. And the execution time of ApproXPath on different number of errors allowed are also presented and discussed.

4.3.1 Testing Suite

The testing suite for our approximate XPath engine is a set of randomly generated trees. We tested our engine by varying the following parameters:

1. Degree of the document node. This factor represents the number of the children of the document root. It controls the top level bushiness of the XPath document tree.

2. Depth of the tree. This factor represents the level of the nesting of elements in the XML document. It controls the depth of the XML tree.

3. The Bushy factor. This factor describes the number of children (degree) in a non-leaf node in the XML tree. The bushiness can be fixed or chosen

randomly from a range.

In [29], one can find more details on benchmark on XPath engine.

From the result one can find that our relaxed XPath query engine outperforms Apache Xalan j-2.5.1 and scales well with the total number of error allowed.

4.3.2 Comparison ApproXPath to Xalan J-2.5.1 on Different Queries.

Bushy Tree

The XML document size is 9.04 MB (9,480,448 bytes), with bush factor being 5, depth being 6 and leaves per internal nodes being 30.

Seven different queries are used to test the performance between Apache Xalan J-2.5.1 and ApproXPath:

Query 1: /doc/level1/level2/level3/level4

Query 2: /doc/level1[@pos='1']/level2[@pos='2']/level3[@pos='3']/level4[@pos='4']

Query 3: /doc

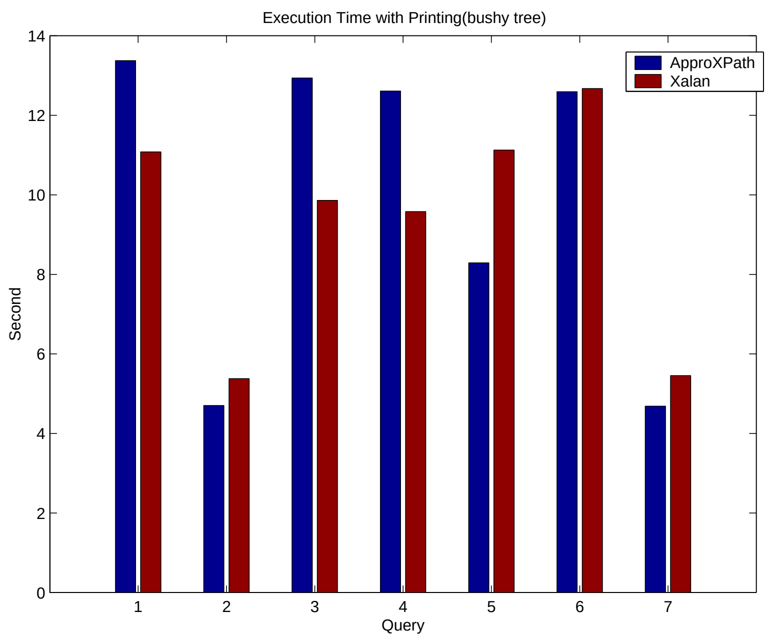
Query 4: /doc/level1

Query 5: //name[@first='Emily']/preceding-sibling::*

Query 6: /doc/level1/level2/level3/level4/level5/level6

Query 7: /doc/level1[@pos='1']/level2[@pos='2']/level3[@pos='3']/level4[@pos='4']/level5[@pos='5']/level6[@pos='5']/name[@first='David'][@last='Marston']

From Figure 4.1, we can find that ApproXPath is on par with Xalan when the result is printed, but consistently faster than Xalan when no result is printed out. On those cases with result printed out, ApproXPath is slightly faster than Xalan on query 2, 5, 6, 7. The reason for that is the node set for print out on these queries is small. Xalan has a better way to traverse XML tree and print out result than ApproXPath.



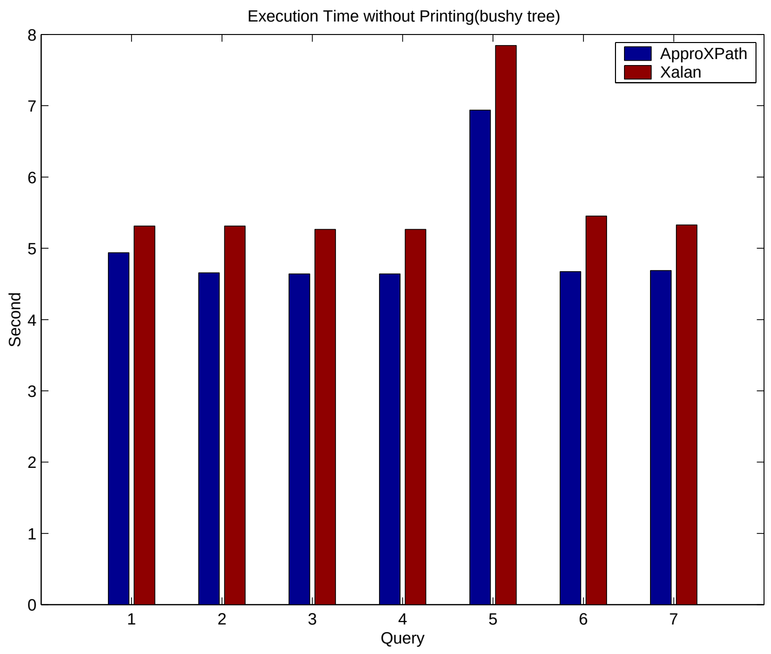


Figure 4.1: Execution time on bushy tree

Tall Tree

The XML document size is 247 KB (253,401 bytes), with the number of root child being 5, depth being 500, bush factor being 1 and leaves per internal nodes being 30.

Seven different queries are used to test the performance between Apache Xalan J-2.5.1 and ApproXPath:

Query 1: “/doc/level1/level2/level3/level4”

Query 2: “/doc/level1[@pos=‘1’]/level2[@pos=‘1’]/level3[@pos=‘1’]/level4[@pos=‘1’]”

Query 3: “/doc”

Query 4: “/doc/level1”

Query 5: “//name[@first=‘Emily’]/preceding-sibling::*”

Query 6: “/doc/level1/level2/level3/level4/level5/level6”

Query 7: “/doc/level1[@pos=‘1’]/level2[@pos=‘1’]/level3[@pos=‘1’]/level4[@pos=‘1’]/level5[@pos=‘1’]/level6[@pos=‘1’]//name[@first=‘David’][@last=‘Marston’]”

From Figure 4.2, we can find that ApproXPath performs very close to or slightly better than Xalan when the result is printed, but consistently faster than Xalan when no result is printed out. As mentioned above, since the node set for printing out is smaller, ApproXPath shows its strength, especially on query 2, 5, 6, 7.

Fat Tree

The XML document size is 5.73 MB (6,012,005 bytes), with number of root child 100, depth 2, bush factor 1, leaves per internal nodes are 30.

