INDEXING XML DATA FOR EFFICIENT TWIG PATTERN MATCHING

by

Praveen R. Rao

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2007
As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Praveen R. Rao entitled Indexing XML Data for Efficient Twig Pattern Matching and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

__________________________________________ Date: 08/07/07
Bongki Moon

__________________________________________ Date: 08/07/07
Gregory Andrews

__________________________________________ Date: 08/07/07
Kobus Barnard

__________________________________________ Date: 08/07/07
Beichuan Zhang

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College. I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

__________________________________________ Date: 08/07/07
Dissertation Director: Bongki Moon
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: Praveen R. Rao
DEDICATION

To my mother Smt. Savithri and father Shri. Ramesh Rao...
ACKNOWLEDGMENTS

First and foremost, I thank my advisor Dr. Bongki Moon for his guidance and encouragement, and nurturing my research skills since Spring 2000 when I first took his database class. He has always been a caring person and a wonderful mentor. I thank him for being available whenever I wanted to discuss any matter. I owe him immensely for his invaluable advice throughout these years whenever I was faced with new challenges. I thank him for the freedom he gave me to choose research problems during my PhD. He has never discouraged me on any matter and has always held a keen ear to all the ideas I tossed across during our meetings, and has given me excellent directions. He has taught me the importance of clear and simple writing. I look forward to learning more from him in the years to come.

I thank Dr. Gregory Andrews for his guidance and encouragement during all my years in this department. I thank Dr. Kobus Barnard for his advice and training over the last few years, particularly during the LSST project. I thank Dr. Beichuan Zhang for his advice over the last year.

My family has played a key role in my education and scholarship. Two most important people in my life because of whom I stand this day – my parents Savithri and Ramesh Rao. Their immense sacrifice, support, encouragement, and love have been instrumental in my achievements all these years. I thank them for standing beside me during my success and failures, and praying for me! When I would be stressed out, my mother always told me: “Don’t worry. Cool it, and do it!” I thank my father for constantly encouraging me to pursue research and development right from an early age. I thank my parents and in-laws for their blessings. I thank Deepthi, Pratish, and Prashanth for their constant love, encouragement, and support. A big hug to Pratish who always prided my achievements more than I did. I thank Kalpana and Divya for their best wishes.

I thank my teachers, mentors, and colleagues over the years for their advice and help. I thank my buddies for all the jovial moments we shared together, especially at Starbucks in
the afternoons. A big thanks to the lab staff for their prompt help on numerous occasions. I thank the folks at the Academic Services and Financial and Administrative Services for their help all these years.

Finally, special thanks to all the funding agencies for providing me with financial assistance during my PhD, which include the LSST Corporation, National Science Foundation (NSF), and the State of Arizona.

Not to leave anyone out, a heart-felt thank you to all my well-wishers!
# Table of Contents

**List of Figures** ............................................................. 10  
**List of Tables** ............................................................. 12  
**Abstract** ........................................................................... 13  

**Chapter 1. Introduction** .................................................. 15  
1.1. Extensible Markup Language XML ................................. 15  
1.1.1. XML Data Model ....................................................... 16  
1.1.2. XML Query Languages ........................................... 17  
1.1.3. XML Twig Pattern Matching .................................. 19  
1.2. Focus of this Dissertation ............................................ 20  
1.2.1. Indexing and Querying XML in a Centralized Database .... 21  
1.2.2. Indexing and Querying XML in a Peer-to-Peer Database .... 23  
1.3. Summary ..................................................................... 26  

**Chapter 2. Background, Motivations, and Contributions** ......... 28  
2.1. Indexing and Querying XML in a Centralized Database ....... 28  
2.1.1. Related Work ........................................................... 28  
2.1.2. Motivations ............................................................. 31  
2.1.3. Contributions ......................................................... 34  
2.1.4. Scope .................................................................... 34  
2.2. Indexing and Querying XML in a Peer-to-Peer Database ...... 35  
2.2.1. Related Work ........................................................... 36  
2.2.2. Motivations ............................................................. 38  
2.2.3. Contributions ......................................................... 39  
2.2.4. Scope ................................................................... 40  
2.3. Summary ..................................................................... 41  

**Chapter 3. The PRIX System** ............................................ 42  
3.1. Prüfer Sequences for Labeled Trees: An Introduction ........ 42  
3.2. Transforming XML Documents into Prüfer Sequences .......... 43  
3.3. Transforming Twig Queries into Prüfer Sequences ............. 44  
3.4. Finding Twig Matches .................................................. 45  
3.4.1. Filtering by Subsequence Matching ............................ 46  
3.4.2. Refinement by Connectedness .................................. 48  
3.4.3. Refinement by Twig Structure .................................. 50  
3.4.4. Refinement by Matching Leaf Nodes ........................... 59
### Table of Contents—Continued

3.4.5. Processing `//` and `*` in Twig Queries ........................................... 59
3.5. Indexing the Document Sequences ......................................................... 61
  3.5.1. Building Prüfer Sequences ............................................................. 61
  3.5.2. Indexing Sequences Using B⁺-trees ................................................ 61
  3.5.3. Cost Analysis ................................................................................. 63
3.6. Twig Query Processing ............................................................................. 64
  3.6.1. Filtering by Subsequence Matching .................................................... 64
  3.6.2. The Refinement Phases in PRIX ......................................................... 69
  3.6.3. Extended Prüfer Sequences ............................................................... 71
  3.6.4. Extensions ....................................................................................... 73
  3.6.5. Cost Analysis ................................................................................. 78
3.7. Implementation Issues ............................................................................. 79
  3.7.1. Document Parsing ............................................................................ 79
  3.7.2. Indexing Engine ............................................................................. 81
  3.7.3. Querying Engine ............................................................................ 82
3.8. Experimental Evaluation .......................................................................... 83
  3.8.1. Experimental Setup ......................................................................... 83
  3.8.2. Performance Analysis ..................................................................... 86
  3.8.3. Evaluation of Bi-directional Subsequence Matching ............................ 96
  3.8.4. CPU Costs For the Refinement Phases ............................................ 97
  3.8.5. Summary of Performance Analysis ................................................... 98
3.9. Other Applications of Sequencing XML Documents and Twig Patterns ... 98
  3.9.1. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns ... 98
  3.9.2. SketchTree: Approximate Twig Pattern Counts over Streaming Labeled Trees 100
3.10. Conclusions ......................................................................................... 103

### Chapter 4. The psiX System ................................................................. 105
4.1. System Overview .................................................................................... 105
4.2. Signature Generation ........................................................................... 106
  4.2.1. Polynomials over Galois Field ......................................................... 107
  4.2.2. Benefits of Irreducible Polynomials ................................................. 108
  4.2.3. Signatures for XML Documents ....................................................... 108
  4.2.4. Signatures for Twig Queries ............................................................. 111
  4.2.5. Time Complexity of Signature Construction .................................... 112
  4.2.6. Representation of Signatures ............................................................ 113
4.3. Finding a Twig Match ........................................................................... 113
  4.3.1. Divisibility Property of Signatures ................................................... 114
  4.3.2. Indexing the Document Signatures .................................................. 116
# Table of Contents—Continued

4.4. Coping with a P2P Environment ........................................ 118  
4.4.1. Distributed Signature Index ....................................... 119  
4.4.2. Publisher .............................................................. 121  
4.4.3. Document Locator .................................................... 132  
4.4.4. Characteristics of the Distributed Signature Index ............. 133  
4.4.5. Failure of Peers and Load Balancing ............................. 134  
4.4.6. Joining and Leaving of Peers ..................................... 135  
4.4.7. Alternate Indexing Method for Signatures ....................... 136  
4.4.8. Handling Values in XML Documents .............................. 137  
4.5. Performance Evaluation ............................................... 138  
4.5.1. Implementation Details ............................................. 138  
4.5.2. Setup for the Experiments ........................................ 139  
4.5.3. Experimental Results ............................................... 141  
4.6. Other Applications of Polynomial Signatures and the Signature Index ........................................ 145  
4.7. Conclusions ......................................................... 146

## Chapter 5. Conclusions ..................................................... 147  
5.1. Contributions ........................................................... 147  
5.1.1. The PRIX System ................................................... 147  
5.1.2. The psiX System ................................................... 148  
5.2. Directions for Future Research ...................................... 148

REFERENCES ................................................................. 151
LIST OF FIGURES

FIGURE 1.1. An XML document example.xml ........................................... 16
FIGURE 1.2. The XML document tree for example.xml ............................ 17
FIGURE 1.3. Example twig pattern queries that can be issued by users over XML databases .................................................. 20
FIGURE 1.4. A scenario where a centralized XML database is queried by users 22
FIGURE 1.5. A scenario depicting a peer-to-peer XML database where users publish content in XML and issue queries for relevant documents .................... 23

FIGURE 2.1. An illustration of false alarms by ViST .............................. 32
FIGURE 3.1. An XML document tree, a query twig, and matches identified by the filtering phase of query processing that represent a disconnected graph and a connected graph .................................................. 43
FIGURE 3.2. Architectural overview of PRIX .......................................... 44
FIGURE 3.3. Two scenarios where nodes are assigned consecutive postorder numbers ..................... 47
FIGURE 3.4. Examples of pruning tree sequences during the refinement-by-structure phase using gap consistency ........................................... 52
FIGURE 3.5. Possible scenarios between two nodes that appear adjacent to each other in a tree sequence and the query sequences with different gaps ........ 54
FIGURE 3.6. Possible scenarios of the nodes that are being tested ............. 56
FIGURE 3.7. Data and query sequences, and the relationships between them that determines a twig pattern match ........................................ 58
FIGURE 3.8. Examples for computing the MaxGap metric for element tags ...... 66
FIGURE 3.9. Different query twig configurations that should be tested for finding unordered twig matches ........................................ 74
FIGURE 3.10. SWISSPROT - Total time - (PRIX vs TwigStackXB) ............ 87
FIGURE 3.11. SWISSPROT - I/O cost comparison - (PRIX vs TSGeneric+) ..... 88
FIGURE 3.12. TREEBANK - Total time (PRIX vs TwigStackXB) ............... 91
FIGURE 3.13. TREEBANK - I/O cost comparison - (PRIX vs TSGeneric+) .... 91
FIGURE 3.14. DBLP - Total time - (PRIX vs TwigStackXB) ..................... 94
FIGURE 3.15. DBLP - I/O cost comparison - (PRIX vs TSGeneric+) ........... 95
FIGURE 3.16. XML-based publish-subscribe system ............................. 99
FIGURE 3.17. Streaming model ................................................................. 102

FIGURE 4.1. The overview of psiX .......................................................... 106
FIGURE 4.2. Example XML documents and their structural summary graphs that they conform to ......................................................... 109
FIGURE 4.3. Example twig pattern queries that can be issued by a user ........ 112
FIGURE 4.4. Storing the signature index nodes as key-value pairs using the Chord DHT framework ......................................................... 119
LIST OF FIGURES—Continued

FIGURE 4.5. Steps involved during a node split operation . . . . . . . . . . . . . . . 128
FIGURE 4.6. Steps involved during a root split operation . . . . . . . . . . . . . . . 130
FIGURE 4.7. The characteristics of the insert and query workloads $I_W$ and $Q_W$ in
terms of the number of documents published per DTD and the number of doc-
ments located for queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
FIGURE 4.8. Cost of publishing the documents in the workload $I_W$ . . . . . . . . 141
FIGURE 4.9. Evaluation of $psiX$ in terms of average number of hops, throughput,
scalability, and precision of signatures . . . . . . . . . . . . . . . . . . . . . . . 142
FIGURE 4.10. Signature construction cost . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 3.1.</td>
<td>Notations used in this chapter</td>
<td>45</td>
</tr>
<tr>
<td>TABLE 3.2.</td>
<td>Datasets used for evaluation</td>
<td>84</td>
</tr>
<tr>
<td>TABLE 3.3.</td>
<td>XPath Queries for SWISSPROT Dataset</td>
<td>84</td>
</tr>
<tr>
<td>TABLE 3.4.</td>
<td>XPath Queries for TREEBANK Dataset</td>
<td>85</td>
</tr>
<tr>
<td>TABLE 3.5.</td>
<td>XPath Queries for DBLP Dataset</td>
<td>85</td>
</tr>
<tr>
<td>TABLE 3.6.</td>
<td>SWISSPROT - I/O cost and total time - TwigStack vs TwigStackXB</td>
<td>87</td>
</tr>
<tr>
<td>TABLE 3.7.</td>
<td>SWISSPROT - Total time - PRIX vs TwigStackXB</td>
<td>88</td>
</tr>
<tr>
<td>TABLE 3.8.</td>
<td>TREEBANK - TwigStack vs TwigStackXB</td>
<td>91</td>
</tr>
<tr>
<td>TABLE 3.9.</td>
<td>TREEBANK - PRIX vs TwigStackXB</td>
<td>92</td>
</tr>
<tr>
<td>TABLE 3.10.</td>
<td>DBLP - I/O cost and total time - TwigStack vs TwigStackXB</td>
<td>94</td>
</tr>
<tr>
<td>TABLE 3.11.</td>
<td>DBLP - I/O cost and total time - PRIX vs TwigStackXB</td>
<td>95</td>
</tr>
<tr>
<td>TABLE 3.12.</td>
<td>Improvements in disk I/O during subsequence matching using Bi-directional subsequence matching</td>
<td>97</td>
</tr>
<tr>
<td>TABLE 3.13.</td>
<td>Maximum CPU costs incurred by the refinement phases</td>
<td>97</td>
</tr>
<tr>
<td>TABLE 4.1.</td>
<td>Insert and query workloads used for evaluation</td>
<td>140</td>
</tr>
</tbody>
</table>
The Extensible Markup Language XML has become the de facto standard for information representation and interchange on the Internet. In this dissertation, I address the problem of indexing and querying XML in two environments, namely, (a) a traditional environment where data is centrally stored and (b) a growingly popular peer-to-peer (P2P) environment.

In a traditional environment, the index built over XML data is typically centralized. On the other hand, due to the distributed nature of the data in a P2P system, the index is also distributed. Due to the different models of storing data in these two environments, I propose two different XML indexing schemes for efficient query processing.

In a traditional environment, a core operation is to find all occurrences of a given query pattern in the database. I propose a new way of indexing XML documents and processing query patterns. Every XML document in the database is transformed into a sequence of labels by Prüfer’s method that constructs a one-to-one correspondence between trees and sequences. During query processing, a query pattern is also transformed into its Prüfer sequence. By performing subsequence matching on the set of sequences in the database, and performing a series of refinement phases that I have developed, all the occurrences of a query pattern can be found in the database. Furthermore, I show that all correct answers are found without any false dismissals or false alarms. I present the design, implementation, and experimental evaluation of the PRIX system that I have developed for this purpose.

Coupled with the growing popularity of P2P systems, XML is commonly used as an underlying data model for P2P applications to handle the heterogeneity of the data and limited expressiveness of queries. Locating relevant data sources across a large number of participating peers is an important challenge. In this environment, the challenge is to quickly test the existence of a query pattern in XML documents published by users rather than finding all their occurrences. PRIX finds all occurrences of a query pattern and hence is not the best solution. Moreover, in a P2P environment, a distributed and decentralized
index is necessary. Therefore, I propose a distributed indexing scheme for XML documents to quickly test for existence of query patterns based on polynomial signatures. In this scheme, each XML document is mapped into an algebraic signature that captures the structural summary of the document. The participating peers in the network collectively maintain a distributed and hierarchical index over the signatures. By virtue of the signature index, the signatures of documents with similar structural characteristics tend to be stored together at the same peer, and a search for document sources is resolved quickly. I present the design, implementation, and empirical evaluation of the $\psi$iX system that I have developed for this purpose. The signature scheme proposed in $\psi$iX can be applied to querying heterogeneous XML databases.
CHAPTER 1
INTRODUCTION

The Internet has enabled users to quickly exchange heterogeneous information (e.g., text documents, images, video clips, audio clips) via emails, web pages, electronic news, RSS feeds and so on. As large volumes of data are constantly being generated by data sources over the Internet, effective data representation and efficient large-scale data management are vital for information sharing and dissemination. Over the last decade, the Extensible Markup Language (XML) has become the de facto standard for information representation and interchange on the Internet. Numerous applications in ecommerce, Internet services, and scientific computing have adopted XML for representing their data [78]. From a research standpoint, the problem of storing, indexing, and querying XML documents has received tremendous attention from the database community. The focus of this dissertation is to develop novel and efficient techniques for indexing and querying XML data.