Seven different queries are used to test the performance between Apache Xalan j-2.5.1 and ApproXPath:

Query 1: “/doc/level1/level2”

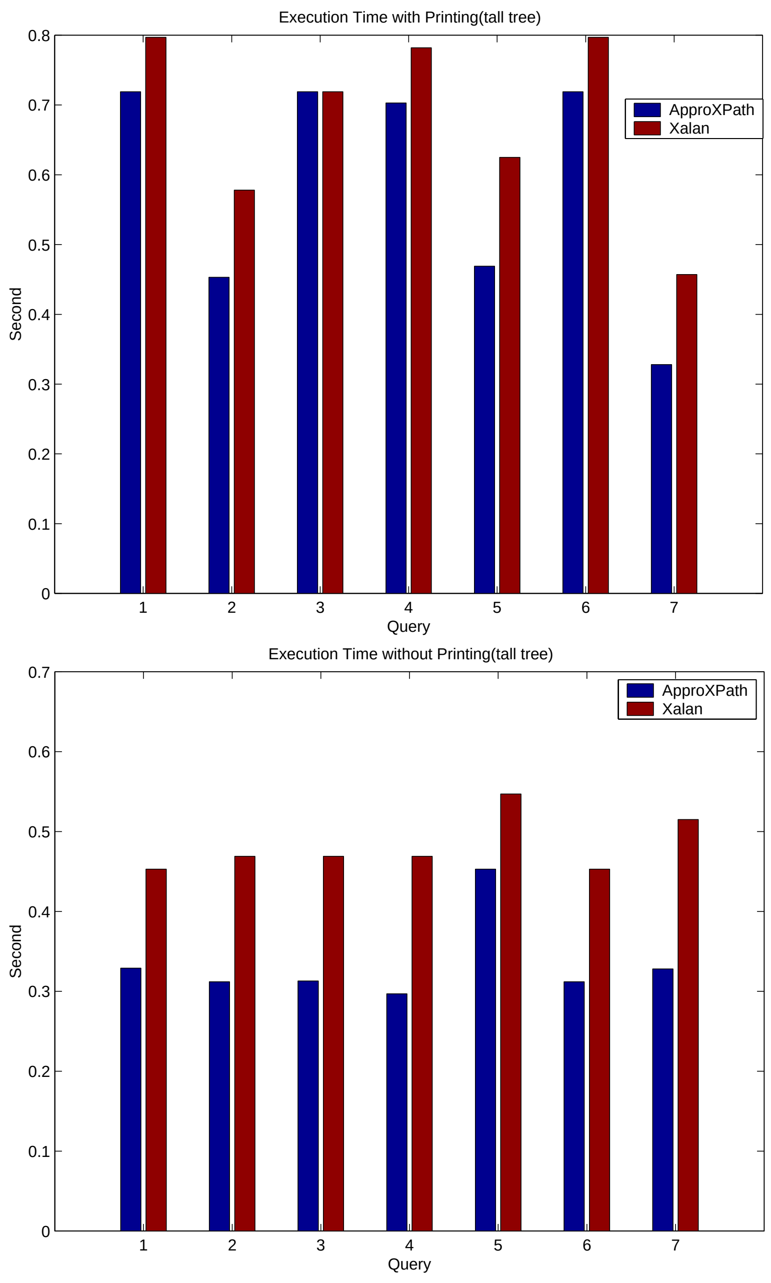


Figure 4.2: Execution time on tall tree

Query 2: “/doc/level1[@pos=‘1’]/level2[@pos=’1’]”

Query 3: “/doc”

Query 4: “/doc/level1”

Query 5: “//name[@first=’Emily’]/preceding-sibling::*”

Query 6: “/doc/level1/level2/name”

Query 7: “/doc/level1[@pos=‘1’]/level2[@pos=‘1’]/name[@first=‘David’][@last=‘Marston’]”

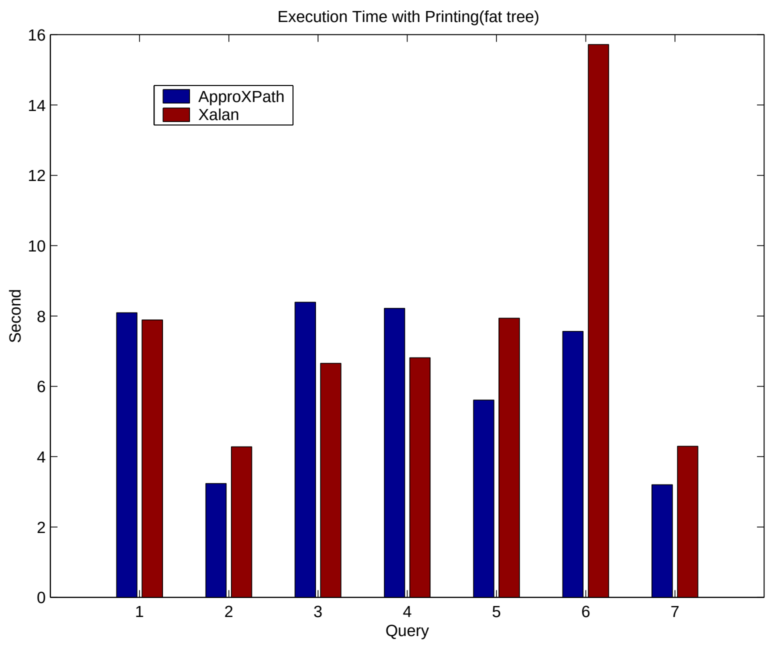
The same situation as the previous two cases can be found in figure 4.3. On those cases with result printed out, ApproXPath is still faster than Xalan on query 2, 5, 6, 7. We can notice that on query 6, ApproXPath is significantly faster than Xalan. The reason for that is ApproXPath has a better algorithm on printing out a large number of disjoint short subtrees. But ApproXPath still falls short on traveling the whole tree such as query 3.

4.3.3 Vary Number of Errors Allowed

The tests are carried out on three different trees used in the above tests: bushy tree, tall tree and fat tree. For each query, we present the data on the overall execution time, the result set size and the execution time per result node when varying the number of error allowed.

Query: /doc/level1

In figure 4.4, it seems that query “/doc/level1” does not exercise ApproXPath engine too much. The curve for total execution time remains flat when the error number is less than 5, due to the small resulting node set. The time cost per result node is relative flat. The initial fluctuation in the time per node curve is caused by the small result node set and other factor such as loading, memory allocation dominated the time cost in that case.