1.1 Extensible Markup Language XML

XML is a simple, flexible text format derived from the Standard Generalized Markup Language (SGML). XML is recommended by the World Wide Web consortium (W3C). While creating the core XML standard, the following design goals were considered [1], which have made XML a success.

- XML shall be straightforwardly usable over the Internet.
- It shall be easy to write programs that process XML documents.
- XML documents should be human readable.
- The XML design should be prepared quickly.
The design of XML shall be formal and concise.

XML documents shall be easy to create.

A sample XML document called example.xml, which is a text file, is shown in Figure 1.1. In this document, <author> denotes the opening of tag name author and </author> denotes its closing. These tag names are called elements. Each element can have a list of (attribute, value) pairs associated with it. For example, the element book has an attribute id with value “12”. Values such as text and numeric data are enclosed by opening and closing tags. Tags can be enclosed by other tags and thus allow nesting of elements in the documents thereby allowing rich structure to the data. A well-formed XML document follows the syntactic rules of XML. For example, every opening of a tag name has a corresponding closing tag.

In essence, XML is simple markup language. XML documents are human readable, can be easily created by users and applications, and can be processed easily by software. These documents are extensible as new elements and attributes can be defined as required by applications.

1.1.1 XML Data Model

While the relational data model proposed by Edgar Codd [15] is suited for structured data that is stored in tables created over well-defined schemas, XML can be used to model data
that is semistructured due to characteristics such as wide variation in structure, parts of data may lack structure (e.g., text), and rapidly evolving schemas.

An XML document is typically modeled as an ordered labeled tree and hence a natural choice for data that is inherently hierarchical. Figure 1.2 shows a tree representation of the XML document in Figure 1.1. A well-formed XML document has an element that denotes the root of the tree. Each node in this tree corresponds to an element or a value. The order among the sibling nodes in the tree is defined by the order in which their elements appear in the XML document. The order in which the elements appear in the document is called the document order. Values are represented by character data and occur at the leaf nodes. The tree edges represent a relationship between two elements or between an element and a value. Each element can have a list of (attribute, value) pairs associated with it. It is common to treat attributes in the same way as elements. (This is shown by dotted lines in Figure 1.2.) Hence, no special distinction will be made between elements and attributes in this dissertation.

1.1.2 XML Query Languages

Existing relational query languages such as SQL are not suited for querying XML. This is because the XML data is heterogeneous in structure, contains nested levels, has sparse nature of attributes, and contains an intrinsic order of elements. The XML Path Language (XPath) [7] and XML Query Language (XQuery) [8] are two popular query languages
In XPath, the structures of XML documents can be specified using a location path expression, which is a sequence of location steps to get from one node to another node in the document. Each step has

- an axis specifier between the nodes selected by the location step (e.g., ‘/’ for parent-child relationship and ‘//’ for ancestor-descendant relationship),
- a node test which specifies the name of the selected nodes,
- and zero or more predicates (specified using square brackets) to refine the set of selected nodes by a location step.

A predicate is a path expression that can be used to test for values of XML elements. A wildcard ‘*’ is a valid node test that can match any element in the document. The evaluation of an XPath expression returns a list of nodes in the input XML document that match the final location step. For example, an XPath expression

```
//book[author="Tom Sawyer"]/title
```

specifies a pattern composed of three elements, namely, book, author, and title in an XML document, and a value-based selection predicate author="Tom Sawyer". The evaluation of this expression on example.xml returns the matching title elements in the document.

XQuery is a functional language which contains various kinds of expressions that return values and have no side-affects. While XPath allows the selection of existing nodes, XQuery allows the creation of new elements and attributes and specify their contents and relationships. The expressions in XQuery are combined using constructs such as for, let, where and return. The for, let, and where constructs use XPath expressions to select document nodes. For example, an XQuery query
for $e$ in document("example.xml")/book
let $f := $e/title
where $e[.//author="Mark Twain"]
return <result> $f </result>

on evaluation will output

<result>
<title>The Adventures of Tom Sawyer</title>
</result>.

1.1.3 XML Twig Pattern Matching

Queries with XPath expressions have been one of the major foci of research for indexing and querying XML documents. A path expression can be mapped to one or more twig patterns, which is defined as follows.

**Definition 1.1.1.** A twig pattern has a tree structure with node labels denoting element names, attributes, and values. The nodes in the twig pattern have either an ancestor-descendant relationship (‘//’) or a parent-child relationship (‘/’).

For example, a twig pattern $Q_1$ (in Figure 1.3(a)) can occur anywhere in the XML document, and can be expressed as ‘//A//B[*/D]/E’ in XPath. However, this XPath expression has no restriction on the order of the sibling nodes ‘E’ and ‘*’. Using the XPath axis following-sibling, $Q_1$ can be rewritten as ‘//A//B/*[D]/following-sibling::E’ to specify that ‘E’ should follow ‘*’ in the matches. Note that an XPath expression can be mapped to more than one twig pattern. As an example, the XPath expression ‘//A[B/D] B/E’ can be interpreted as either of the two distinct twig patterns $Q_2$ and $Q_3$ shown in Figures 1.3(b) and 1.3(c).

Throughout this dissertation, twig patterns will be dealt with instead of XPath expressions. Note that the semantics of twig pattern match is different from that of a XPath
expression. A twig pattern match identifies a set of nodes in the document that match the nodes of the pattern. For example, a match of $Q_2$, identifies an instance of A, B, D, and E in the document that together have the structure denoted by $Q_2$. More than one match can appear in a document. An ordered match is a match where the order among the sibling nodes in a twig pattern match the document order. An unordered match ignores the ordering between the sibling nodes in a twig pattern.

1.2 Focus of this Dissertation

The focus of this dissertation is to develop efficient twig pattern matching algorithms by building suitable indexes over XML data. One way to process a twig query (Definition 1.1.1) is by first decomposing it into parent-child and and ancestor-descendant node pairs, and then finding their matches individually. Finally, their resulting matches are joined to obtain complete twig matches. For example, the query $Q_2$ in Figure 1.3(b) can be decomposed and processed by first finding all A/B matches, B/D matches, and B/E matches in the database. A join of these three sets of matches yields matches for $Q_2$. Alternatively, a twig query can be broken down into root-to-leaf paths. Each path can be matched individually, and then the results of individual root-to-leaf paths can be joined to output the final result.

The twig pattern matching techniques that I present in this dissertation take a holistic
approach – twig patterns are processed as a whole without breaking them into smaller units such as root-to-leaf paths and then combining the results. The advantage is that unnecessary intermediate results that do not appear in the final result can be avoided, which significantly improves the query processing time.

As part of my research, I have developed two XML twig pattern matching techniques based on novel indexing schemes for two popular environments: (a) a traditional environment where XML data is stored in a centralized database, and (b) a growingly popular environment where XML data is stored in a peer-to-peer database. For the traditional environment, a centralized index has been developed. However, for a P2P system, since the data is distributed, the indexing scheme is also distributed and decentralized in nature. Note that it is common to find all occurrences of a twig pattern in a traditional environment. However, locating relevant XML documents by testing for the existence of a twig pattern is common operation in a P2P system. Due to different models of storing data and query processing requirements in the aforementioned environments, I propose two different XML indexing schemes in this dissertation.

1.2.1 Indexing and Querying XML in a Centralized Database

Numerous applications have adopted XML as a data model for storing and retrieving data. Often, data stored in legacy databases (e.g., relational databases) is exposed to applications in XML. Below are a few examples [74]:

- DBLP Computer Science Bibliography contains bibliographic information on major computer science journals and conferences and is available in XML format.
- Protein Sequence Database contains integrated collection of functionally annotated protein sequences and is available in XML format.
- Mondial is a world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources and is also
FIGURE 1.4. A scenario where a centralized XML database is queried by users available in XML.

- NASA dataset is converted from legacy flat-file format into XML and is made available to the public.

It is typical for users to retrieve portions of the documents by posing queries. A scenario is illustrated in Figure 1.4, where XML documents are centrally stored in the database. A user issues a query (e.g., in XPath or XQuery) and the database system returns relevant portions of the XML documents to the user. Finding all occurrences of a twig pattern is a core operation over an XML database. Formally, the problem that I address in this dissertation is stated as follows.

Given a collection of XML documents and a query twig, report all occurrences of the twig in the XML documents.

I propose a new way of indexing XML documents and processing twig patterns in an XML database. I have developed the PRIX (PRüfer sequences for Indexing XML) system [60, 63, 56], where every XML document in the database can be transformed into a sequence of labels by Prüfer’s method that constructs a one-to-one correspondence between trees and sequences. During query processing, a twig pattern is also transformed into its Prüfer sequence. By performing subsequence matching on the set of sequences in the

1PRIX is pronounced without the ‘x’ like French word Grand Prix.
Figure 1.5. A scenario depicting a peer-to-peer XML database where users publish content in XML and issue queries for relevant documents.

database, and performing a series of refinement phases that I have developed, all the occurrences of a twig pattern can be found in the database. This approach allows holistic processing of a twig pattern without decomposing the twig into root-to-leaf paths and processing these paths individually. Furthermore, I show that all correct answers are found without any false dismissals or false alarms. Experimental results demonstrate the performance benefits of PRIX.

1.2.2 Indexing and Querying XML in a Peer-to-Peer Database

The peer-to-peer (P2P) model of computing has gained much attention over the years from researchers in various disciplines as well as users surfing the Internet. P2P systems now provide an infrastructure for a wide variety of applications that rely on scalable decentralized computing, distributed and redundant storage, self-organizing overlay networks, and require resilience to a dynamically changing network and adaptability to varying workloads.

The extensible markup language XML has become the popular standard for information
representation and interchange over the Internet. Coupled with the growing popularity of P2P systems, XML is commonly used as an underlying data model for P2P applications to overcome the limited expressiveness of queries and to deal with the heterogeneity of data [41]. This is a natural and synergistic combination of XML and P2P, because XML provides a means to represent, index and query semistructured data, while P2P systems enable peers to actually locate data sources across the network and share among peers.

One of the most important challenges in a peer-to-peer (P2P) network is to efficiently locate relevant data sources across a large number of participating peers. A scenario is illustrated in Figure 1.5 where XML documents are published by users in a P2P network. When a user issues a query (e.g., in XQuery), the system locates all the publishers of documents relevant to the query and reports them to the user. (The circled users denote the publishers returned for the query.)

Consider, for example, a P2P network where users publish bibliographic information in XML format. Suppose a user issues the following XQuery query to find all authors who have published a book entitled “XML & P2P”.

```xml
for $d in collection("P2P")
let $e := $d/book/author
where $d/book/title="XML & P2P"
return <name>$e/text()</name>
```

Within the P2P network, this type of query can be processed in two phases. In the first phase, all the peers should be located who store any XML document that matches the path expression /book/title="XML & P2P". In the second phase, either the user can pull all the matching documents or send the XQuery query to the selected peers. The XQuery query is actually processed over the matching XML documents and a list of authors are generated.

The past few years have witnessed tremendous research endeavors aiming at developing efficient query processing strategies for XML data. In the second phase of the ex-
ample above, processing the XQuery query can be done efficiently by applying one of the many query processing strategies reported in the literature (e.g., XISS [43], TwigStack [11], TSGeneric+ [36], PRIX [60]). Therefore, my research focuses on the first phase of the bibliography scenario, and addresses the problem of locating all relevant XML documents across an Internet-scale P2P network. Formally, the problem that I address in this dissertation is stated as follows.

Let $P = \{p_1, p_2, ..., p_n\}$ denote a set of peers. Suppose each peer publishes a set of XML documents and let $D$ denote all the documents published by peers in $P$. For a twig query $q$ issued by some peer $p_i$, locate, by best-effort, the set of XML documents $D' \subseteq D$ such that $q$ has a match in each document in $D'$. For each document in $D'$, also identify the peer that published it.

The notion of best-effort service is adopted, as is traditionally done in the Internet regarding the delivery of network traffic. In the problem statement, I mean by best-effort that a lookup is expected to return as much accurate information of peer locations as possible without guaranteeing that the result is complete.

One of the main challenges posed by this problem is to organize and index XML data in a distributed way such that the data sources storing relevant XML data can be located by contacting as a small number of hosts in the network as possible. A document is relevant to a user if it has at least one match for the input twig pattern. This is a striking difference from the twig pattern matching problem in a traditional environment where all twig occurrences are reported. Due to a distributed model of computation and a different query processing requirement, PRIX is not suitable for a P2P environment.

I have developed a new system called psiX [62] (polynomial signature index for XML) that runs on top of an existing distributed hashing framework. Under my system, each XML document is mapped into an algebraic signature that captures the structural summary of the document. The participating peers in the network collectively maintain a distributed and hierarchical index over the signatures. By virtue of the signature index, the signa-
tures of documents with similar structural characteristics tend to be stored together at the same peer, and a search for document sources is resolved quickly. The experimental study demonstrates that $\psi iX$ provides an effective and scalable location service for a wide variety of XML documents in the P2P environments.

1.3 Summary

In this dissertation, I present two efficient XML twig pattern matching techniques based on novel indexing schemes that I have developed for (a) a traditional environment where XML data is stored in a centralized database and (b) a growing popular environment where XML data is stored in a P2P database. These two environments differ in the model of storing data and query processing requirements. Hence, the indexing scheme for the traditional environment is centralized in nature and allows finding all occurrences of a twig pattern. However, this scheme is not suitable for a P2P environment, and thus a distributed and decentralized indexing scheme has been proposed to quickly test the existence of a twig pattern in XML documents.

For the traditional environment, I propose a new paradigm for twig pattern matching by transforming XML documents and twig patterns into sequences based on Prüfer’s method. By performing subsequence matching over the document sequences and a series of refinement phases, all occurrences of a twig pattern are found. The design of PRIIX is described in Chapter 3.

For the P2P environment, I have developed a signature scheme that captures the structural summary of the document. I have also developed a distributed signature index that stores the document signatures. By virtue of the signature index, the signatures of documents with similar structural characteristics tend to be stored together at the same peer, and a search for document sources is resolved quickly. The design of $\psi iX$ is described in Chapter 4.

The rest of the dissertation is organized as follows. The related work, motivations, and
contributions of my research are described in Chapter 2. The design of PRIX is described in Chapter 3. The design of psiX is described in Chapter 4. Chapter 5 summarizes the contributions of this dissertation. In addition, directions for future research are discussed in this chapter.
CHAPTER 2
BACKGROUND, MOTIVATIONS, AND CONTRIBUTIONS

In this chapter, I present the background and motivations for my research work. I describe the related work and motivations for indexing and querying XML in a traditional environment where data is centrally stored, followed by the main contributions of my work to address this issue. After that, I describe the related work and motivations for indexing and querying XML in a popular peer-to-peer environment, followed by the main contributions of my work to address this issue. Note that the model of storing data and query processing requirements in these two environments are different.

2.1 Indexing and Querying XML in a Centralized Database

Since the extensible markup language XML emerged as a new standard for information representation and exchange on the Internet [10], the problem of storing, indexing and querying XML documents has been among the major issues of database research. Queries with path expressions have been one of the major foci of research for indexing and querying XML documents. Finding all occurrences of a twig pattern is one of the core operations over XML databases.

2.1.1 Related Work

In the past few years, there have been two main thrusts of research activities for processing path join queries for retrieving XML data, namely, approaches based on structural index and numbering schemes.
Structural Indexes  The approaches based on the structural index facilitate traversing through the hierarchy of XML documents by referencing the structural information of the documents (e.g., dataguide [29], representative objects [51], 1-index [49], approximate path summary [39], F&B index [38]). These structural indexes can help reduce the search space for processing linear path and twig queries.

Path join algorithms based on building structural indexes were proposed in this regard. The Lore system [29, 48] addressed several issues in query processing. DataGuide [29] provides concise and accurate summaries of semistructured databases. McHugh et al. [48] addressed different aspects of cost-based query optimizer for XML. The concept of representative objects was proposed by [51] for concise representation of the structure of semistructured, hierarchical data. Index Fabric [17] stores encodings of paths in a structure based on Patricia tries to support simple XML path queries in a single lookup. The T-index mechanism supports query processing for path expressions [49]. In addition, a family of approximate structural summaries called A(k)-indices was proposed by [39] for evaluating path expressions. The F&B index [38] was also proposed for evaluating branching path queries.

Numbering Schemes  The other class of approaches are based on a form of numbering scheme that encodes each element by its positional information within the hierarchy of an XML document it belongs to. Most of the numbering schemes reported in the literature are designed by a tree-traversal order (e.g., pre-and-postorder [22], extended preorder [43]) or textual positions of start and end tags (e.g., containment property [83], absolute region coordinate [81]). If such a numbering scheme is embedded in the labeled trees of XML documents, the structural relationship (such as ancestor-descendant) between a pair of elements can be determined quickly without traversing an entire tree. Several join algorithms have been developed to take advantage of this extraordinary opportunity to efficiently process path and twig queries [2, 11, 14, 31, 43, 83]. In particular, it has been shown that PathStack and TwigStack algorithms [11] are optimal for processing path and twig queries.
in that the processing cost is *linearly* proportional to the sum of input data and query results.

Zhang *et al.* [83] addressed efficient processing of containment queries in relational database systems. The XISS system [43] decomposes a complex path expression into a collection of basic path expressions and processes them. The nodes in the XML document trees are numbered in extended preorder. The Dewey numbering scheme was proposed to store and query ordered XML in relational databases [71]. [2] developed structural join algorithms *Tree-merge* and *Stack-tree* for matching parent-child and ancestor-descendant structural relationships. Bruno *et al.* [11] proposed *PathStack* and *TwigStack* algorithms and showed that they were I/O and CPU optimal for a large class of query twig patterns. In addition, a modification of B+-trees called XB-Trees were developed to speed up query processing. In order to quickly find ancestors of a node in XML documents during structural joins, Jiang *et al.* [36] proposed the XR-Tree index. Subsequently, they developed the TS Generic+ algorithm [37] that used indexes to speed up the twig pattern matching. Grust *et al.* [32] proposed *Staircase join* to speed up XPath processing using relational storage. Similar to PathStack, Staircase join exhibits optimal linear behavior during path processing. ORDPATHs were proposed as a labeling scheme, that are insert-friendly, to store XML in a relational database [52]. Wu *et al.* [77] proposed a prime number labeling scheme for indexing XML with support for order-sensitive queries and updates. Recently, Zezula *et al.* [82] proposed the use of tree signatures for unordered XML pattern matching, and Wang *et al.* [75] proposed an indexing method called ViST that uses subsequence matching for processing twig queries by mapping trees to sequences. Recently, TJFast [46] was developed that proposed an extended Dewey numbering scheme for indexing XML together with a holistic twig join algorithm based on TwigStack. Inspired by the PRIx system, Prasad *et al.* [55] and Tatikonda *et al.* [72] use variants of Prüfer sequences to avoid the refinement phases in PRIx and further improve the query performance. Tatikonda *et al.* [72] modify the longest common subsequence algorithm to perform subsequence matching.
2.1.2 Motivations

Below I briefly describe two of the recent contributions made for XML pattern matching: TwigStack [11] and ViST [75]. I discuss some of their drawbacks to motivate our proposed approach.

**TwigStack Algorithms** Bruno et al. [11] proposed optimal XML pattern matching algorithms. These stack-based algorithms process input lists of element instances for tags that appear in a query twig. TwigStack and PathStack algorithms operate on the positional representation of the element instances to find twig matches. A variant of TwigStack (denoted hereinafter by TwigStackXB) uses XB-Trees to speed up the processing when the input lists are long. The XB-Trees are useful in skipping sections of the input lists without missing any matches.

However, there are some limitations of TwigStackXB. The effectiveness of skipping data depends on the distribution of the matches in the input lists. If the matches are scattered all over the dataset, then the TwigStackXB algorithm drills down to lower regions of the XB-trees (including leaves) in order to avoid missing matches. Another drawback of TwigStack and TwigStackXB is that it suffers from sub-optimality for parent-child relationships in a query twig. The algorithm might produce a partial match of a path of a twig that cannot be combined with any other partial match of another path of the twig. For example, consider a query twig with 3 nodes P, Q and R where nodes Q and R are child nodes of P. The algorithm will match a pattern in the data where P is a common ancestor of Q and R but is not their parent. This match will be discarded in the merge post-processing step of the algorithm. However, the cost of post-processing may not always be trivial.

**ViST** Wang et al. [75] proposed a new method called ViST that transforms XML data trees and twig queries into structure-encoded sequences. The structure-encoded sequence is a two-dimensional sequence of (symbol, prefix) pairs \{(a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)\} where \(a_i\) represents a node in the XML document tree, and \(p_i\) represents the path from the
D = (P, e) (Q, P) (T, PQ) (Q, P) (S, PQ) 

D1 = (P, e) (Q, P) (T, PQ) (S, PQ) (R, P) (U, PR) (T, PR) 
D2 = (P, e) (Q, P) (T, PQ) (Q, P) (S, PQ) 
Q = (P, e) (Q, P) (T, PQ) (S, PQ) 

Figure 2.1. An illustration of false alarms by ViST

root node to node a1. The nodes a1, a2, . . . , an are in preorder. ViST performs subsequence matching on the structure-encoded sequences to find twig patterns in the XML documents. These sequences are stored in a disk-based virtual trie built using B+-trees.

One of the imminent drawbacks of the tree transformation used by ViST is that the worst-case storage requirement for a B+-tree index named D-Ancestorship index is higher than linear in the total number of elements in the XML documents. For example, consider a tree with n nodes and maximum depth d. In this case, the total size of the structure-encoded sequence of the tree is O(nd). Thus the D-Ancestorship index requires O(nd) space to store all the (symbol, prefix) keys. Another drawback of ViST is that the query processing strategy by straightforward subsequence matching may result in false alarms. Figure 2.1 illustrates such a case. The structure-encoded sequence of the query twig Q is a subsequence of the structure-encoded sequence of Doc1 and Doc2. However, the twig pattern Q occurs only in Doc1, and the match detected in Doc2 is a false alarm. Such false matches could be discarded by adding a post-processing step that examines the input document trees.

Our Motivations The key motivations of our work are (1) to develop a method that allows holistic processing of twig queries without breaking a twig into root-to-leaf paths and processing them individually, (2) to transforms trees into sequences, thereby reducing the problem of finding twig matches to that of finding relevant subsequences and (3) to
construct a method for finding twig matches with no *false alarms or false dismissals*.

I propose a new way of indexing XML documents and finding twig patterns in an XML database. I have developed a system called **PRIX** (*PR"ufer sequences for Indexing XML*) for indexing XML documents and processing twig queries.1 In PRIX, every XML document in the database is transformed into a sequence of labels by Pr"ufer’s method that constructs a one-to-one correspondence between trees and sequences. During query processing, a twig pattern is also transformed into its Pr"ufer sequence. By performing subsequence matching against the indexed sequences in the database, and by performing a series of filtering and refinement phases that I have developed, all the occurrences of a twig pattern can be found in the database. This work was developed independently of and differs considerably from the indexing method called ViST [75], which also converts trees into sequences.

Furthermore, most of the previous approaches (e.g., TwigStack [11], TSGeneric+ [37]) do not attempt the problem of *ordered twig pattern matching* that is useful in applications where the twig pattern nodes follow the document order in XML. For example, an XML data model was proposed by Bow et al. [13] for representing interlinear text for linguistic applications, which is used to demonstrate various linguistic principles in different languages. The XML model provides a four-level hierarchical representation for the interlinear text, namely, text level, phrase level, word level and morpheme level. For the purpose of linguistic analysis, it is essential to preserve linear order between the words in the text [76]. Thus, there is a compelling need for ordered twig pattern matching. In addition to interlinear text, language treebanks have been widely used in computational linguistics. Treebanks capture syntactic structure of textual data and provide a hierarchical representation of the sentences in the text by breaking them into syntactic units such as noun clauses, verb phrases, adjectives and so on. A recent paper by Muller et al. [50] used ordered pattern matching over treebanks for question answering systems. By design, PRIX supports ordered twig pattern matching.

1PRIX is pronounced without the ‘x’ like French word Grand Prix.
2.1.3 Contributions

The main contributions of this research work are summarized as follows.

- A new idea of transforming XML documents into sequences by Prüfer’s method has been developed. I show that twig matches can be found by performing subsequence matching on the set of sequences and by performing a series of refinement phases. I also show that our approach returns correct answers without any false alarms and false dismissals.

- My approach allows holistic processing of twig queries without breaking a twig into root-to-leaf paths and processing them individually. Additionally, my tree-to-sequence transformation guarantees a worst-case bound on the index size that is linear in the total number of nodes in the XML document trees.  

- Ordered twig pattern matching that is useful for applications that require the twig pattern nodes to follow the document order in XML is supported.

- I have developed effective optimizations to speed up the subsequence matching phase during query processing.

2.1.4 Scope

The scope of this research work is summarized as follows.

- This work targets twig pattern matching which is a fundamental problem as compared to the evaluation of XPath expressions that are used in practice. As mentioned in Section 1.1.3, the semantics of XPath evaluation is different from the semantics of a twig pattern match. Thus XPath evaluation can be faster since only the matching occurrences of a particular node label are output, and predicates are tested for their truth value.

---

2In contrast, ViST [75] does not guarantee linear space requirement for its tree-to-sequence transformation.
This work focuses on ordered twig pattern matching, and does not deal with XPath axes such as ancestor and ancestor-or-self.

PRIX can handle updates but was not designed to be update-efficient.

The design, implementation, and evaluation of PRIX is presented in Chapter 3.

2.2 Indexing and Querying XML in a Peer-to-Peer Database

The recent advances in peer-to-peer (P2P) computing has drawn much attention from researchers in various disciplines as well as users surfing the Internet. Although early P2P systems such as Napster and Gnutella made their debut as a music file sharing application and are still best known for similar file sharing services, P2P systems are gaining popularity rapidly in entrepreneurial sectors and scientific communities. For example, a project called SEED aims to provide a suite of tools for biologists working in distributed teams so that they can quickly annotate new genomes by sharing and synchronizing data in the peer-to-peer environment [53].

Such P2P systems now provide an infrastructure for a wide variety of applications that increasingly rely on decentralized computing, distributed and redundant storage, and self-organizing overlay networks. Unlike traditional client-server based systems, where server capabilities can become a bottleneck with an increasing amount of workload, P2P systems can enable all participating peers to share and distribute workload across the network in a cooperative and autonomous fashion. Thus, large-scale or Internet-scale applications running on a P2P system can adapt gracefully to the workload fluctuation and a dynamic set of peers joining and leaving the network at any time.

Only in a few years since adopted by the World Wide Web Consortium, the extensible markup language (XML) has emerged as the lingua franca for information representation and exchange over the Internet. Coupled with the growing popularity of P2P systems, XML is commonly used as an underlying data model for P2P applications to overcome the
limited expressiveness of queries and to deal with the heterogeneity of data [41]. This is a
natural and synergistic combination of XML and P2P, because XML provides a means to
represent, index and query semistructured data, while P2P systems enable peers to actually
locate data sources across the network and share among peers.

2.2.1 Related Work

In order to find any useful information efficiently in the Internet-scale P2P networks, queries
should be directed only to sources that store relevant information so that the amount of
wasted resource is minimized. Providing an efficient and scalable location service is thus
among the most critical challenges that one has to cope with to build a distributed data
repository on a P2P network because of the dynamic nature of the network.

The early P2P systems supported keyword searches and were unstructured in nature
(e.g., Gnutella, Napster). Structured P2P systems were proposed subsequently with support
for keyword searches while providing load balancing and logarithmic hop routing (e.g.,
Chord [69], Pastry [65], Tapestry [84], CAN [64]).

The increased use of XML as an underlying data model for P2P applications poses
yet another challenge. In traditional P2P systems, file or document names are used as
keys, and hashed into a distributed lookup framework of the network. However, for the
P2P applications that adopt XML to represent data, a lookup strategy exploiting document
content is more desirable than lookup by names, because the query expressiveness can
be considerably improved by posing queries in such a language as XPath and XQuery.
For content-based full-text search, Tang et al. have developed a P2P information retrieval
system call pSearch [70], in which document semantics are computed by the latent semantic
indexing (LSI) [20] in a vector space. Although the LSI is an effective method for keyword-
based information retrieval, it is not well-suited to capture the hierarchical structure of
XML documents.

A distributed catalog framework is proposed for locating XML data sources in the P2P
networks [25]. Under this framework, element tags are chosen as keys. For each distinct tag, all unique element paths leading to the key are stored in its data summary. All the key-summary pairs are stored in a distributed hash table (DHT) in a similar way that keys and instances are organized into an inverted list. For a given XPath path expression, the element tag at its leaf position is used as a lookup key to the DHT. Instead of element tags, paths can be used as keys. XP2P [9] also builds on a distributed hashing framework and allows peers to store whole or fragments of XML documents locally whose path expressions are encoded into a distributed hash table. Path expressions are encoded by Rabin’s fingerprinting method [58]. The concatenation property allows the fingerprints to be computed incrementally and facilitates fragment lookup. Garces et al. [27] proposed building a hierarchy of indexes using a DHT containing query-to-query mappings such that a user can look up more specific queries for a given broader query, thereby refining his or her interests.

XPeer [66] is an XML database system that has been developed to manage resource descriptions in the GRID environments. Unlike the aforementioned approaches, this system is built on a hybrid P2P system, where designated peers (known as superpeers) take up extended responsibilities for the clusters of peers that they belong to. Peers export a summary of their XML data in the form of tree-shaped DataGuide [29]. Locating data sources is performed in a hierarchical way by matching a query twig with summaries by traversing the superpeers organized to form a tree until any interesting location is detected.

Skobeltysn et al. [67] proposed a path based distributed index for XPath queries. Paths are hashed to obtain keys that are then stored in a structured overlay. Path queries with ‘//’ and ‘*’ are processed by first performing a lookup on a query fragment that only contains ‘/’ axis, and then issuing a broadcast to certain other peers to match the remaining portion of the query. Recently, BATON [35], a balanced tree overlay structure, was proposed to support exact match and range queries. Such a hierarchy was built since a linear ordering of the data was possible. However, an ordering of XML documents does not aid the process of twig matching to locate relevant documents. Therefore, BATON cannot be directly used
for locating XML documents.

P2P index structures have been proposed for multi-dimensional data [45, 26]. In these approaches, the entire multi-dimensional space is partitioned and merged as peers join and leave the P2P system similar to CAN [64]. The approach proposed by Liu et al. [45] relies on the existence of more powerful nodes called super-peers in the system. Although, R-trees were used, the index is not completely decentralized as compared to the index structure developed for psiX. In psiX, the signature space cannot be mapped to multi-dimensional space directly, and hence these approaches cannot be used.

2.2.2 Motivations

Most of the previous work uses a form of path summarization method to index XML documents and locate them across a P2P network. There are a few limitations of the previous work. If path summaries are grouped by leaf tags and stored as a form of inverted list in the DHT, then the cost of lookup may be substantial, particularly when identical tags appear in different documents published by different users (presumably using different schemas). The workload may be imbalanced among peers, if there are a few tags commonly used by many different users or schemas. If lookup is done for a twig query with at least one branching node in its expression, then the twig should be broken to multiple linear paths, and the results from the linear paths should be intersected to find relevant peers. This can potentially increase both the network traffic and the number of node hops, especially when the number of false positives is large. The cost of publishing an XML document to a P2P network may also be increased, because a DHT insert operation needs to be performed for each distinct tag or path in the document. To the best of our knowledge, none of the previous approaches allow holistic processing of twig queries.

These shortcomings have motivated us to build the psiX system. As opposed to the approaches based on inverted lists, psiX adopts a holistic approach for XML lookup by considering all the tags in a given twig query as a whole without breaking the twig into
multiple linear paths. XML documents are mapped to algebraic signatures, which are organized into a hierarchical signature index. The structural similarity of different XML documents can then be measured by applying low-cost algebraic operations to their signatures. Unlike a tree- or graph-based structural summary (e.g., DataGuide [29]), expensive operations such as edit-distance computations can be avoided, because the algebraic signatures are amenable to more efficient operations required to measure structural similarity. A twig query, which may contain ‘//’ and ‘*’, is mapped to algebraic signatures, and relevant documents published by peers are located by searching the index holistically without breaking the twig into multiple path queries.

2.2.3 Contributions

The main contributions of this research work are summarized as follows.

- A novel signature scheme based on irreducible polynomials in finite fields has been proposed. Each XML document is mapped to an algebraic signature that captures its structural characteristics. A query twig is mapped similarly. Twig queries with ‘//’ and ‘*’ are supported, and value predicates can be handled by simple extensions to the proposed signature scheme.

- I show that the existence of a twig pattern in the document can be detected by performing division of the document signature with the query signature.