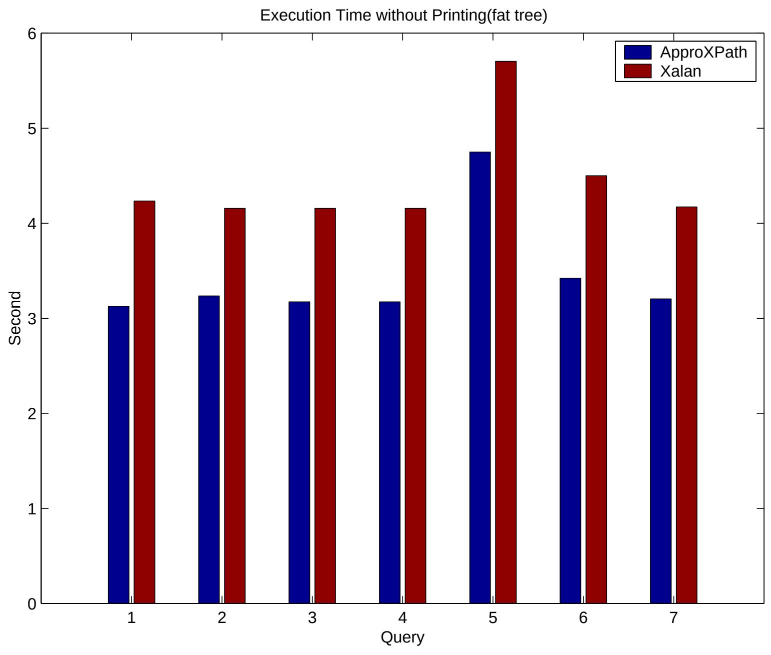
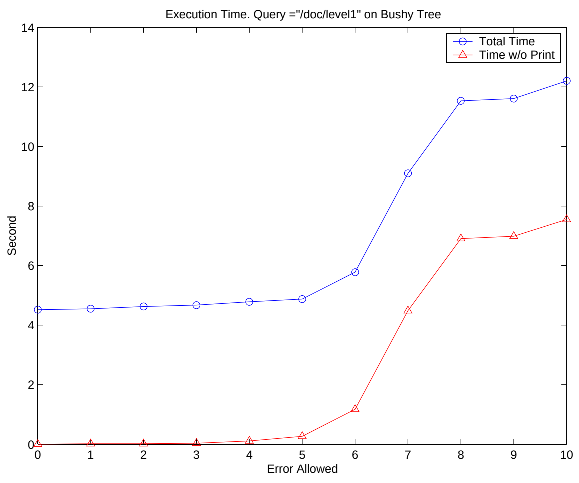
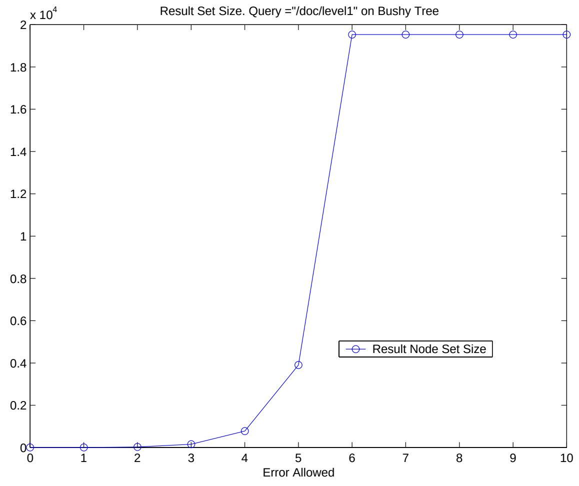


Figure 4.3: Execution time on fat tree





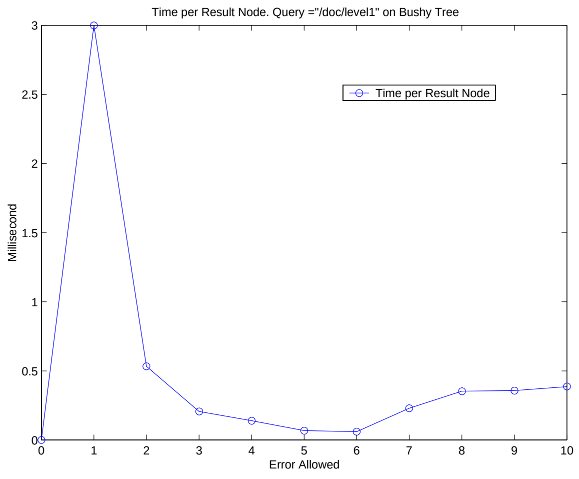


Figure 4.4: Execution time of query “/doc/level1” on bushy tree

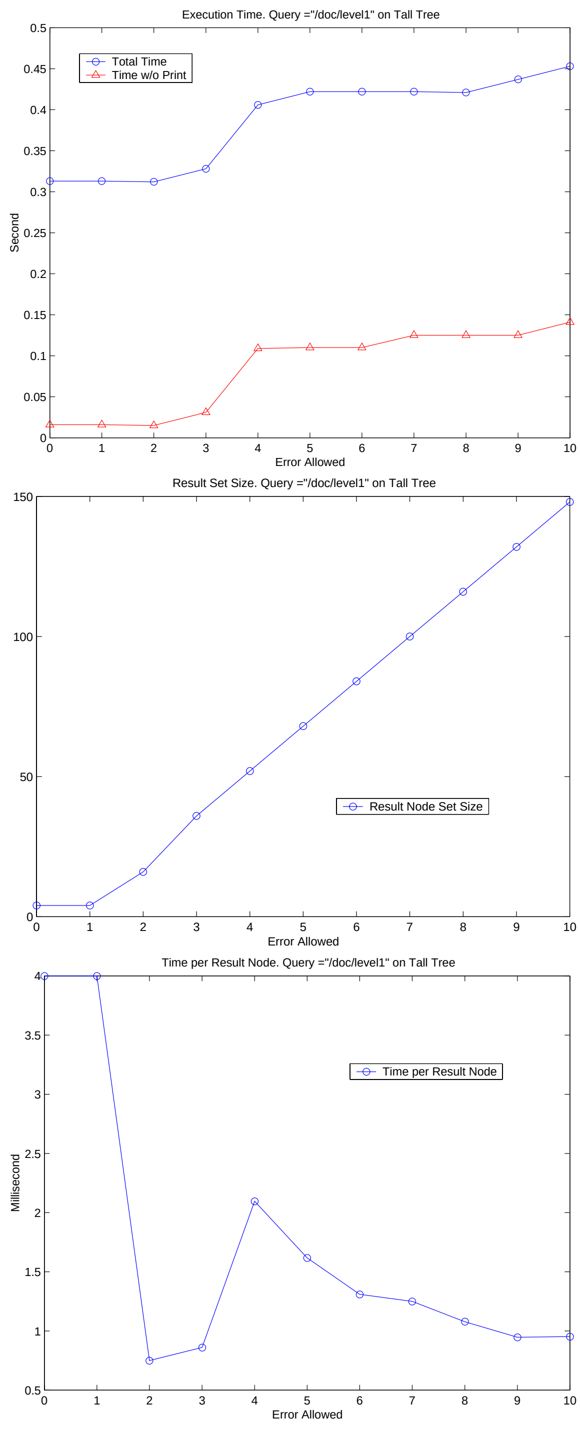


Figure 4.5: Execution time of query “/doc/level1” on tall tree

Basically, we have the same situation in figure 4.5. The size of the result node set grows linearly, since as more error allowed, the query just explores deeper and deeper in the XML tree. Also, the tree is tall enough so that it can keep on providing new nodes. The initial fluctuation in the time per node curve is caused by the small result node set and other factor such as loading, memory allocation dominated the time cost in that case.

In this case (figure 4.6), the tree is too short to provide any more nodes when more error is allowed.

Query: //level1

This test (figure 4.7) the ApproXPath engine on wildcard queries. When given more errors, //level1 just returns all nodes with name beginning with "level". As usual, the time grows linearly. The initial fluctuation in the time per node curve is caused by some irrelevant factors.

The exact same situation can be found on tall tree case (figure 4.8) and fat tree case (figure 4.9).

Query: /doc/level1/level2/level3/level4/level5

This query put some stress on the query engine, since it has a lot of steps, and there are some chores on each step. From figure 4.10, we can find that the curve is steeper than other cases, but it still remains its linearity.

The time per node curve on tall tree case (figure 4.11) is declining, if we ignore the initial couple of points. This is because there is less work on the tall skinny tree when try to evaluate "child" axis.