- A distributed signature index over a P2P network has been developed to organize document signatures. The index is collectively maintained by participating peers in the network. By virtue of the signature index, the signatures of documents with similar structural characteristics tend to be stored together at the same peer, and a search for document sources is resolved quickly.

- The distributed signature index allows data to be partitioned across the peers, adapts to data changing over time, adapts to peers leaving and joining the network, and is
completely decentralized.

- The proposed signature scheme allows holistic processing of twig patterns, and un-ordered twig pattern matching is inherently supported.

- Since $\psi X$ is built atop a Distributed Hash Table (DHT) framework, it inherits the benefits of the DHT such as scalability, load balancing, and robustness.

- The proposed signature scheme can also be used for query processing over heterogeneous XML databases.

2.2.4 Scope

The scope of this research work are summarized as follows.

- The $\psi X$ system has been designed for applications with a typical P2P workload where the queries issued by users are more frequent than document publications. Thus $\psi X$ is not suited for update-intensive applications.

- A schema like information either in the form of a DTD or a structural summary graph over the data is necessary to process user queries.

- Deletion of document signatures is not supported by $\psi X$.

- The $\psi X$ system relies on the underlying DHT framework to handle the dynamism, scalability, robustness, and load fluctuations in a P2P network.

- Currently, $\psi X$ assumes that all peers are non-malicious.

The design, implementation, and evaluation of $\psi X$ is presented in Chapter 4.
2.3 Summary

In this chapter, I have described the background, motivations, and contributions of my dissertation work. I described two major problems namely (a) indexing and querying XML in a centrally stored XML database and (b) indexing and querying XML in a peer-to-peer XML database. I described the challenges that arise in these scenarios and my solutions to address these challenges. I described the salient features and the scope of the PRIX system and the psiX system.
CHAPTER 3
THE PRIX SYSTEM

In this chapter, I present the design, implementation, and empirical evaluation of the PRIX system for indexing and querying XML documents arising in a traditional environment where the data is centrally stored. I begin by describing Prüfer’s method that constructs a one-to-one correspondence between trees and sequences, and describe how Prüfer’s sequences are used for indexing XML data and processing twig queries in the PRIX system. I prove that my sequencing approach finds all occurrences of a twig pattern without any false alarms or false dismissals. I describe the design and implementation of various components of PRIX such as indexing the document sequences using B+-trees. I describe effective optimizations and extensions to PRIX and provide an empirical evaluation. Finally, I shall describe briefly my research on two data-intensive problems: (a) XML document filtering, and (b) XML data streaming, where I have successfully applied the proposed sequencing paradigm for XML documents and twig queries.

3.1 Prüfer Sequences for Labeled Trees: An Introduction

Prüfer (1918) proposed a method that constructed a one-to-one correspondence between a labeled tree and a sequence by removing nodes from the tree one at a time [57]. The algorithm to construct a sequence from tree $T_n$ with $n$ nodes labeled from 1 to $n$ works as follows. From $T_n$, delete the leaf with the smallest label to form a smaller tree $T_{n-1}$. Let $a_1$ denote the label of the node that was the parent of the deleted node. Repeat this process on $T_{n-1}$ to determine $a_2$ (the parent of the next node to be deleted), and continue until only two nodes joined by an edge are left. The sequence $(a_1, a_2, a_3, \ldots, a_{n-2})$ is called the Prüfer sequence of tree $T_n$. From the sequence $(a_1, a_2, a_3, \ldots, a_{n-2})$, the original tree $T_n$ can be reconstructed.
The length of the Prüfer sequence of tree $T_n$ is $n - 2$. In PRIX, however, a Prüfer sequence of length $n - 1$ for $T_n$ is constructed by continuing the deletion of nodes till only one node is left. (The one-to-one correspondence is still preserved). This modified construction simplifies the proofs of the lemmas and theorems presented in Section 3.4.

3.2 Transforming XML Documents into Prüfer Sequences

In the discussions to follow, each XML document is represented by a labeled tree such that each node is associated with its element tag and a number. For example, in Figure 3.1(a), the root element of the XML document has $(A, 15)$ as its tag-number pair. In PRIX, the nodes of an XML document tree are numbered in postorder from one to the total number of nodes. Thus each node is associated with a distinct number.

With tree nodes labeled with unique postorder numbers, a Prüfer sequence can be constructed for a given XML document using the node removal method described in Section 3.1. This sequence consists entirely of postorder numbers and is called the NPS (Numbered Prüfer sequence) of the document. If each number in this NPS is replaced by its corresponding tag, a new sequence that consists of XML tags can be constructed. This sequence is called the LPS (Labeled Prüfer sequence) of the document.\footnote{Occasionally I will refer to an NPS as a postorder number sequence of an LPS.}

Example 3.2.1. In Figure 3.1(a), tree $T$ has $LPS(T) = A C B C B A C A E E E D A$, and

\begin{figure}
\centering
\includegraphics{fig3_3}
\caption{An XML document tree, a query twig, and matches identified by the filtering phase of query processing that represent a disconnected graph and a connected graph}
\end{figure}
For each XML document in the database, its LPS and NPS are constructed. The set of LPS’s are indexed for efficient query processing. The set of NPS’s are stored in the database (e.g., as records in a heap file) together with their unique document identifiers.

3.3 Transforming Twig Queries into Prüfer Sequences

A query twig is transformed into its Prüfer sequence just like an XML document. Non-matches are filtered out by performing subsequence matching on the indexed sequences, and twig matches are then found by applying a series of refinement strategies. These filtering and refinement phases are described in Section 3.4.

Figure 3.2 shows an architectural overview of the indexing and query processing units in PRIX that highlights the steps described in Section 3.2 and Section 3.3. For a detailed description of PRIX’s architecture, refer to Section 3.7. With this high level overview of the system, I shall now explain the process of finding twig matches.
### Table 3.1. Notations used in this chapter

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q )</td>
<td>Query twig</td>
</tr>
<tr>
<td>( \Delta )</td>
<td>A collection of XML documents</td>
</tr>
<tr>
<td>( \Gamma )</td>
<td>A set of Labeled Prüfer sequences of ( \Delta )</td>
</tr>
<tr>
<td>( \Theta )</td>
<td>A set of subsequences in ( \Gamma ) that are identical</td>
</tr>
<tr>
<td>( \Theta_C )</td>
<td>A subset of ( \Theta ) that represent trees (connected)</td>
</tr>
<tr>
<td>( \text{LPS}(T) )</td>
<td>Labeled Prüfer sequence of tree ( T )</td>
</tr>
<tr>
<td>( \text{NPS}(T) )</td>
<td>Numbered Prüfer sequence of tree ( T )</td>
</tr>
<tr>
<td>( \text{Label}(v, T) )</td>
<td>returns the label associated with vertex ( v ) in ( T )</td>
</tr>
<tr>
<td>( \text{Number}(v, T) )</td>
<td>returns the number associated with vertex ( v ) in ( T )</td>
</tr>
<tr>
<td>( S )</td>
<td>A subsequence of an LPS</td>
</tr>
<tr>
<td>( N )</td>
<td>The postorder number sequence of ( S )</td>
</tr>
</tbody>
</table>

### 3.4 Finding Twig Matches

To simplify the presentation of concepts in this section, I use the notations listed in Table 3.1. Formally the problem for finding twig matches can be stated as follows: \textit{Given a collection of XML documents} \( \Delta \) \textit{and a query twig} \( Q \), \textit{report all the occurrences of twig} \( Q \) \textit{in} \( \Delta \). Note that the problem description is based on twig patterns as opposed to XPath expressions. The twig patterns that I deal with in this chapter can be mapped to equivalent XPath expressions. (See Section 1.1.3.)

I will initially deal with the problem of finding all occurrences of twig \( Q \) without the axis `//` and wildcard `*`. Later in Section 3.4.5, I explain how query twigs with `//` and `*` can be processed. Note that my work focuses primarily on \textit{ordered twig pattern matching}. Hence I will first address the problem of finding ordered twig matches of \( Q \) with equality predicates. Later in Sections 3.6.4 and 3.6.4 I explain how PRIX can be extended to handle unordered twig matches and inequality predicates.

Finding twig matches in PRIX involves the following phases, namely (1) \textit{filtering by subsequence matching}, (2) \textit{refinement by connectedness}, (3) \textit{refinement by structure} and (4) \textit{refinement by leaf nodes}. 
3.4.1 Filtering by Subsequence Matching

The filtering phase in PRIX involves subsequence matching. The classical definition of a subsequence is stated as follows.

**Definition 3.4.1.** A subsequence is any string that can be obtained by deleting zero or more symbols from a given string.

Given a query twig $Q$, we find all the subsequences in $\Gamma$ (the set of LPS’s) that match $\text{LPS}(Q)$. I shall discuss the significance of subsequence matching using the following lemma and theorem.

**Lemma 3.4.2.** Given a tree $T$ with $n$ nodes, numbered from 1 to $n$ in postorder, the node deleted in the $i$th step during the Prüfer sequence construction is the node numbered $i$.

**Proof.** I prove the theorem by Induction on the number of nodes in the tree. In each step of the construction of Prüfer sequences, the leaf node with the smallest label is deleted from the tree to give a smaller tree. For simplicity, we will refer to the node numbered $i$ as node $i$. Let $T_{n-k}$ be the tree obtained after $k$ node deletions. Let $P(k)$ denote the proposition that the id of the node deleted in the $k$th step of the Prüfer sequence construction is $k$. Note that $1 \leq k < n$.

1. **Basis of Induction:** $P(1)$ is true. This is because a node numbered one in postorder is a leaf and is deleted first.

2. **Induction Hypothesis:** Assume that $P(i)$ is true for $1 \leq i \leq k$. We show that $P(k+1)$ is also true. Only the two following scenarios are possible for nodes numbered $k$ and $k+1$ in postorder: (1) node $k+1$ is a leaf, (2) node $k+1$ is not a leaf and node $k$ is the last child of node $k+1$. (Refer to Figure 3.3.) If node $k+1$ is a leaf in $T_n$, then by induction hypothesis, node $k+1$ is the smallest leaf in $T_{n-k}$. Thus $P(k+1)$ is true. If node $k$ is the last child of node $k+1$, then node $k+1$ becomes a leaf after
Figure 3.3. Two scenarios where nodes are assigned consecutive postorder numbers

By induction hypothesis, all nodes \( i \) such that \( i \leq k \) have been deleted. The smallest leaf node in \( T_n - k \) is \( k + 1 \) and is deleted next. Thus \( P(k + 1) \) is true.

As a result, if \( a \) and \( b \) are two nodes of a tree such that \( a \) has a smaller postorder number than \( b \), then node \( a \) is deleted before node \( b \) during the Prüfer sequence construction. In addition, the \( i^{th} \) element in the NPS denotes the postorder number of the parent of node \( i \).

In the subsequent discussions, we shall frequently use two notations namely \( \text{Number}(\cdot) \) and \( \text{Label}(\cdot) \) that are described in Table 3.1. Based on Lemma 3.4.2, we can state the following theorem.

**Theorem 3.4.3.** If tree \( Q \) is a subgraph of tree \( T \), then \( \text{LPS}(Q) \) is a subsequence of \( \text{LPS}(T) \).

**Proof.** Assume that the tree nodes in \( T \) and \( Q \) are numbered in postorder. Let \( v_1, v_2, v_3, \ldots, v_{n-1} \) be the order of deletion of nodes by the Prüfer sequence construction for tree \( Q \) (with \( n \) nodes). Since \( Q \) is a subgraph of \( T \), the same set of nodes are also deleted during the Prüfer sequence construction for tree \( T \) in some order along with other nodes in \( T \). We would like to prove that for \( 1 \leq i < n \), \( v_i \) is deleted before \( v_{i+1} \) in tree \( T \).

By Lemma 3.4.2, it is true that \( \text{Number}(v_i, Q) < \text{Number}(v_{i+1}, Q) \) for any pair of nodes \( v_i \) and \( v_{i+1} \) in \( Q \) \((1 \leq i < n)\). Furthermore, for any pair of nodes \( v_i \) and \( v_j \) in \( Q \),
if \( \text{Number}(v_i, Q) < \text{Number}(v_j, Q) \) is true, then \( \text{Number}(v_i, T) < \text{Number}(v_j, T) \) is also true. (Note that nodes \( v_i \) and \( v_j \) are also in \( T \).) Therefore nodes \( v_i \) and \( v_{i+1} \) satisfy the condition \( \text{Number}(v_i, T) < \text{Number}(v_{i+1}, T) \) for \( 1 \leq i < n \). By Lemma 3.4.2, \( v_i \) is deleted before \( v_{i+1} \) in tree \( T \). Since for \( 1 \leq i \leq n \), \( \text{Label}(v_i, Q) = \text{Label}(v_i, T) \), \( \text{LPS}(Q) \) is a subsequence of \( \text{LPS}(T) \).

From Theorem 3.4.3, it is evident that by finding every subsequence in \( \Gamma \) that matches \( \text{LPS}(Q) \), we are guaranteed to have no \textit{false dismissals}.

**Example 3.4.4.** Consider trees \( T \) and \( Q \) in Figure 3.1(a) and Figure 3.1(b). Tree \( T \) in Figure 3.1(a) has \( \text{LPS}(T) = A C B C B A C A E E D A \) and \( \text{NPS}(T) = 15 \ 3 \ 7 \ 6 \ 6 \ 7 \ 15 \ 9 \ 15 \ 13 \ 13 \ 14 \ 15 \). Tree \( Q \) in Figure 3.1(b) has \( \text{LPS}(Q) = B A E D A \) and \( \text{NPS}(Q) = 2 \ 6 \ 4 \ 5 \ 6 \). \( Q \) is a (labeled) subgraph of \( T \) and \( \text{LPS}(Q) \) matches a subsequence \( S \) of \( \text{LPS}(T) \) at positions \( 6, 7, 11, 13, 14 \). The postorder number sequence of \( S \) is \( 7 \ 15 \ 13 \ 14 \ 15 \). Note that there may be more than one subsequence in \( \text{LPS}(T) \) that matches \( \text{LPS}(Q) \).

### 3.4.2 Refinement by Connectedness

The subsequences matched during the filtering phase are further examined for the property of \textit{connectedness}. This is because the nodes that correspond to the labels in a subsequence may not be connected (represent a tree) in the data tree. Let \( \Theta_C \) denote a set of subsequences that satisfy the connectedness property. Formally, I state a \textit{necessary} and \textit{sufficient} condition for a subsequence \( S \in \Theta_C \).

**Theorem 3.4.5.** Given a tree \( T \), let \( S \) be a subsequence of \( \text{LPS}(T) \), and let \( N[i] \) denote the \( i^{th} \) element in the postorder number sequence of \( S \). (\( N[i] \) is the postorder number taken from \( \text{NPS}(T) \) for a node corresponding to the \( i^{th} \) element in \( S \).) Then the tree nodes in \( T \) corresponding to the elements of \( S \) constitute a connected subgraph (or sub-tree) of \( T \), if and only if, condition 3 is true for every \( N[i] \) that satisfies conditions 1 and 2.

1. \( N[i] \neq \max(N[1], N[2], ..., N[|S|]) \),
2. $N[i]$ is the last occurrence of the same number in the postorder number sequence of $S$

3. $N[i + 1]$ is equal to the $N[i]^{th}$ element of NPS($T$).

**Proof.** (If) part: For each node corresponding to the elements in $S$, we shall check if its parent node in $T$ also appears in $S$. If $N[i]$ is the maximum in the postorder number sequence of $S$, then neither the parent nor the ancestor of its corresponding node is in $S$. Thus the node for $N[i]$ need not be checked for its parent in $S$. Moreover, if $S$ represents a sub-tree of $T$ then the node for $N[i]$ would represent the root of that sub-tree. On the other hand, if $N[i]$ is not the maximum in the postorder number sequence of $S$, then the node for $N[i]$ has to be connected to its parent in order for $S$ to represent a sub-tree of $T$.

From Lemma 3.4.2, if the last occurrence of a node’s postorder number $n$ occurs at the $i^{th}$ position in an NPS, then the number at the $(i + 1)^{th}$ position in the NPS is the postorder number of the parent of $n$. This is because, during Prüfer sequence construction, the deletion of the last child of $n$ makes $n$ the smallest leaf node. Hence if $N[i + 1]$ is equal to the $N[i]^{th}$ element of NPS($T$), then $N[i]$’s node is connected to its parent node in $S$. If this is true for every $N[i]$ in $S$ ($N[i]$ is not the maximum), then the tree nodes in $T$ corresponding to the elements of $S$ represent a connected subgraph of $T$.

(Only If) part: It is given that the tree nodes in $T$ corresponding to the elements in $S$ constitute a connected subgraph. Let us call it tree $T’$. First we number these $|S + 1|$ nodes in postorder. Then the last occurrence of a postorder number in $NPS(T’)$ (except the root) is followed by its parent’s postorder number. Now let us replace the postorder numbers of the nodes in $T’$ with their corresponding postorder numbers from $T$. As a result, we can transform $NPS(T’)$ to the postorder number sequence of $S$. Since each node is assigned a unique number, the old and new postorder numbers of nodes in $T’$ have a one-to-one correspondence. Without loss of generality, let $n_c$ and $n_p$ be the old postorder numbers of a child and parent node in $T’$ respectively. Let $n_c’$ and $n_p’$ be their new postorder numbers. Since the last occurrence of $n_c’$ is followed by $n_p’$ in $NPS(T’)$, the last occurrence of $n_c$
(say at \( N[i] \)) is followed by \( n_p \) (at \( N[i + 1] \)) in the postorder number sequence of S. The \( N[i]^{th} \) element in NPS(T) corresponds to the parent of the node numbered \( n_c \) i.e., \( n_p \). As a result, the conditions 1, 2 and 3 are true. Hence we have proven the Only If case.

The intuition for the above theorem is as follows. Let \( i \) be the index of the last occurrence of a postorder number \( n \) in an NPS. This last occurrence is a result of deletion of the last child of \( n \) during Prüfer sequence construction. Hence the next child to be deleted (based on Lemma 3.4.2) is the node \( n \) itself. Hence the number at the \((i + 1)^{th}\) index in the NPS, say \( m \), is the postorder number of the parent of node \( n \). Thus \( n \) followed by \( m \) indicates that there is an edge between node \( m \) and node \( n \).

**Example 3.4.6.** Consider 2 subsequences \( S_A \) and \( S_B \) of LPS(T) where T is the tree in Figure 3.1(a). Let \( S_A \) be C B C E D whose postorder number sequence \( N_A \) is 3 7 9 13 14. Let \( S_B \) be C B A C A E D A whose postorder number sequence \( N_B \) is 3 7 15 9 15 13 14 15. Let \( N_T \) be the NPS of T. Then \( N_T \) is 15 3 7 6 7 15 9 15 13 13 13 14 15. The nodes represented by labels of \( S_A \) form a disconnected graph as shown in Figure 3.1(c). In this case, \( \max(N_A[1], N_A[2], \ldots, N_A[5]) = 14 \). The last occurrence of postorder number 7 in \( N_A \) is at the 2\(^{nd}\) position since there is no index \( j > 2 \) such that \( N_A[j] = 7 \). However \( N_A[2] \) is not followed by \( N_T[7] \), i.e., \( N_A[3] \neq 15 \). Hence Theorem 3.4.5 is not satisfied. The nodes represented by elements of \( S_B \) represent a tree as shown in Figure 3.1(d) because Theorem 3.4.5 is satisfied.

I shall refer to sequences that satisfy Theorem 3.4.5 as tree sequences.

### 3.4.3 Refinement by Twig Structure

The tree sequences obtained in the previous refinement phase are further refined based on the query twig structure. In this phase, we would like to determine if the structure of the tree represented by a tree sequence matches the query twig structure.
Notion of Gaps Between Tree Nodes  Before I delve into the details of refinement by structure, I shall first introduce the notion of gap between two tree nodes and gap consistency between two tree sequences.

**Definition 3.4.7.** The gap between two nodes \( a \) and \( b \) in a tree is defined as the difference between the postorder numbers of the nodes \( a \) and \( b \).

By using the NPS of a tree, the gaps between tree nodes can be computed.

**Definition 3.4.8.** Tree sequence \( S_A \) is said to be gap consistent with respect to tree sequence \( S_B \) if

1. \( S_A \) and \( S_B \) have the same length \( n \),

2. For every pair of adjacent elements in \( A \) and the corresponding adjacent elements in \( B \), their gaps \( g_A \) and \( g_B \) have the same sign, and if \( g_A > 0 \) then \( g_A \leq g_B \), else \( g_A = g_B = 0 \).

Note that gap consistency is not a symmetric relation.

**Example 3.4.9.** Consider the tree \( T \) in Figure 3.1(a). \( LPS(T) = A \ C \ B \ C \ C \ B \ A \ C \ A \ E \ E \ E \ D \ A \), and \( NPS(T) = 15 \ 3 \ 7 \ 6 \ 6 \ 7 \ 15 \ 9 \ 15 \ 13 \ 13 \ 14 \ 15 \). Let \( S_1 = B \ A \ E \ E \ D \ A \) be a subsequence of \( LPS(T) \) and let \( N_1 = 7 \ 15 \ 13 \ 13 \ 14 \ 15 \) be the postorder number sequence of \( S_1 \). Consider a query tree with \( LPS \ S_2 = B \ A \ E \ E \ D \ A \), and \( NPS \ N_2 = 2 \ 7 \ 5 \ 5 \ 6 \ 7 \). Then \( S_2 \) is gap consistent with \( S_1 \) because the gap between

- the 1st pair of elements in \( S_2 \) is -5,
- the 1st pair of elements in \( S_1 \) is -8,
- the 2nd pair of elements in \( S_2 \) is 2,
- the 2nd pair of elements in \( S_1 \) is 2,
Figure 3.4. Examples of pruning tree sequences during the refinement-by-structure phase using gap consistency

- the 3rd pair of elements in $S_2$ is 0,
- the 3rd pair of elements in $S_1$ is 0, and so on.

Intuitively, the gap between two nodes in a data tree gives an idea of how many nodes occur between these two nodes during the postorder traversal. Similar is the case with the nodes of a query twig. Suppose $M$ denotes the NPS of a tree. Let $M[i]$ and $M[i + 1]$ denote the postorder number of nodes corresponding to the $i^{th}$ and the $(i + 1)^{th}$ entry in $M$. A negative gap between $M[i]$ and $M[i + 1]$ indicates that node $M[i]$ is a child of node $M[i + 1]$. However, a positive gap between $M[i]$ and $M[i + 1]$ indicates that node $M[i]$ is an ancestor of $M[i + 1]$. A zero gap implies that $M[i]$ and $M[i + 1]$ correspond to the same node in the XML document. If more nodes are traversed in the query twig as compared to a twig matched in the data, then this indicates that there is a structural difference between the match in the data and the query twig. This concept forms the basis for Theorem 3.4.13 that states a necessary and sufficient condition for a match by twig structure.

Example 3.4.10. I illustrate how gap consistency can be used to prune away tree sequences that are false matches. Consider a tree $T_g$ and a query pattern $Q_g$ as shown in Figures 3.4(a) and 3.4(b). Note that $LPS(T_g) = C C A B A$, $NPS(T_g) = 3 3 4 5 6$, $LPS(Q_g)$
= C A B A, and \( \text{NPS}(Q_g) = 2 5 4 5 \). \( \text{LPS}(Q_g) \) matches a subsequence \( S = C A B A \) in \( \text{LPS}(T_g) \) whose postorder number sequence is 3 4 5 6. Although \( S \) is a tree sequence by Theorem 3.4.5, \( \text{LPS}(Q_g) \) is not gap consistent with \( S \) (by Definition 3.4.8). Therefore \( Q_g \) does not have a structural match in \( T_g \).

Another key observation that will be used in Theorem 3.4.13 is the following. The number of times a number \( n \) occurs in an NPS indicates the number of child nodes of \( n \) in the tree, and the positions that \( n \) occurs in the NPS depend on the subtrees rooted at node \( n \).

I formalize the above observation by defining a property called frequency consistency.

**Definition 3.4.11.** Tree sequences \( S_A \) and \( S_B \) are frequency consistent if

1. \( S_A \) and \( S_B \) have the same length \( n \),

2. Let \( N_A \) and \( N_B \) be the postorder number sequences of \( S_A \) and \( S_B \) respectively. Let \( N_A[i] \) and \( N_B[i] \) be the \( i \)th element in \( N_A \) and \( N_B \) respectively. For every \( i \) from 1 to \( n \), \( N_A[i] \) occurs \( k \) times in \( N_A \) at positions \( \{p_1, p_2, \ldots, p_k\} \), if and only if, \( N_B[i] \) occurs \( k \) times in \( N_B \) at positions \( \{p_1, p_2, \ldots, p_k\} \).

Note that frequency consistency is an equivalence relation.

**Example 3.4.12.** In Example 3.4.9, sequences \( S_1 \) and \( S_2 \) are frequency consistent. The 1st element in \( N_1 \) ‘7’ occurs once at position 1. The 1st element in \( N_2 \) ‘2’ also occurs once at position 1. The 2nd element in \( N_1 \) ‘15’ occurs at positions 2 and 6. The 2nd element in \( N_2 \) ‘7’ also occurs at positions 2 and 6. Similar is the case with the remaining elements in \( N_1 \) and \( N_2 \).

It should be noted that the LPS of a tree contains only the non-leaf node labels. So in addition to the LPS and NPS, the label and postorder number of every leaf node should be stored in the database. Since the LPS of a tree contains only non-leaf node labels, filtering
by subsequence matching followed by refinement by connectedness and structure can only find twig matches in the data tree whose tree structure is the same as the query tree and whose non-leaf node labels match the non-leaf node labels of the query twig. Let us call such matches *partial twig matches*. To find a *complete twig match*, the leaf node *labels* of a partially matched twig in the data should be matched with the leaf node *labels* of the query. This is explained in Section 3.4.4.

I now state a necessary and sufficient condition for a *partial twig match*.

**Theorem 3.4.13.** Tree $Q$ has a partial twig match in tree $T$ if and only if

1. $LPS(Q)$ matches a subsequence $S$ of $LPS(T)$ such that $S$ is a tree sequence, and

2. $LPS(Q)$ is gap consistent and frequency consistent with $S$.

**Proof. (Only If) part:** Suppose $Q$ has a partial match in tree $T$, then $Q$ matches a sub-graph $T'$ of $T$, except that the leaf node labels of $Q$ may not match with the leaf node labels of $T'$. By Lemma 3.4.2, $LPS(T')$ is a subsequence of $LPS(T)$. The set of nodes in $T'$ and $Q$ are deleted in the same relative order during Prüfer sequence construction (using postorder numbering) of trees $T$ and $Q$ respectively. Furthermore, since $Q$ and $T'$ have the same structure, the corresponding non-leaf nodes in $Q$ and $T'$ have the same number of child nodes. Thus $LPS(Q)$ matches a subsequence $S$ of $LPS(T)$ that represents tree $T'$ (i.e., tree sequence), and $S$ and $LPS(Q)$ are frequency consistent.
Our goal is to show that LPS(Q) is gap consistent with S. Consider two nodes p and q in tree Q. Let Label(p, Q) and Label(q, Q) be two adjacent elements in LPS(Q) where Label(p, Q) occurs before Label(q, Q). Let \( n_{pQ} \) and \( n_{qQ} \) be the postorder numbers of nodes p and q in tree Q. Note that nodes p and q are internal nodes in Q. Let nodes r and s be nodes in \( T' \) that match p and q respectively. Label(r, T) and Label(s, T) are adjacent in S. Let \( n_{rT} \) and \( n_{sT} \) be the postorder numbers of nodes r and s in tree T. Let \( k \) denote the number of nodes in a tree. Let \( g_T \) be the gap between r and s in T, and let \( g_Q \) be the gap between p and q in Q. The following are the only possible scenarios for nodes p, q, r and s.

**Case 1** r is a child of s, and p is a child of q. (Refer to Figures 3.5(a) and 3.5(b).) Let \( K_t \) be the subtree of T rooted at s whose nodes have their postorder number \( n_{K_t} \), \( n_r < n_{K_t} \leq n_s \). Let \( K_q \) be the subtree of Q rooted at q whose nodes have their postorder number \( n_{K_q} \), \( n_p < n_{K_q} \leq n_q \). We know that \( \|K_q\| \leq \|K_t\| \), since Q matches a tree \( T' \) that is a subgraph of T. Also, by virtue of postorder numbering,

\[
g_T = n_{sT} - n_r = \|K_t\| \quad \text{and} \quad g_Q = n_q - n_p = \|K_q\|.
\]

Therefore \( g_Q \) and \( g_T \) have the same sign and \( |g_Q| \leq |g_T| \).

**Case 2** s is a descendant of r, and q is a descendant of p. (Refer to Figures 3.5(c) and 3.5(d).) Let \( m_Q \) be the number of nodes in Q whose postorder number \( n_q \), \( n_Q < n_{pQ} \). Let \( m_T \) be the number of nodes in T whose postorder number \( n_T \), \( n_s < n_T < n_r \). By virtue of postorder numbering,

\[
g_T = n_r - n_s = m_T + 1 \quad \text{and} \quad g_Q = n_{pQ} - n_q = m_Q + 1.
\]

Since Q matches tree \( T' \) that is a subgraph of T, \( m_Q \leq m_T \). Therefore \( g_Q \) and \( g_T \) have the same sign and \( |g_Q| \leq |g_T| \).
Case 3  \(p\) and \(q\) are the same nodes (i.e., \(\text{Label}(p, Q) = \text{Label}(q, Q)\) and \(\text{Number}(p, Q) = \text{Number}(q, Q)\)), and \(r\) and \(s\) are the same nodes (i.e., \(\text{Label}(r, T) = \text{Label}(s, T)\) and \(\text{Number}(r, T) = \text{Number}(s, T)\)). This is a trivial case. \(g_Q\) and \(g_T\) are both zero.

**If** part: Given LPS(\(Q\)) matches a subsequence \(S\) of LPS(\(T\)) where \(S\) is a tree sequence. Also LPS(\(Q\)) is gap consistent and frequency consistent with \(S\). Let tree \(T'\) be a subgraph of \(T\) that \(S\) represents (i.e., LPS(\(T'\)) = \(S\)). From the proof of (Only If) part, LPS(\(T'\)) is gap consistent with \(S\), and LPS(\(T'\)) and \(S\) are frequency consistent. We know that LPS(\(Q\)) and LPS(\(T'\)) are identical. Our goal is to show that NPS(\(Q\)) and NPS(\(T'\)) are identical. We shall use contradiction to prove this.

\(Q\) and \(T'\) are trees with \(n\) nodes numbered from 1 to \(n\) in postorder. Let \(N_Q\) be the NPS of \(Q\) and let \(N_{T'}\) be the NPS of \(T'\). Let us assume that \(\exists i\) where \(i\) is the largest index, such that, \(N_Q[i] \neq N_{T'}[i]\). The index \(i\) will always be less than \(n - 1\), since \(N_Q[n - 1] = N_{T'}[n - 1] = n\), where \(n\) is the number of nodes in trees \(Q\) and \(T'\).\(^2\) Let \(N_{T'}[i] = a\) and let \(N_Q[i] = b\) such that \(a \neq b\). Let \(N_Q[i + 1] = N_{T'}[i + 1] = c\). Let us refer to the node numbered \(i\) as node \(i\). Based on Lemma 3.4.2, in tree \(Q\), the parent of node numbered \(i\) has postorder number \(b\), and the parent of node \(i + 1\) has postorder number \(c\). Similarly, in tree \(T'\), the parent of node numbered \(i\) has postorder number \(a\), and the parent of node \(i + 1\) has postorder number \(c\). Consider the following relationships between \(a, b\) and \(c\).

\(^2\)Due to postorder numbering, the root of the tree has the highest number and its child is the last node to be deleted during Prüfer sequence construction.
**Case 1** \(a < c < b\) (or \(b < c < a\)). This situation cannot occur since LPS(\(Q\)) and LPS(\(T'\)) are each gap consistent with \(S\).

**Case 2** \(a = c\) (or \(b = c\)). Then the gap is zero. Since LPS(\(Q\)) is gap consistent with \(S\) and LPS(\(T'\)) is gap consistent with \(S\), \(b = c\) (or \(a = c\)) should be true. Hence our assumption that \(a \neq b\) is false. Therefore \(a = b = c\).