The time per node curve on fat tree case (figure 4.12) becomes flat when error number is growing. This is because the tree is short and *child* axis quickly runs

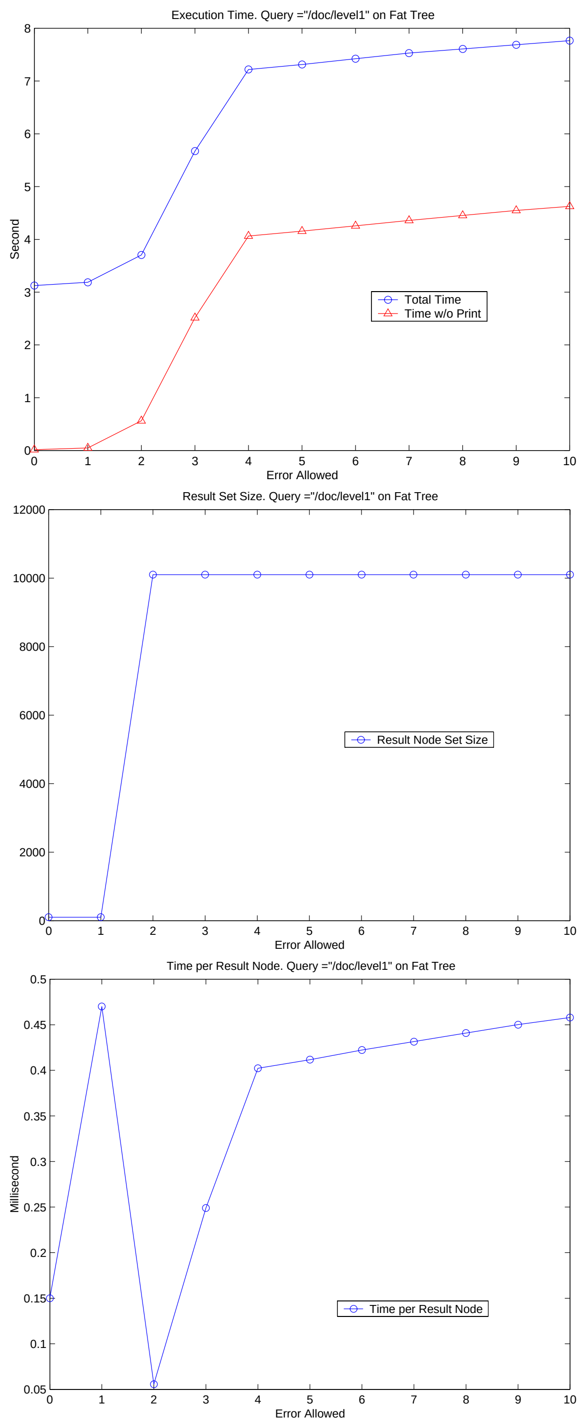


Figure 4.6: Execution time of query “/doc/level1” on fat tree

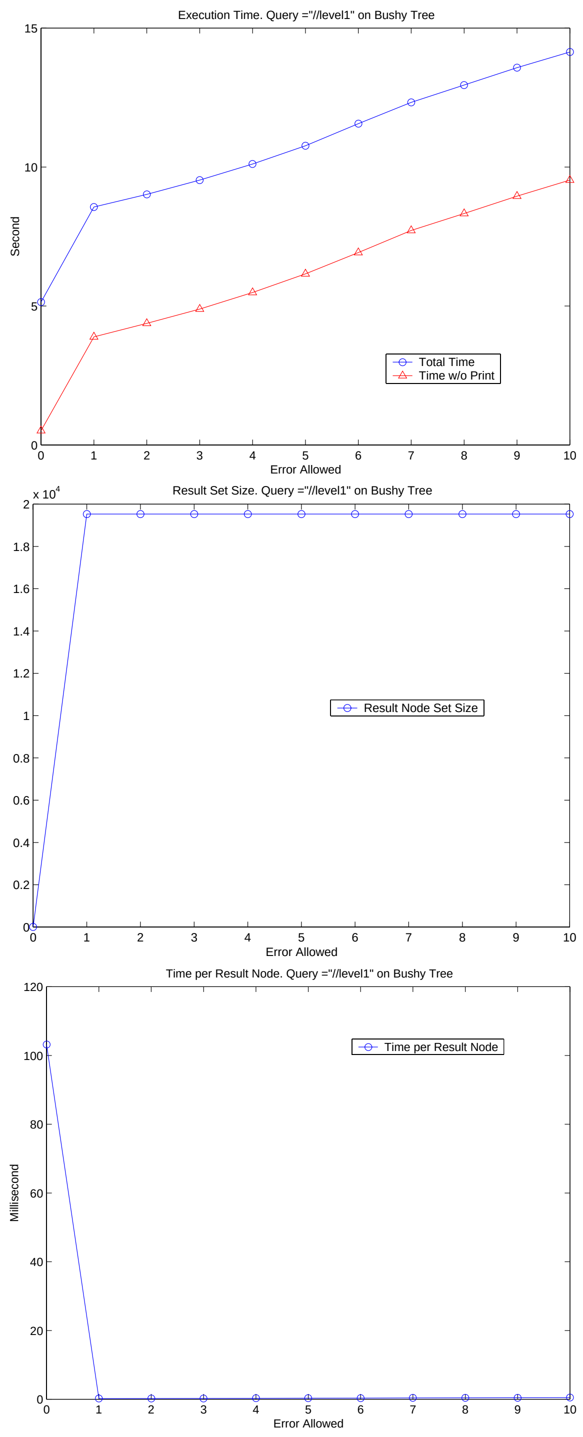


Figure 4.7: Execution time of query “//level1” on bushy tree

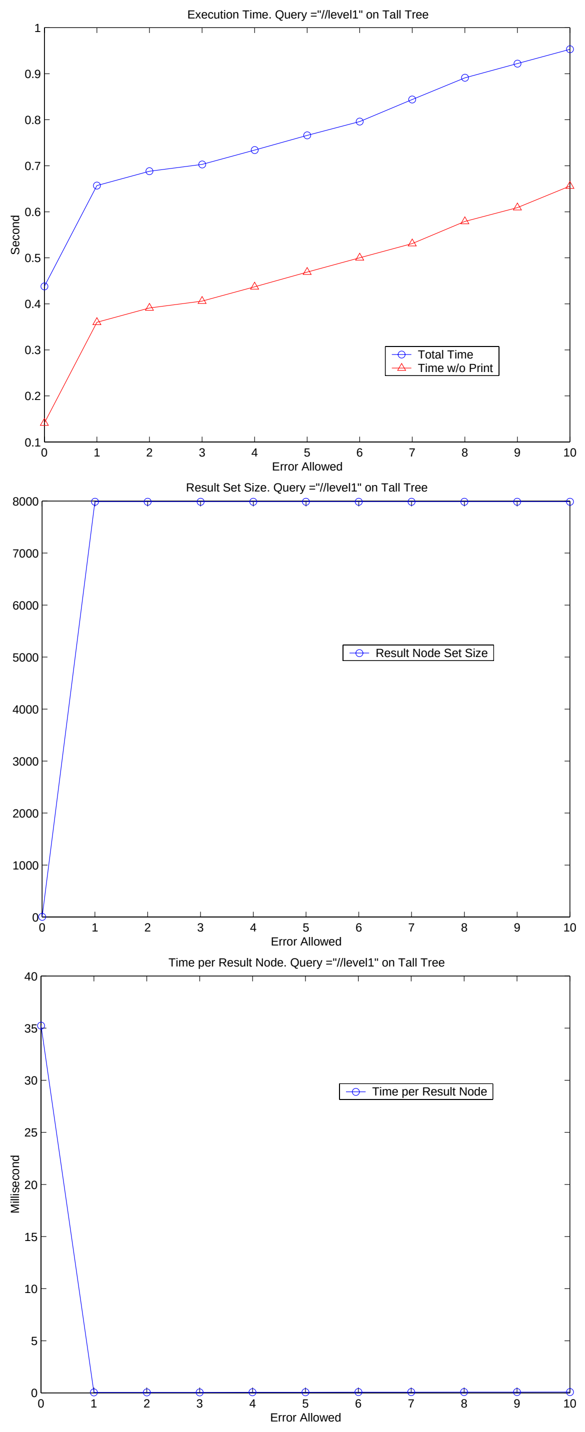


Figure 4.8: Execution time of query “//level1” on tall tree

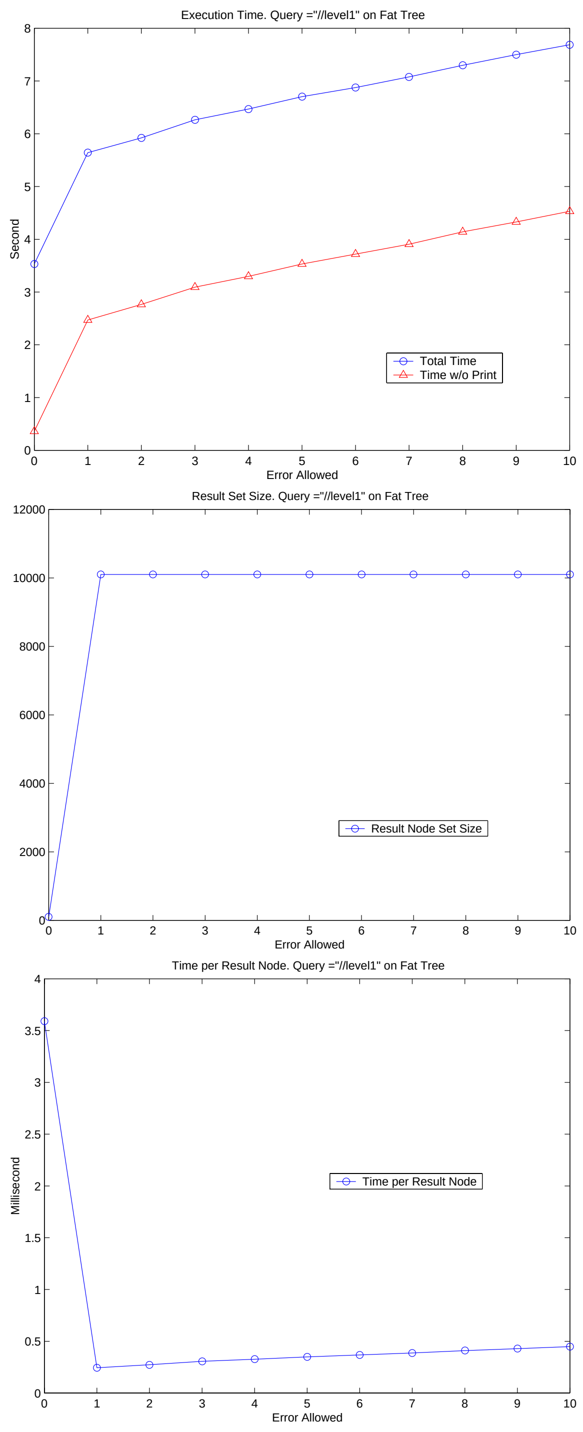


Figure 4.9: Execution time of query “//level1” on fat tree

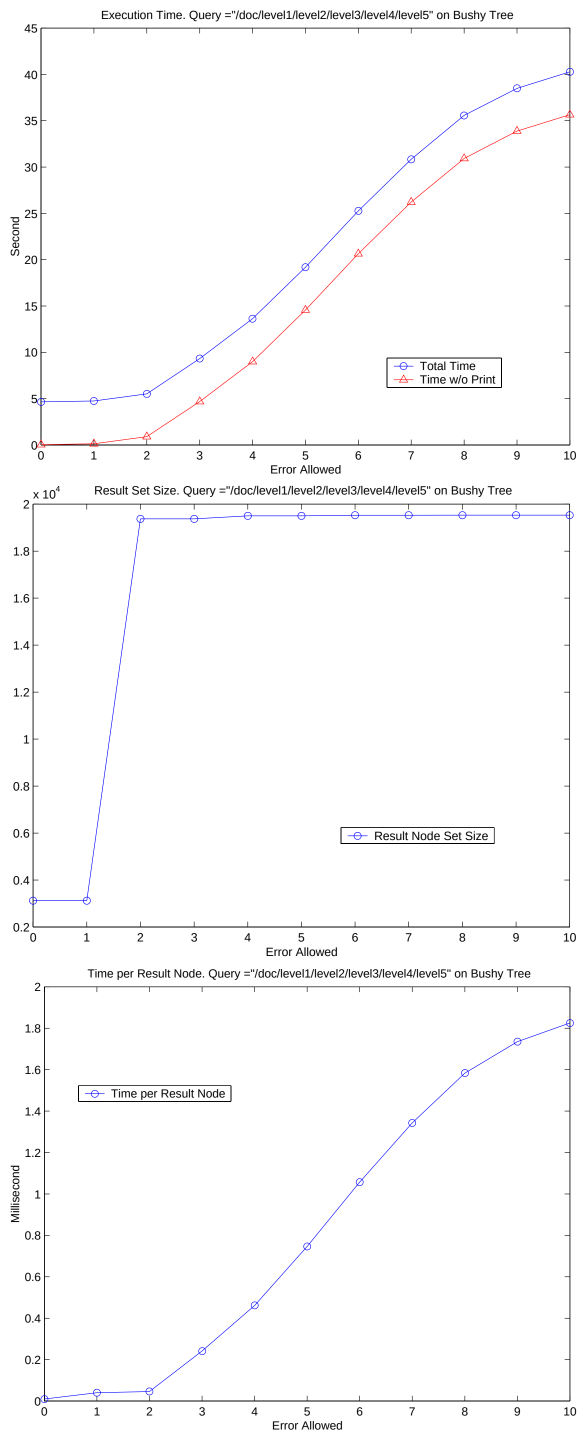


Figure 4.10: Execution time of query “/doc/level1/level2/level3/level4/level5” on bushy tree