**Case 3** \(c > a; c > b\). (Refer to Figures 3.6(a) and 3.6(b).) Without loss of generality, let us assume that \(a < b\). Since \(T'\) is numbered in postorder and \(a < c, i + 1 = a\), and \(i = a - 1\) (Figure 3.6(a)). Now in tree \(Q\) (Figure 3.6(b)), the node \(i\) (i.e., \(a - 1\)), has its parent numbered \(b\) and the node \(i + 1\) (i.e., \(a\)), has its parent numbered \(c\). Also \(a < b < c\). This contradicts the fact that \(Q\) is numbered in postorder i.e., node \(c\) cannot be a descendant of node \(b\). Hence our assumption that \(a < b\) is false. Using a similar argument and swapping the roles of \(T'\) and \(Q\), we can show that an assumption \(a > b\) is also false. Thus our assumption that \(a \neq b\) is false, and \(a = b\) must be true.

**Case 4** \(c < a; c < b\). (Refer to Figures 3.6(c) and 3.6(d).) Without loss of generality, let us assume that \(a < b\). Since \(c < a, c < b\), we have a scenario as shown in Figures 3.6(c) and 3.6(d) for the nodes \(i\) and \(i + 1\). The dotted line in the figure indicates that there is an ancestor-descendant relationship between the nodes. Node \(c\) is the parent of node \(i + 1\). Node \(a\) in \(T'\) and node \(b\) in \(Q\) have at least one child with postorder number greater than \(i\). Also node \(a - 1\) is a child of \(a\) in \(T'\) and node \(b - 1\) is a child of \(b\) in \(Q\). As a result of Lemma 3.4.2, \(N_{T'}[a - 1] = a\) and \(N_Q[b - 1] = b\). We know that LPS(\(T'\)) is frequency consistent with \(S\). This means that the \(i^{th}\) and \((a - 1)^{th}\) element in the postorder number sequence of \(S\) should be equal. Since LPS(\(Q\)) is frequency consistent with \(S\), the \(i^{th}\) and \((a - 1)^{th}\) element in \(N_Q\) should be equal. This implies that \(N_Q[a - 1] = b\) and \(N_{T'}[a - 1] = a\). This contradicts the original assumption that \(i\) is the largest index such that \(N_Q[i] \neq N_{T'}[i]\) since \(i < c < a\).
Hence our assumption that \( a \neq b \) (i.e., \( a < b \)) is false. Using a similar argument, we can show that an assumption \( a > b \) is also false. Hence \( a = b \) must be true.

From the above four cases, we conclude that our original assumption that \( \exists i \) where \( i \) is the largest index s.t. \( N_Q[i] \neq N_T'[i] \) is false. Thus \( \text{NPS}(T') \) and \( \text{NPS}(Q) \) are identical. Since every Prüfer sequence corresponds to a unique labeled tree (one-to-one correspondence), \( Q \) matches \( T' \) except that the labels of their leaf nodes (i.e., \( \text{Label}(.) \)) may not match. Thus \( Q \) has a partial match in \( T \).

The different relationships between the data and query sequences are illustrated in Figure 3.7. Consider the tree \( T \) (XML document) and its subgraph tree \( Q \) (query twig) in the figure. The dark regions in \( \text{LPS}(T) \) and \( \text{NPS}(T) \) correspond to the deletion of nodes in \( T \) during Prüfer sequence construction that are also in \( Q \) (except the root of \( Q \)). The dark regions in \( \text{LPS}(T) \) and \( \text{NPS}(T) \) form sequences \( S \) and \( N \) respectively. From the lemmas and theorems described in Section 3.4.1, Section 3.4.2, and Section 3.4.3, it is evident that (1) \( \text{LPS}(Q) \) and \( S \) are identical, (2) \( \text{NPS}(Q) \) is gap consistent with \( N \), and (3) \( \text{NPS}(Q) \) and \( N \) are frequency consistent.
3.4.4 Refinement by Matching Leaf Nodes

In the final refinement phase, the leaf node labels of the query twig are tested to find complete twig matches.

**Example 3.4.14.** The leaf nodes of tree $T$ in Figure 3.1, namely, (D,2), (D,4), (E,5), (G,10), (F,11), and (F,12) are stored in the database. Let tree $Q$ (Figure 3.1(b)) be a twig query. $LPS(Q)$ matches a subsequence $S = B A E D A$ in $LPS(T)$ at positions $P = (3, 7, 11, 13, 14)$. The postorder number sequence of $S$ is $N = 7 15 13 14 15$. $LPS(Q)$ is gap consistent and frequency consistent with $S$. The leaf (F,3) in $Q$ can be matched as follows. Since the leaf has postorder number 3, its parent node matches the node numbered 13 (i.e., the 3rd element of $N$) in the data tree. Also since this node numbered 13 occurs at the 11th position (3rd element in $P$) in $LPS(T)$, it may have a leaf (F,11). And indeed, we have (F,11) in the leaf node list of $T$. In a similar way, we can match the leaf (C,1) of $Q$. The parent of (C,1) in $Q$ matches node 7 (1st element in $N$) at position 3 in $NPS(T)$. Hence the child of node 7 in $T$, i.e., node 3, matches leaf (C,1), except that the labels may not match (partial twig match). Since there are no nodes with number 3 in the leaf list of $T$, we search $LPS(T)$ and $NPS(T)$ to find (C,3) in $T$. Indeed we have ‘C’ and ‘3’ as the 2nd element in $LPS(T)$ and $NPS(T)$ respectively.

However, this refinement phase can be eliminated by special treatment of leaf nodes in the query twig and the data trees. The key idea is to make the leaf nodes of the query twig and the data trees appear in their LPS’s, so that all the nodes are examined during subsequence matching and refinement by connectedness and structure phases. This process is explained in Section 3.6.3.

3.4.5 Processing ‘//’ and ‘*’ in Twig Queries

I describe how the axis ‘//’ and the wildcard ‘*’ can be handled in twig queries using the following example.
Example 3.4.15. Let us find the pattern $Q = //A//C/D$ in tree $T$ (in Figure 3.1(a)). $Q$ is transformed to its Prüfer sequences by ignoring `//' (and `*' if present). As a result, $LPS(Q) = CA$, and $NPS(Q) = 2\ 3$. The wildcard at the beginning of the query is handled by my current method as it allows finding occurrences of a query tree anywhere in the data tree. To process `//` in the middle of the query, a simple modification to the refinement-by-connectedness phase is done. $LPS(Q)$ matches a subsequence $S = CA$ at positions 2 and 7 in $LPS(T)$. The postorder number sequence of $S$ is $N = 3\ 15$. Based on Theorem 3.4.5, this subsequence would be discarded as the last occurrence of 3 in $N$ is not followed by 7 (parent of node numbered 3 in $T$). To avoid this, we check if the last occurrence of node 3 in $N$ can lead to node 15 (15 follows 3 in $N$) by following a series of edges in $T$. Recall that the $i^{th}$ element in an NPS is the postorder number of the parent of node $i$ in a tree (Lemma 3.4.2). Let $n_0 = 3$ and let $N_T$ be $NPS(T)$. We recursively check if $n_1 (= N_T[n_0])$ equals 15, then if $n_2 (= N_T[n_1])$ equals 15 and so on until for some $i$, $n_{i+1} (= N_T[n_i])$ equals 15. In the above example, a match at $i = 2$ is found. For processing wildcard `*`, we simply test whether the match is found at $i = 2$. Thus all the subsequences that pass the above test will be examined in the next phase.

If the wildcard `*` appears as a branch node in the query, then a few modifications are required. For example, consider the query $//A/*[B/D]//C/F$. In this case, $BCA$ is the sequence representation for the query. If matching subsequences are found, then the NPS is used to test if $A$ can be reached from $B$ and $C$ in one step as explained before. In addition, it should be ensured that the parent of $B$ and $C$ are one and the same. This can be done as follows. Let $n_B$ and $n_C$ represent the postorder numbers of $B$ and $C$ in the data respectively. If the condition $N_T[n_B] = N_T[n_C]$ is true, then $B$ and $C$ have the same parent.

Note that in the above example, the leaf nodes of the query patterns do not appear in the sequences. In Section 3.6.3, I explain how the leaf nodes can be made to appear in the sequences.
3.5 Indexing the Document Sequences

Given the theoretical background in Section 3.4, we shall move on to explain the implementation issues in PRIX.

3.5.1 Building Prüfer Sequences

In the PRIX system, Prüfer sequences are constructed for XML document trees (with nodes numbered in postorder) using the method described in Section 3.1. Our proposed tree-to-sequence transformation causes the nodes at the lower levels of the tree to be deleted first. This results in a bottom-up transformation of the tree. An algorithm describing the sequence construction using a SAX based interface is provided in Section 3.7. Note that the entire XML document need not be available in memory during sequence construction and can be processed in document order. I shall show in the experiments that the bottom-up transformation is useful to process twig queries efficiently.

3.5.2 Indexing Sequences Using B+-trees

The set of Labeled Prüfer sequences for the XML documents are indexed in order to support fast subsequence matching during query processing. Maintaining an in-memory index for the sequences like a trie is unsuitable, as the index size grows linearly with the total length of the sequences. In essence, we would like to build an efficient disk-based index. Note that the set of Numbered Prüfer sequences are stored in the database (e.g., as records in a heap file).

In fact, Prüfer sequences can be indexed using any good method for indexing strings. In the current version of PRIX, we index Labeled Prüfer sequences using B+-trees in the similar way that [75] build a virtual trie using B+-trees. Since the virtual trie is a dynamic index, XML documents can be added to and deleted from the database.
Virtual Trie I shall briefly explain the process of indexing sequences using a virtual trie. Essentially, we provide positional representations for the nodes in the trie by labeling them with ranges. Each node in the trie is labeled with a range \((\text{LeftPos}, \text{RightPos})\) such that the containment property is satisfied [43, 83]. Typically, the root node can be labeled with a range \((1, \text{MAX\_INT})\). The child nodes of the root can be labeled with subranges such that these subranges are disjoint and are completely contained in \((1, \text{MAX\_INT})\). This containment property is recursively satisfied at every non-leaf node in the trie. We can then obtain all the descendants of any given node \(A\) by performing a range query that finds nodes whose \text{LeftPos} falls within the \((\text{LeftPos}, \text{RightPos})\) range of node \(A\).

In the PRIX system, for each element tag \(e\), we build a B\(^+\)-tree that indexes the positional representation of every occurrence of element \(e\) in the trie using its \text{LeftPos} as the key. This index is called \emph{Trie-Symbol} index. In addition, we store the identifier of each document tree in a separate B\(^+\)-tree and index it using the \text{LeftPos} of the node, where its LPS ends in the virtual trie, as the key. This index is called \emph{Docid} index. Note that it is sufficient to store only the LPS’s in the virtual trie. The suffixes of the LPS’s need not be indexed at all, since all the subsequences can be found by performing range queries on the \emph{Trie-Symbol} indexes as described in Section 3.6.1.

ViST proposed a dynamic labeling scheme that can assign number ranges without building a physical trie on the set of sequences [75], hence the name virtual trie. However, this dynamic labeling scheme suffers from \emph{scope underflows} [75] for long sequences and large alphabet sizes, which makes it difficult to implement. In order to reduce the scope underflows, we \emph{pre-allocate} the number ranges for a small subset of nodes in the trie. The remaining nodes are assigned ranges using the dynamic labeling scheme. In order to do so, we build an in-memory trie for all the prefixes of the sequences of length \(\alpha\) (where \(\alpha\) is a small number compared to the actual length of the sequences). A node in this in-memory trie is allocated a number range based on the \emph{frequency} and \emph{length} of the sequences whose prefixes share that node.
Local Range Assignment  Alternatively, the Trie-Symbol indexes can be created even without taking subranges from the global number range \((1, MAX\_INT)\) for the nodes in a virtual trie. Instead, each element in an LPS can be represented by \((Docid, [LeftPos, RightPos])\), where Docid is a unique identifier of the sequence, and LeftPos and RightPos are assigned locally such that the containment property is satisfied recursively between each pair of adjacent elements in the sequence \([56]\). A Trie-Symbol index is built for each element tag using \((Docid, LeftPos)\) pairs as keys, while RightPos values are stored as data. Consequently, the Docid index need not be constructed, and this scheme does not suffer from the problem of scope underflows as the ranges are assigned to a sequence independent of other sequences.

Example 3.5.1. Consider two LPSs A B C D and B D C with document ids 1 and 2, respectively. For document id 1, the elements A, B, C, and D are represented as \((1, [1,8])\), \((1, [2,7])\), \((1, [3,6])\), \((1, [4,5])\), respectively. Note that the range of A contains the ranges of B, C, and D. For document id 2, the elements B, C, and D are represented as \((2, [1,6])\), \((2, [2,5])\), \((2, [3,4])\), respectively. To index these two sequences, we build a B\(^+\)-tree index for tag A with key \((1,1)\), a B\(^+\)-tree index for tag B with keys \((1,2)\) and \((2,1)\), a B\(^+\)-tree index for tag C with keys \((1,3)\) and \((2,3)\), and a B\(^+\)-tree index for tag D with keys \((1,4)\) and \((2,2)\).

3.5.3 Cost Analysis

Space Complexity The size of a trie grows linearly with the total length of the sequences stored in it. The length of the Prüfer sequence of a tree is linear in the number of nodes in the tree. Hence the index size is linear in the total number of tree nodes, while ViST does not guarantee a linear worst-case bound on the index size. (Refer to Section 2.1.2.)
3.6 Twig Query Processing

3.6.1 Filtering by Subsequence Matching

Let $Q_s = Q_{s1}Q_{s2}...Q_{sk}$, a sequence of length $k$, denote the LPS of a query twig $Q$. The process of finding all occurrences of $Q_s$ using the Trie-Symbol indexes is shown in Algorithm 1. The algorithm is invoked by $FindSubsequence(Q_s, 1, (0, \text{MAX}_\text{INT})$). A range query in the open interval $(q_t, q_r)$ is performed on $T_{Q_{si}}$ (Trie-Symbol index of $Q_{si}$) (line 1). For every node id $r$ returned from the range query (line 1), if the sequence $Q_s$ is found then all the documents in the closed interval $[r_l, r_r]$ are fetched from the Docid index (line 5). $(r_l, r_r)$ is the positional representation of node id $r$. (In this case $r_l = r_r$.) Otherwise, $FindSubsequence(\cdot)$ is recursively invoked for the next element $Q_{si}(i+1)$ in the sequence using the range $(r_l, r_r)$. In line 3, the position of match of the $i^{th}$ element of $Q_s$ (i.e., the level of node $r$ in the trie) is stored in $S$. The solutions of the range query in line 1 are the ids of nodes $Q_{si}(i+1)$ that are descendants of nodes $Q_{si}$ in the virtual trie. In line 4, the algorithm outputs a set of document (tree) identifiers $D$ and a list $S$. $S$ contains the positions in the LPS’s of trees corresponding to tree identifiers in $D$ where $Q_s$ has a subsequence match.

It should be noted that the subsequence matching phase is I/O bound. The total number of range queries issued in this phase depends on the length of the sequence $Q_s$ and $|R|$ in Algorithm 1. Our goal is to reduce the number of paths explored in the virtual trie to find all the subsequences.

**Optimized Subsequence Matching** In order to speed up subsequence matching, it is desired to reduce the number of range queries performed in Algorithm 1 without causing any false dismissals. In this regard, we propose two optimizations, namely (1) **Bi-directional subsequence matching** and (2) **Subsequence matching using MaxGap metric**.

**Bi-directional Subsequence Matching** Nodes in a query twig may have different selectivities for range searches, and this may affect the performance of subsequence matching. The cost of subsequence matching can be higher if a node with low selectivity is in the
Algorithm 1: Filtering Algorithm

**Input:** \( Q_s, i, (q_l, q_r) \): \( Q_s \) is a query sequence; index \( i \); \( (q_l, q_r) \) is a range;

**Output:** \((D, S)\); \( D \) is a set of document (tree) identifiers; \( S \) denotes the positions of a subsequence match;

**procedure** \( \text{FindSubsequence}(Q_s, i, (q_l, q_r)) \)

begin

1: \( R \leftarrow \text{RangeQuery}(T_{Q_{si}}, (q_l, q_r)) \); // find the descendants of the current node in the trie
2: \( \text{for each } r \text{ in } R \text{ do} \)
3: \( S_i \leftarrow \text{Level}(r); \) // note the match position
4: \( \text{if } (i = |Q_s|) \text{ then} \)
5: \( D \leftarrow \text{RangeQuery}(\text{DocidIndex}, [r_l, r_r]); \) // find the candidate documents
6: \( \text{output } (D, S); \)
7: \( \text{else} \)
8: \( \text{FindSubsequence}(Q_s, i + 1, (r_l, r_r)); \) // recursively find the subsequences
9: \( \text{endif} \)
10: \( \text{endfor} \)
11: \( \text{end} \)

I propose an optimization called **Bi-directional subsequence matching** to handle such cases. We build a trie “\( LTrie \)” that indexes all the LPS’s of the data trees *from the left to the right*. We build another trie “\( RTrie \)” that indexes the LPS’s of the data trees in the reverse order (i.e., *from the right to the left*). For each \( LTrie \) and \( RTrie \), we build a separate Docid index. (Note that the alternate scheme for indexing sequences described in Section 3.5.2 can be used too [56].) Let \( Q_{sj} \) denote the \( j^{th} \) label in \( Q_s \). \( Q_s \) is the LPS of \( Q \). Let us call \( Q_{sj} \) the pivot. In order to find all subsequences matching \( Q_s \), we first invoke \( \text{FindSubsequence}(\cdot) \) for the sequence \( Q_{sj}, Q_{sj+1}, ..., Q_{sk} \) using \( LTrie \). Next we invoke \( \text{FindSubsequence}(\cdot) \) for the sequence \( Q_{sj}, Q_{sj-1}, ..., Q_1 \) using \( RTrie \). The two partial results are combined to determine all the matching subsequences in the data. After this step, the regular refinement steps are performed.