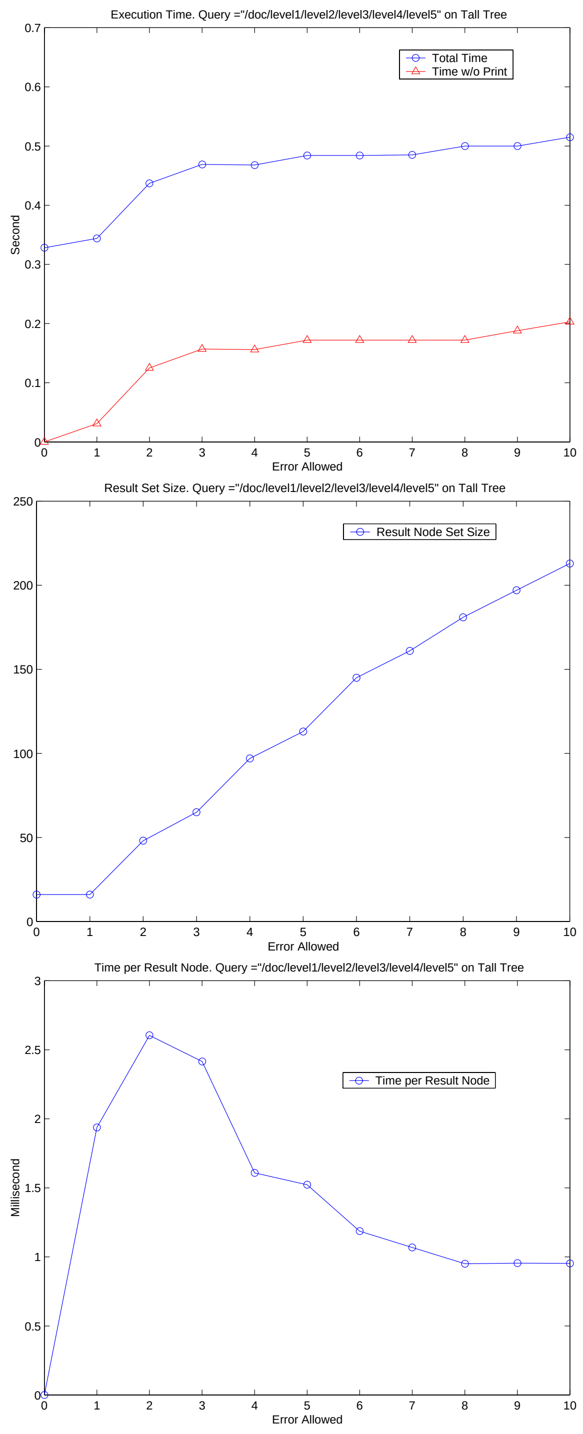


Figure 4.11: Execution time of query “/doc/level1/level2/level3/level4/level5” on tall tree

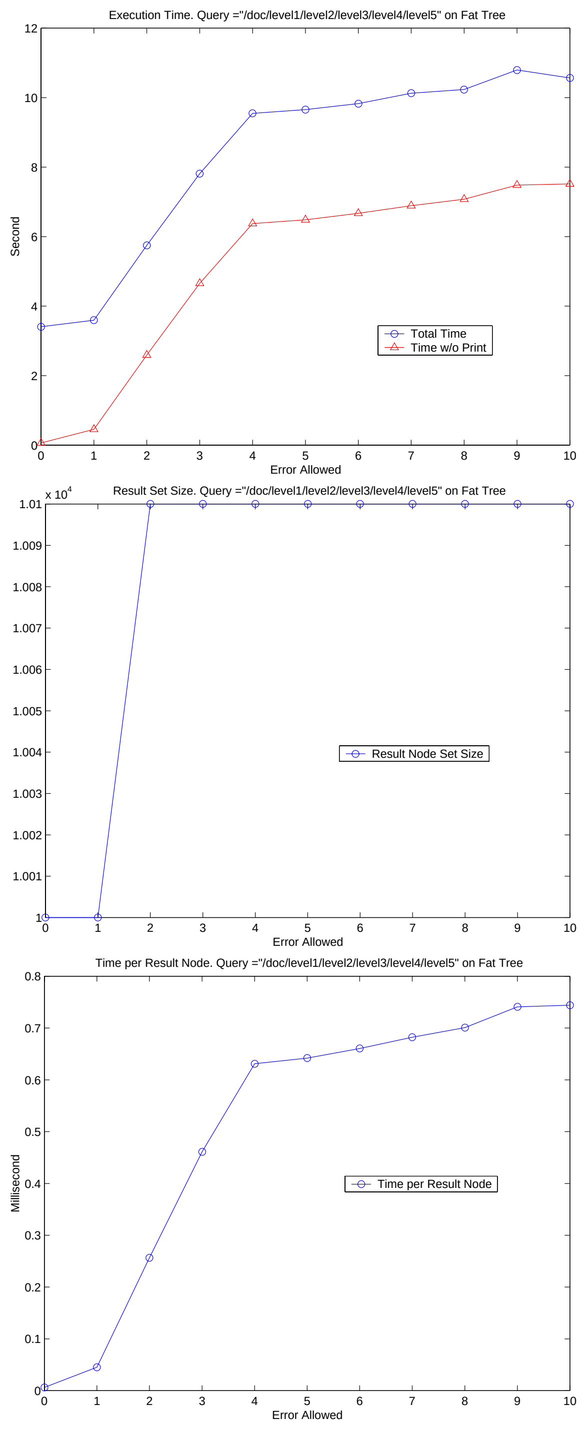


Figure 4.12: Execution time of query “/doc/level1/level2/level3/level4/level5” on fat tree

out of nodes.

Query: //name[@first='Emily']/preceding-sibling::*

This query tests the query engine on combination of wildcard, predicate and content matching. From figure 4.13, figure 4.14, figure 4.15 , we find that ApproXPath handles them very well. The execution times, total and per node, are still growing linearly.

From all the testing results, we can claim that

1. Navigation-based ApproXPath with 0 error allowed has better or similar performance when comparing to Xalan. This means there is little overhead in ApproXPath when no error is allowed.

2. Navigation-based ApproXPath scales well with the number of error allowed. In all test cases, it grows linearly with the number of error allowed at worst. This demonstrates our theoretical analysis in chapter 3 is correct.

From the experiments, it is clear shown that we can claim the ApproXPath a good approach to approximate XML query.