A query optimizer can use the selectivity information of the labels in \( Q_s \) to determine the pivot for subsequence matching. In the current version of PRIX, the frequency of
occurrence of a label in the collection of Prüfer sequences (constructed over the XML documents) is used to determine the pivot. Thus among the node labels in $Q_s$, we choose the one that has the minimum frequency of occurrence. A simple histogram can be built over the dataset to estimate the frequency of each node label (i.e., elements/attributes/values) in the Prüfer sequence collection for the XML documents. Thus we can avoid low selectivity node labels being searched first during subsequence matching, thereby reducing the total I/O cost. It was observed from the experiments that Bi-directional subsequence matching yielded considerable improvement in performance during query processing.

**Subsequence Matching using MaxGap Metric**  We can prune some nodes ($r$ in line 2 of Algorithm 1) with an additional requirement on the gap between node labels in a query sequence. In this regard, I have developed an upper-bounding distance metric called *MaxGap* based on the property of Prüfer sequences.

Given a collection $\Delta$ of XML document trees and node label $e$ in $\Delta$, I define the distance metric on the pair $(e, \Delta)$ as follows.

**Definition 3.6.1 (MaxGap(e, $\Delta$)).**  Maximum postorder gap of a node label $e$ is defined as the maximum of the difference between the postorder numbers of the first and the last children of the node labeled $e$ in $\Delta$.

For example, in Figure 3.8, the difference in the postorder numbers of the first child and the last child of node label $A$ is $14 - 8 = 6$ in tree $P$ and $3 - 1 = 2$ in tree $Q$. Hence
$MaxGap(A, \{P, Q\})$ is 6. If every occurrence of label $e$ in $\Delta$ has at most one child, then $MaxGap(e, \Delta) = 0$.

I shall now explain the usefulness of this distance metric for subsequence matching. Recall that in Lemma 3.4.2, I showed that the $i^{th}$ node to be deleted during Prüfer sequence construction is the node numbered $i$. Consider tree $P$ in Figure 3.8. The deletion of node 1 (the first child of node 3) corresponds to the first $C$ in LPS($P$). The deletion of node 2 (last child of node 3) corresponds to the second $C$ in LPS($P$). As can be observed in this example, the postorder gap between the first and the last child of a node $e$ denotes how far apart the first and the last occurrence of its label $e$ can be in the sequence. Furthermore, the last occurrence of a node’s label is always followed by its parent node label.

Suppose that a node with label $B$ is the parent of a node with label $C$ in a query twig and $C$, $B$ are adjacent in the query sequence. The $CB$ of this query has eight matches in LPS($P$) (Figure 3.8) which are denoted by position pairs $(1,3)$, $(1,4)$, $(1,7)$, $(1,9)$, $(2,3)$, $(2,4)$, $(2,7)$, and $(2,9)$. Each number pair represents an instance of $CB$ match in the data sequence. Since $MaxGap(C, \{P, Q\})$ is $13 - 10 = 3$, the gap between the first and last occurrences of $C$ in the sequence cannot be more than 3, and the gap between the first occurrence of $C$ and its parent $B$ cannot be more than 4. Among the eight matches listed above, only four position pairs $(1,3)$, $(1,4)$, $(2,3)$, and $(2,4)$ may be considered for further processing. Thus this example illustrates how $MaxGap$ helps to discard certain subsequences that will definitely not be part of the final result.

The following theorem summarizes the use of the $MaxGap$ as an upper-bounding distance metric for pruning the search space and shortening the subsequence matching process.

**Theorem 3.6.2.** Given a query twig $Q$ and the set $\Theta$ of LPS’s for $\Delta$, let $A$ and $B$ denote adjacent labels in LPS($Q$) such that $A$ occurs before $B$.

1. In case node $A$ is a child of node $B$ in $Q$, any subsequence $AB$ in $\Theta$ cannot result in a twig match, if its position pair $(i, j)$ is such that $j - i > MaxGap(A, \Delta) + 1$.

2. In case node $A$ is an ancestor of node $B$ in $Q$, any subsequence $AB$ in $\Theta$ cannot result in a twig match, if its position pair $(i, j)$ is such that $j - i \geq MaxGap(A, \Delta)$. 
Algorithm 2: Filtering Algorithm Using MaxGap Metric

Input: \( \{Q_s, i, (q_l, q_r)\} \): \( Q_s \) is a query sequence; index \( i \); \((q_l, q_r)\) is a range;
Output: \((D, S)\); \( D \) is the document (tree) identifier; \( S \) denotes the positions of a subsequence match;

procedure FindSubsequenceExt\((Q_s, i, (q_l, q_r))\)
begin
1: \( R \leftarrow \text{RangeQuery}(T_{Q_s}, (q_l, q_r)); \) \( // \) find the descendants of the current node in the trie
2: for each \( r \) in \( R \) do
3: \( S_i \leftarrow \text{Level}(r); \) \( // \) note the match position
4: if \( i > 1 \) and \( Q_i-1 \) is a child of \( Q_i \) then
5: \( \text{if } S_i - S_{i-1} > \text{MaxGap}(Q_{i-1}, \Delta) + 1 \) then goto 2; \( // \) discard current \( r \)
else
6: \( \text{if } i > 1 \) and \( Q_{i-1} \) is an ancestor of \( Q_i \) then
7: \( \text{if } S_i - S_{i-1} > \text{MaxGap}(Q_{i-1}, \Delta) \) then goto 2; \( // \) discard current \( r \)
endif
endfor
8: if \( (i = |Q_s|) \) then
9: \( D \leftarrow \text{RangeQuery}(\text{DocidIndex}, [r_l, r_r]); \) \( // \) find the candidate documents
10: output \((D, S)\);
else
11: FindSubsequenceExt\((Q_s, i+1, (r_l, r_r)); \) \( // \) recursively find the subsequences
endif
end

Proof. Part(1): If \( A \) is a child of \( B \) in \( Q \) and in the data, then the first and the last occurrence of \( A \) in the data sequence can occur at a distance of at most \( \text{MaxGap}(A, \Delta) \) apart. The element \( B \) will follow the last occurrence of \( A \). Thus the distance between \( A \) and \( B \) (\( A \) occurs before \( B \)) in the data sequence is at most \( \text{MaxGap}(A, \Delta) + 1 \).

Part(2): If \( A \) is an ancestor of \( B \) in \( Q \) and in the data, then the first and the last occurrence of \( A \) in the data can occur at a distance of at most \( \text{MaxGap}(A, \Delta) \) apart. Since \( B \) is a descendant of \( A \), it occurs between the first and the last occurrence of \( A \) (Lemma 3.4.2). Then the distance between \( A \) and \( B \) (\( A \) occurs before \( B \)) in the data sequence is at most \( \text{MaxGap}(A, \Delta) - 1 \).

It is straightforward to extend Algorithm 1 to incorporate the upper-bounding distance metric by computing \((S_i - S_{i-1})\) after line 3 and testing the appropriate conditions in Theorem 3.6.2 using \( \text{MaxGap} \) of \( Q_{i-1} \). The extended algorithm called \( \text{FindSubsequenceExt} \) is described in Algorithm 2. The algorithm is invoked by \( \text{FindSubsequenceExt}(Q_s, 1, (0, \text{MAX}_\text{INT})). \)
Both “Bi-directional subsequence matching” and “Subsequence matching using Max-Gap metric” can be combined together by simple modifications to the filtering algorithm.

3.6.2 The Refinement Phases in PRIX

The set of ordered pairs \((D, S)\) returned by Algorithm 1 are further examined during the refinement phases. The steps for the refinement phases are shown in Algorithm 3. The goal of the algorithm in its current form is to demonstrate the core ideas during the refinement phases in a simple way. However, the time complexity of the algorithm can be improved by using sorting. The details are deferred until Section 5.12 where I analyze the CPU and I/O costs incurred by PRIX during query processing.

The NPS’s and the set of leaf nodes for the documents in \(D\) are read from the database and passed as input to this algorithm. Each input subsequence is refined by connectedness (Theorem 3.4.5) in lines 1 through 4. Next, the subsequence is refined by structure by testing for gap consistency (Definition 3.4.8) in lines 5 through 11. The subsequence is then tested for frequency consistency (Definition 3.4.11) in lines 12 through 14. Finally, the algorithm matches the leaf nodes of the query twig in lines 15 through 17. This step can be eliminated by special treatment of leaf nodes in the query twigs and the data trees. (Refer to Section 3.6.3). A twig match is reported in line 18. Note that this algorithm does not handle the axis ‘\(//\)’ and wildcard ‘\(*)\’, but it can be easily extended (as mentioned in Section 3.4.5) by replacing lines 1 through 4 by the procedure \(\text{Connectedness}()\) described in Algorithm 4. In Algorithm 4, we follow a series of edges till we reach a node’s ancestor in the data trees. For wildcard ‘\(*)\’, we check if the node’s ancestor can be reached by following two edges (lines 7 through 8). For the axis ‘\(/\)’, we check if the node’s ancestor can be reached by following one or more edges (lines 9 through 11). It is assumed that the procedure \(\text{NodeType}()\) returns the type of wildcard associated with a query node.
Algorithm 3: Refinement Phases

**Input:** \{\(N_D, N_Q, L_D, L_Q, S\)}: \(N_D\) is the NPS of tree \(D\); \(N_Q\) is the NPS of query twig; \(L_D\) is a list of leaves in tree \(D\); \(L_Q\) is a list of leaves in \(Q\); \(S\) denotes the positions of a subsequence match in LPS(\(D\));

**Output:** report twig match;

**procedure** \(\text{RefineSubsequence}(N_D, N_Q, L_D, L_Q, S)\)

```
\begin{align*}
// test for connectedness (refinement by connectedness)
& \text{maxN} \leftarrow \max(N_D[S_1], N_D[S_2], \ldots, N_D[S_S]); // maximum postorder number \\
& \text{for } i = 1 \text{ to } |S| \text{ do} \\
& \quad \text{if } N_D[S_i] \neq \text{maxN AND } \exists (j > i) \text{ s.t. } N_D[S_j] = N_D[S_j] \text{ then} \\
& \quad \quad \text{if } N_D[S_i] \neq S_{i+1} \text{ then return; // check the last occurrence for connectedness} \\
& \text{endfor} \\
// test for gap consistency (refinement by structure) \\
& \text{for } i = 1 \text{ to } |S| - 1 \text{ do} \\
& \quad \text{dataGap} \leftarrow N_D[S_i] - N_D[S_{i+1}];
& \quad \text{queryGap} \leftarrow N_Q[i] - N_Q[i+1];
& \quad \text{if } (\text{dataGap} = 0 \text{ AND } \text{queryGap} \neq 0) \text{ OR } (\text{queryGap} = 0 \text{ AND } \text{dataGap} \neq 0) \text{ then} \\
& \quad \quad \text{return;} \\
& \quad \text{else if } |\text{queryGap}| > |\text{dataGap}| \text{ then return;} \\
& \text{endfor} \\
// test for frequency consistency (refinement by structure) \\
& \text{for } i = 1 \text{ to } |S| \text{ do} \\
& \quad \text{for } j = 1 \text{ to } |S| \text{ AND } j \neq i \text{ do} \\
& \quad \quad \text{if } N_Q[i] = N_Q[j] \text{ AND } N_D[S_i] \neq N_D[S_j] \text{ then return;} \\
& \quad \text{endfor} \\
// match leaves (refinement by matching leaves) \\
// (can be omitted when Extended Prüfer sequences are used) \\
& \text{for each } l \text{ in } L_Q \text{ do} \\
& \quad \text{if } l \text{ not found in } L_D \text{ then} \\
& \quad \quad \text{if } l \text{ not found in LPS/NPS of } D \text{ then return;} \\
& \text{endfor} \\
\text{report twig match;} \\
\text{return;}
\end{align*}
```
Algorithm 4: Refinement by Connectedness with support for Wildcards

Input: \{N_D, N_Q, S\}; \(N_D\) is the NPS of tree \(D\);
\(N_Q\) is the NPS of query \(Q\);
\(S\) denotes the positions of a subsequence match in LPS(\(D\));
Output: true if the subsequence passes the test, false otherwise;

procedure Connectedness(\(N_D, N_Q, S\))
begin
1: \(\text{maxN} \leftarrow \text{max}(N_D[S_1], N_D[S_2], \ldots, N_D[S|S|]);\) // find the maximum postorder number
2: for (\(i = 1\) to |\(S|\)) do
3: \(\text{if } N_D[S_i] \neq \text{maxN AND } \exists (j > i) \text{ such that } N_D[S_i] = N_D[S_j] \text{ then}
4: \(q \leftarrow N_D[S_i];\)
5: \(\text{if } \text{NodeType}(N_Q[i]) = '*' \text{ OR } '//' \text{ then}
6: \(\text{let } N_Q[k] \text{ be the closest ancestor of } N_Q[i] \text{ in } Q \text{ such that } k > i;\)
7: \(\text{if } \text{NodeType}(N_Q[i]) = '*' \text{ then}
8: \(\text{if } N_D[N_D[q]] \neq N_D[S_k] \text{ then return false;}
9: \(\text{else if } \text{NodeType}(N_D[S_i]) = '//' \text{ then}
10: \(\text{while } q < N_D[S_k] \text{ do } q \leftarrow N_D[q];\)
11: \(\text{// check if ancestor has been reached eventually}
12: \(\text{if } q \neq N_D[S_k] \text{ then return false;}
end\)
else
13: \(\text{if } q \neq S_{i+1} \text{ then return false;}
end\)
end
endfor
end

3.6.3 Extended Prüfer Sequences

The Prüfer sequence of a tree as described in Section 3.1 contains only the labels of non-leaf nodes. I call this sequence Regular-Prüfer sequence. If we extend the tree by adding a dummy child node to each of its leaf nodes, the Prüfer sequence of this extended tree will contain the labels of all the nodes in the original tree. I shall refer to this new sequence as Extended-Prüfer sequence. In the case of XML, all the value nodes (strings/character data) in the XML document appear as leaf nodes in the document tree. The document tree is transformed into a sequence after adding dummy child nodes. Similarly, query twigs are also extended before transforming them into sequences. I refer to the index based on
Regular-Prüfer sequences as $\text{RPIndex}$ and the index based on Extended-Prüfer sequences as $\text{EPIndex}$.

Indexing Extended-Prüfer sequences is useful for processing twig queries with values. Since queries with value nodes usually have high selectivities, Extended-Prüfer sequences provide higher pruning power than Regular-Prüfer sequences during subsequence matching. As a result, during subsequence matching, a fewer number of root-to-leaf paths are explored in the virtual trie of $\text{EPIndex}$ than in the virtual trie of $\text{RPIndex}$. If twig queries have no values, then indexing Regular-Prüfer sequences is recommended. Note that Extended-Prüfer sequences are longer than Regular-Prüfer sequences and the increase in length is proportional to the number of leaf nodes in the original tree.

In the PRIX system, both $\text{RPIndex}$ and $\text{EPIndex}$ can coexist. A query optimizer can choose either of the indexes based on the presence or absence of values in twig queries.

Avoiding Refinement by Matching Leaf Nodes I now explain how the final refinement phase of matching leaf nodes can be avoided by special treatments of leaf nodes in a query twig. The key idea is to allow the leaf nodes of a query twig appear in its LPS, so that all the query nodes are examined during the filtering and refinement phases.