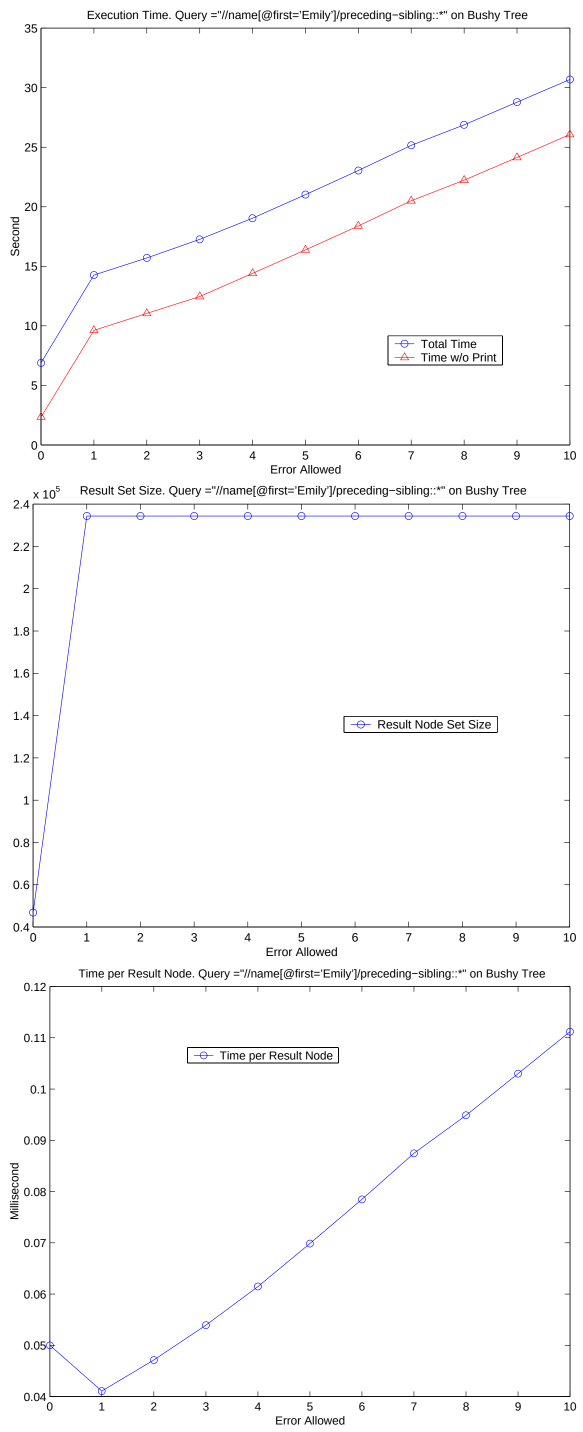


Figure 4.13: Execution time of query "//name[@first='Emily']/preceding-sibling::*" on bushy tree

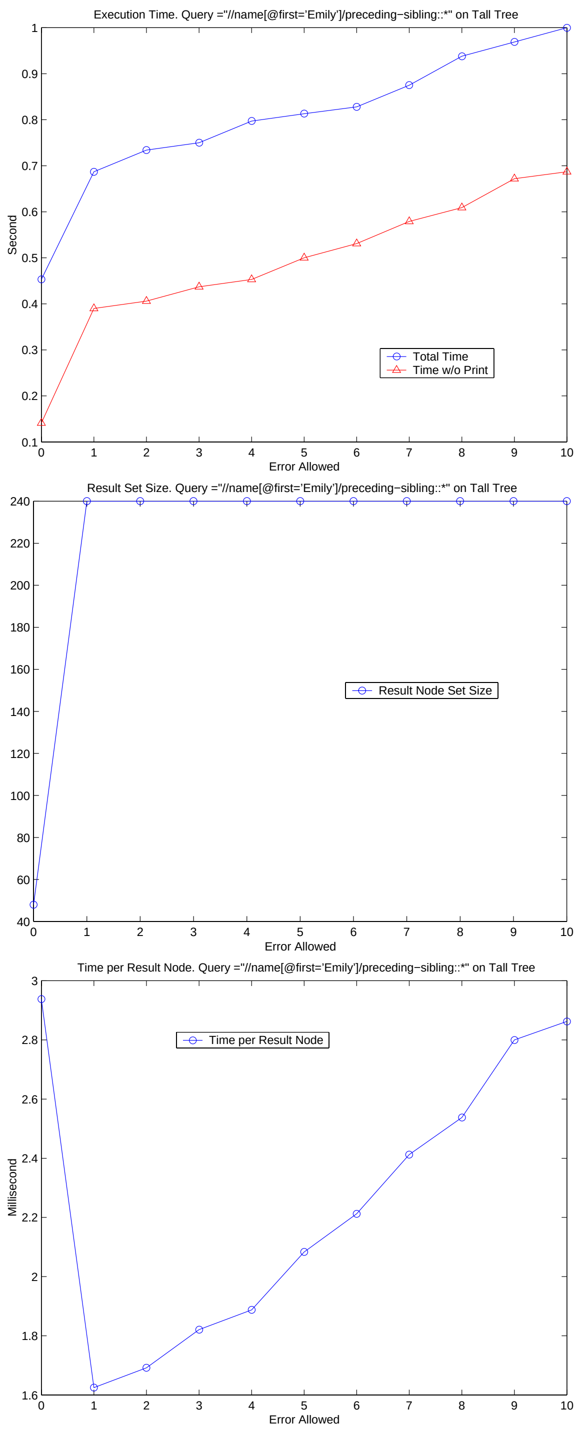
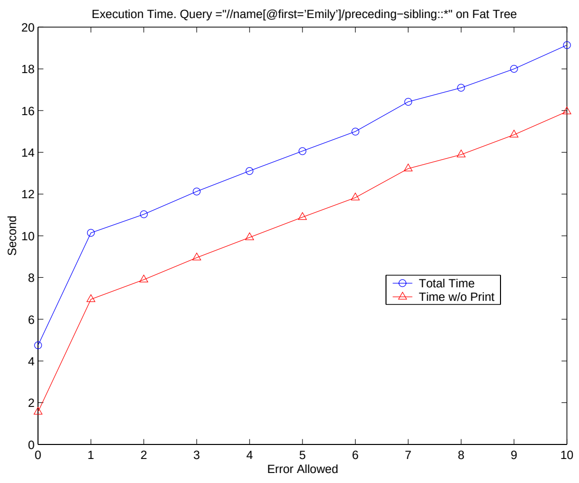
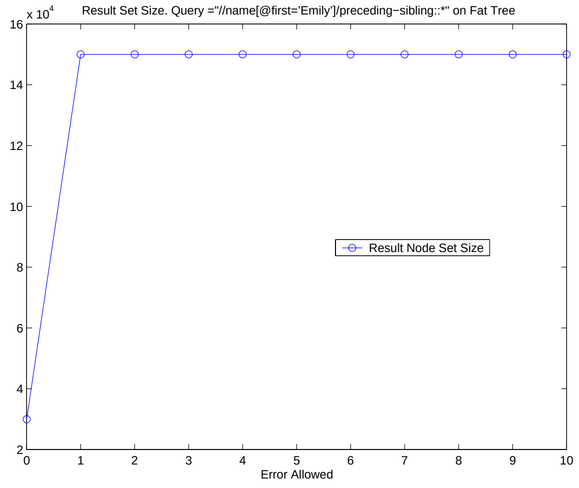


Figure 4.14: Execution time of query “//name[@first=‘Emily’]/preceding-sibling::*” on tall tree





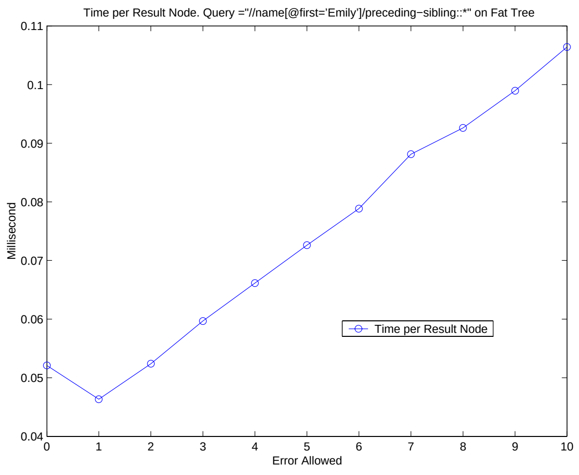


Figure 4.15: Execution time of query "//name[@first='Emily']/preceding-sibling::*" on fat tree

Chapter 5

Conclusions

Over the past several years, there has been a tremendous surge of interest in XML as a universal, queryable representation for data. This has in part been stimulated by the growth of the Web and e-commerce, where XML has almost instantly emerged as the *de facto* standard for information interchange and integration. Nearly every vendor of data management tools has added support for exporting, viewing, and in some cases even importing, XML formatted data. Tools for querying and integrating XML are still in their infancy. There are still more need to be done in the XML query area.

In this chapter, we conclude this thesis by summarizing the research and contributions discussed in the previous chapters, followed by a section on direction for future research.

5.1 Conclusions

In this thesis, we proposed an approximate XML query language. This language is compatible with existing XPath language and could be used in place where conventional XPath is used. Our approach is query relaxation based on tree distance metrics and approximate string matching. By revising conventional definition of the conventional XPath location step, we defined inexact axes, inexact node test and inexact prediction. We presented two different approaches to ApproXPath, navigation-based and index-based. The experiments show that our algorithm is linear to the error number and on par or better than Xalan when no error is allowed.

The key contributions of the thesis are:

1. An Approximate XML query language proposal, ApproXPath, is introduced. This allows users can specify XML queries with limited knowledge of XML documents precise structure, which sometimes difficult to get.

2. ApproXPath can handle both structure and content errors. Sound results are returned based on the tree edit distance and string edit distance metrics. The result is grouped by the error introduced with evaluating.

3. ApproXPath is build with full backwards compatibility with conventional XPath. It has the same syntax and semantics with conventional XPath. This allows our approximate query engine be used where conventional XPath is used. This also ease the learning curve of the user, they don't need to learn a whole new tool.

4. Two implementations are presented. Experiments show that navigation-based ApproXPath performs well.

5.2 Future Work

There is still a lot of work could be done in the approximate XML query. Since XML query itself is still in its early age, every thing is rapid changing. The language proposal for XML query is not finalized. It is highly possible that one may extend our current work to reflect the changes on the query requirement.

On the other hand, the algorithm of approximate tree matching is a hot area in both database and information retrieval communities. There is still room left for time and space complexity to improve. New algorithms keep on coming out, especially due to the current interests in the tree like structure (XML) and World Wide Web information retrieval.

In our point of view, improvement still can be made in the following areas.

1. With the stream-like processing becoming more and more important, we should make our ApproXPath implementation suitable for that situation. There are still more work to be done in that area, since it is different to handle the reverse axis in stream processing environment. There are already some researches on that area, such as revise reverse axis(not applicable to all reverse axis), using stack to store temporary XML fragments. etc. We need extend our research on that area.

2. Index for approximate string matching. Currently, the cost is still $O(n)$, which is suboptimal comparing to the traditional index on exact string matching, which is $O(log(n))$, a big difference.

Bibliography

[1] S.Abiteboul, "Querying semi-structured data," in *Proc. Intl. Conf. on Database Thoery* (F.Afrati and P.Kolaitis, eds.), no. 1187 in Lecture Notes in Computer Science, (Berlin), pp. 1–18, Springer-Verlag, 1997.

[2] S. Abiteboul, D. Suciu, and P.Buneman, *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco: Morgan-Kaufmann, 1998.

[3] D. Suciu, "Special issue on management of semistructured data," *SIGMOD Record*, vol. 26, no. 4, 1997.

[4] "XML path language (XPath) version 1.0," (http://www.w3.org/TR/xpath).

[5] Y. Diao, M. Franklin, H. Zhan, and P. Fischer, "Path sharing and predicate evaluation for high-performance XML filtering," *ACM Transactions on Database Systems (TODS)*, vol. 28, pp. 467–516, December 2003.

[6] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman, "On supporting containment queries in relational database management systems," in *SIGMOD Conference*, 2001.

[7] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel, "Structural joins: A primitive for efficient XML query pattern matching," (http://citeseer.nj.nec.com/476845.html).

[8] S. Wu and U. Manber, "Text searching allowing errors," *Communication of the ACM*, vol. 35, pp. 83–91, October 1992.

[9] R. Baeza-Yates and G. Gonnet, "A new approach to text searching," in *Proceedings of the 12th Annual ACM-SIGIR Conference on Information Retrieval*, (Cambridge, MA), pp. 168–175, June 1989.

[10] K.-C. Tai, "The tree-to-tree correcting problem," *Journal of the Assciation for Computing Machinery*, vol. 26, July 1979.

[11] D. T. Barnard, G. Clarke, and N. Duncan, "Tree-to-tree correction for document trees," Tech. Rep. 95-372, Department of Computing and Information Science, Queen's University, January 95.

[12] K. Zhang, D. Shasha, and J. T. L. Wang, “Approximate tree matching in the presence of variable length don’t cases,” *Journal of Algorithms*, vol. 16, pp. 33 – 66, January 1994.

[13] R. A. Wagner and M. J. Fischer, “The string-to-string correction problem,” *Journal of ACM*, vol. 21, pp. 168–173, Jan 1974.

[14] R. Lowrance and R. A. Wangner, “An extension of the string-to-string correction problem,” *Journal of ACM*, vol. 22, pp. 177–183, April 1975.

[15] S. Wu and U. Manber, “Fast text searching with errors,” Tech. Rep. TR91-11, Department of Computer Science, University of Arizona, 1991.

[16] G. Navarro, “A guided tour to approximate string matching,” *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[17] P. Kilpelainen, *Tree Matching Problems with Application to Structured Text Database*. PhD thesis, Department of Computer Science, University of Helsinki, November 1992.

[18] Y. Chiaramella, P. Mulhem, and F. Fourel, “A model for multimedia information retrieval,” Tech. Rep. FERMI ESPRIT BRA 8134.

[19] N. Fuhr and K. Großjohann, “XIRQL: An extension of XQL for information retrieval,” July 2000. In ACM SIGIR Workshop On XML and Information Retrieval, Athens, Greece.

[20] “Searching text-rich XML documents with relevance ranking,” (Athens, Greece), July 2000. ACM SIGIR 2000 Workshop on XML and Information Retrieval.

[21] T. Schlieder, “Similarity search in XML data using cost-based query transformations,” May 2001. ACM SIGMOD 2001 Web and Databases Workshop.

[22] A. Theobald and G. Weikum, “Adding relevance to XML,” *Lecture Notes in Computer Science*, vol. 1997, pp. 105–131, 2001.

[23] J. Robie, J. Lapp, and D. Schach, “XML query language (XQL),” 1998. Proceedings of QL’98 – The Query Languages Workshop.

[24] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, “A query language for XML,” *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 31, no. 11–16, pp. 1155–1169, 1999.

[25] C. Delobel and M. Rousset, “A uniform approach for querying large tree-structured data through a mediated schema,” 2001. International Workshop on Foundations of Models for Information Integration(FMII).

[26] Y. Kanza and Y. Sagiv, “Flexible queries over semistructured data,” 2001. Proceedings of the ACM Symposium on Principles of Database Systems.

[27] G.Salton and M.J.McGill, *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, 1983.

[28] S. Amer-Yahia, S. Cho, and D. Srivastava, “Tree pattern relaxation,” in *International Conference on Extending Database Technology (EDBT)*, 2002.

[29] H. Jin and C. Dyreson, “XPath benchmark.” School of EECS, Washington State University, Pullman, WA, 99163.