For twig queries with value nodes (i.e., CDATA, PCDATA), PRIX uses Extended Prüfer sequences. These value nodes, which are leaf nodes in the original query twig (before extending the twig by adding dummy child nodes), are examined during the subsequence matching phase and the first two refinement phases. The dummy child nodes of the query twig will match the dummy child nodes in the data trees, because the value nodes are extended in a consistent way in both the data trees and query twigs. As a result, the final refinement by leaf nodes phase need not be performed.

On the other hand, for element nodes (i.e., element and attribute tags), which are leaf nodes in the original query twig, I do the following. I extend these nodes in the original query twig with dummy child nodes and perform subsequence matching and the first two refinement phases. However a simple additional step is required at the end of the refinement
phases. Suppose I want to find all occurrences of the query pattern "/\texttt{A}/\texttt{B}/\texttt{C}" in tree $T$ shown in Figure 3.1(a). The LPS of $T$ is $A\ C\ B\ C\ B\ A\ C\ A\ E\ E\ E\ D\ A$ and the NPS is $15\ 3\ 7\ 6\ 6\ 7\ 15\ 9\ 15\ 13\ 13\ 14\ 15$. The LPS and NPS of this query pattern after extending the leaf node $C$ is $C\ B\ A$ and $2\ 3\ 4$ respectively. Two subsequence matches are found in the data with an identical postorder number sequence (i.e., $6\ 7\ 15$) at the end of the refinement phases. One match occurs at positions $4, 6, 7$ and the other match occurs at positions $5, 6, 7$ in the document sequence. However both these matches correspond to the same set of nodes in $T$ i.e., (C, 6), (B, 7) and (A, 15). Hence only one of these matches should be output as a solution. A simple way to achieve this in the above example is to choose the subsequence where label $C$ corresponds to the last occurrence of node 6 in NPS$(T)$ i.e., the match at positions $5, 6, 7$ in the document sequence.

If the element nodes that are leaf nodes in the original query twig have parent-child edges, I assign an upper bounding distance metric of zero to these element nodes. Algorithm \textit{FindSubsequenceExt} discards redundant matches without any modifications. If these element nodes have ancestor-descendant edges, then I discard these redundant matches by using the NPS’s after the refinement phases.

### 3.6.4 Extensions

**Unordered Twig Matches** The PRIX system can be extended to find unordered twig matches by a simple modification of the Prüfer sequence construction. For a given twig query, Prüfer sequences should be constructed for different arrangements of the branches in the query twig, and they should be tested for twig matches. Since the number of twig branches in a query is usually small, only a small number of configurations (arrangements) need to be tested. This process could be implemented efficiently by identifying common prefixes among the sequences for different twig configurations in order to avoid repeated subsequence matching on common prefixes of the sequences. However, not all configurations may yield valid solutions. In the following example I explain how some twig configurations can be quickly eliminated.
Figure 3.9. Different query twig configurations that should be tested for finding unordered twig matches

Example 3.6.3. Consider a query twig $Q = //article[@key="TR"]['title="XML"]/year=2003$. Since attribute tag name key can be considered as a child node of article, $Q$ has 3 root-to-leaf paths (branches). We shall construct the Extended-Prüfer sequence for this query twig. In one twig arrangement, value tag XML appears before value tag 2003 in the LPS of $Q$. In another arrangement, value tag 2003 appears before tag XML in the LPS. If tag 2003 never appears before tag XML in the data sequences, I can eliminate the latter arrangement by ensuring that 2003 does not have XML as its descendant in the virtual trie. This check can be done faster than performing a complete subsequence match. In the case of the root-to-leaf path resulting from attribute tag key, I need not rearrange it w.r.t. other paths. This is because at the time of indexing XML documents, we can always treat the attribute tag key as the first child of article. The query twig can be transformed similarly by treating key as the first child of article. Thus in all, query twig $Q$ has only one arrangement to be tested in which XML appears before 2003 in the LPS of $Q$.

Note that in order to be consistent with the XPath semantics, the twig configurations need to be carefully chosen. For example, let us consider an XPath expression

$\text{//article[author/homepage][author/name]}$. Then the different ordered twig pattern configurations that are possible are shown in Figure 3.9. If the DTD is available, some of the query configurations can be discarded. For example, if author element always has both homepage and name as its subelements, then Prüfer sequences need to be constructed for configurations in Figures 3.9(a) and (b). As a result, the query processing will output the correct matches.

A more general approach to handle such cases is by constructing a structural summary of the input XML document. A structural summary essentially captures the parent-child relationships between tag names, the order among sibling tag names that appear in the
data etc. A structural summary is usually much smaller than the input document and can be constructed by reading the input document once. Using the structural summary, the existence of different configurations can be verified. In addition, for an XPath query such as A[//B][//C], the structural summary can help decide whether B and C have a ancestor-descendant relationship in the data or not. Based on this, an appropriate twig pattern can be constructed.

Another approach to handle unordered twig pattern matching is to break the document order on the siblings and force an ordering based on lexicographic order of their tag names. Given a twig query, the siblings are again ordered in lexicographic order. If a twig has identical sibling tag names, then different ordering of these siblings has to be searched for in the data.

Queries with Inequality and Positional Predicates PRIX can be extended to handle inequality predicates. Consider an XPath query //book[year = “2004”][price > “100”]. This query should return all the book elements whose price is greater than ‘100’ published in the year ‘2004’. To process such queries, I can first find all matches for the pattern //book[year=“2004”][price]. Since the tag-number pair of the leaf nodes in the data trees can be stored in the database, I can fetch them for each matched document and perform the Refinement by Matching Leaf Nodes phase using the leaf node’s tag-number pair (‘100’, 1) with a slight modification. Thus instead of an equality match, I need to perform an inequality match.

Another way to handle inequality predicates is to build a value index for each element (or attribute) for different data types of the values such as integer, text etc. Each value for the element in the document is used as the index key, and the the postorder number of its element and the docid is stored as the data. To process the query //book[year = “2004”][price > “100”], the subquery //book[year=“2004”][price] is first matched. Then the value index for price is range searched for “> 100”. All matching keys are joined with price matches from the subquery evaluation in the documents using
the docid and the postorder number.

Sometimes values are highly selective. In such cases, using the value index the candidate docids can be first obtained. Given an indexing scheme for sequences based on local range assignment (Section 3.5.2), the search for matching subsequences can be performed only on those candidate docids.

PRIX can also be extended to handle queries that contain positional predicates. For example, an XPath query `//book/author[n]`, returns the $n^{th}$ author element under each book element in the XML documents. Since the Prüfer tree-to-sequence transformation preserves all information about the data trees, PRIX can process such queries with additional post-processing.

First, all occurrences of the query pattern `//book/author` can be found in the data using the proposed techniques. Let $(L, N)$ be the LPS and NPS of a document with a match. Let $(a_i, b_i)$ denote the $a_i^{th}$ and $b_i^{th}$ entry in $L$ (and $N$) that match author and book respectively. Let $(a_1, b_1), (a_2, b_2), ..., (a_t, b_t)$ denote all such entry pairs for all the matches of (author, book) in the document. Note that $a_i < b_i$ for $1 \leq i \leq t$ because in the LPS of the query pattern, author appears before book (Lemma 3.4.2). Note that $N[a_i]$ and $N[b_i]$ denote the postorder numbers of the nodes labeled author and book respectively in the input document. First I group each pair $(a_i, b_i)$ based on the value $N[b_i]$. For each group, the pair (match) with the $n^{th}$ smallest $N[a_i]$ value is output. If the positional predicate has a descendant axis (e.g., `//book//author[n]`) then the $(a_i, b_i)$ pairs are first grouped using the value $N[b_i]$ as before. In addition, each group is further grouped into smaller groups using the value $N[N[a_i]]$ (postorder number of the parent of author in the document). From each smaller group, the pair (match) with the $n^{th}$ smallest $N[a_i]$ value is output.

**Updating XML Documents** The problem of updating XML documents is important – a document tree node can be inserted, deleted, or relabeled. Due to the assignment of unique numbers to tree nodes in PRIX, updates are not supported by design. However, I believe
that a reasonable method for updates is still possible. PRIX can be used to index a collection of documents by transforming each document to a sequence. It can also be used to index a single large XML document like DBLP [74] with a small change. Sequences are constructed for each subtree rooted at the child of the document root. In the case of an update, I propose the following method. The old LPS and NPS for the updated tree (or subtree) are deleted from the virtual trie and the collection of NPSs. Note that the virtual trie is a dynamic index. The new LPS and NPS for the updated tree (or subtree) are then inserted into the virtual trie and the collection of NPSs.

If the indexes are built over sequences using local range assignment described in Section 3.5.2, then by using extended preorder numbering [43], the process of updating the LPS can be improved – the entire old LPS need not be deleted before inserting the new LPS into the index. However, the old NPS should be deleted and the new NPS should be inserted into the database.

**Example 3.6.4.** Consider an LPS A B C D. Suppose the elements A, B, C, D in this sequence are indexed using the number ranges [1,100], [20, 80], [30, 70], [40, 60], respectively. If a new element E is inserted such that the new LPS becomes A B E C D. Then E can be assigned a number range [25,75] and the B⁺-tree index for E can be updated. Thus deletion of A, B, C, and D can be avoided from their B⁺-tree indexes. However, the old NPS should be deleted and the new NPS should be inserted into the database.

A follow-up work by Prasad et al. [55], has attempted to support document updates by using modified Prüfer sequences.

**Alternate Strategy for Subsequence Matching** In this section, I shall discuss the use of a different strategy for subsequence matching. The subsequence matching process in PRIX is I/O bound. As the selectivity of the queries reduces, the number of range queries increases due to the increase in the number of paths in the virtual trie that need to be traversed. For such twig queries, the use of Algorithm 1 cannot guarantee a worst-case I/O that is linear in the total number of instances of the tags (of the query pattern) appearing in the Prüfer sequences of the XML data. This is due to the nature of subsequence matching that does random I/O to process the range queries. It is interesting to note that the process of
subsequence matching in a virtual trie is similar to that of finding matches for simple path expressions in an XML document. For example finding all subsequences of the sequence $ABC$ in a virtual trie, is equivalent to finding the path expression $//A//B//C$. As a matter of fact, the PathStack algorithm proposed by [11] can be used for this purpose. The use of PathStack can guarantee a worst-case linear I/O for queries as their selectivity reduces. Furthermore, XB-tree indexes can be built to speed up the subsequence matching process.

3.6.5 Cost Analysis

During query processing in PRIX, the cost of subsequence matching is dominated by I/O. Once the NPS’s are fetched from the database, all the subsequent refinement phases are performed in memory without additional I/O. Therefore, for the purpose of analyzing the query performance of PRIX, I will be focused on the I/O cost of the subsequence matching phase and on the CPU cost of the refinement phases.

It is hard to show that Algorithm 1, which performs range queries, guarantees a worst-case disk I/O cost that is linear in the size of the input lists (data pages in the Trie-Symbol indexes). This is because a data page may be read from the disk more than once during range searches. However, by using the alternate subsequence matching strategy explained in Section 3.6.4, the filtering phase can be shown to have a linear worst-case I/O cost like the TwigStack algorithm. As for the I/O pattern, the data pages accessed by TwigStackXB and TSGeneric$^+$[37] algorithms tend to be scattered, because these algorithms scan input lists through indexes potentially several times in different orders. The I/O pattern of PRIX is also random since the Trie-Symbol indexes for the elements and values are accessed in different orders during the subsequence matching phase.

Our algorithm does not guarantee the optimality of the CPU cost during the refinement phases. This is because a subsequence found in the filtering phase may not be part of the final answer. Note that these subsequences can be stored in memory using a compact stack encoding proposed by [11]. Let $l$ be the length of $LPS(Q)$ and $k$ be the number
of subsequences that match LPS(Q). Suppose $N$ denotes the postorder number sequence of one such matching subsequence $S$. For testing connectedness, sorted versions of both $N$ and $NPS(Q)$ are created in the ascending order using $[\text{postorder number, position in sequence}]$ as the key. Let $m$ be the postorder number and $i$ be the position in the original sequence. Then in the sorted sequence, any key $[n, j]$ to the left of $[m, i]$ satisfies the following property $(n < m) \lor ((n = m) \land (j < i))$. This property holds for every $[m, i]$ except the first key in the sorted sequence. So cost for sorting the $k$ subsequences is $O(k \cdot l \cdot \log(l))$. Using the sorted version of $N$, we can test if the last occurrence of each postorder number in $N$ is followed by its parent (Theorem 3.4.5) in linear time (i.e., $O(l)$). Gap consistency can be tested in linear time using $N$ and $NPS(Q)$. Finally frequency consistency can be tested in linear time using the sorted versions of $N$ and $NPS(Q)$. Overall the CPU cost is $O(k \cdot l \cdot \log(l))$.

Now let us consider the case when ‘//’ is present in the query. As explained in Section 4.5, the respective NPS’s are examined by traversing a series of edges. However the fact that the NPS’s are read in blocks (e.g., heap files), the I/O cost is already accounted for in the $d$ pages. Hence the additional cost for processing ‘//’ in the query is a CPU cost of $O(k \cdot h)$ where $h$ is the height of the document tree. In the case of ‘*’, the query processing incurs an additional CPU cost of $O(k \cdot b)$ where $b$ is the number of child nodes of the ‘*’ node.

### 3.7 Implementation Issues

In this section, I describe the implementation issues in PRIX. (Refer to Figure 3.2 for an architectural overview of the system.)

#### 3.7.1 Document Parsing

An XML document is first parsed using an event-driven SAX parser. The output of the SAX parser is a stream of start and end tags for each element, attribute, and value in
document order. The SAX parser output is then input to the indexing engine.

3.7.2 Indexing Engine

The elements, values, and attributes are first mapped to unique integers, because it is more efficient to store and process integers than raw strings. The indexing engine constructs Prüfer sequences for XML documents by reading their parsed output. Given a well-formed XML document T, I assume that its SAX parser output stream \( T_{sax} \) has functions \( next(\cdot) \) and \( eof(\cdot) \) associated with it. The function \( T_{sax}.next() \) returns the next line in the stream \( T_{sax} \) i.e., tag name and tag type (\textbf{start} or \textbf{end}). The function \( T_{sax}.eof() \) checks for the end of the stream. I also assume that a function \( hash(\cdot) \) uniquely maps every element, attribute and value in the input data to a number. Algorithm 5 describes the steps involved in constructing Regular Prüfer sequences. The algorithm recursively traverses the tree structure and numbers the tree nodes in postorder starting from 1. The algorithm outputs two sequences \((L, N)\) representing the LPS and NPS of the document T. In the algorithm, once a non-leaf node is numbered, the entries corresponding to the deletion of its child nodes in the LPS and NPS are updated (lines 20 through 23). The above algorithm can be extended to construct Extended Prüfer sequences by making small modifications. Line 18 should be replaced with the following lines: \( \text{numChildren} = 1; \text{Stk.push(nodeID)}; \text{nodeId} = \text{nodeId} + 1; \). Depending on the query workload, the indexing engine can choose to build Regular and/or Extended Prüfer sequences and index them (i.e., \( \text{RPIndex} \) and \( \text{EPIndex} \)).

The purpose of the stack in Algorithm 5 is to store the postorder numbers assigned to a node’s children. Once a node is assigned a postorder number, all its children are popped out of the stack. As a result, the maximum depth of the stack is upper-bounded by the maximum fanout of the XML document tree. Moreover, once a node is numbered, the LPS and NPS for the subtree rooted at that node can be generated completely. However, only when the root node is numbered, the LPS and NPS for the entire tree can be generated. If memory is a concern, then we could first write the partial sequences for the subtree
Algorithm 5: Constructing Prüfer Sequences

Input: \(\{T_{\text{sax}}\}\): SAX parser output for a well-formed XML document \(T\)
Output: \(\{(L, N)\}\): L - LPS of \(T\); N - NPS of \(T\)

Global data structures: stack \(\text{Stk}\); integer \(\text{nodeID}\), \(\text{count}\);

procedure \(\text{RegularPrüfer}(T_{\text{sax}})\)
begin
\(\text{Stk}.\text{clear}();\) // clean up the stack
\(\text{nodeID} \leftarrow 1;\) // postorder numbering starts from 1
\(\text{count} \leftarrow 0;\)
\((\text{tagName}, \text{type}) \leftarrow T_{\text{sax}}.\text{next}();\) // read the first line of the SAX output
\(\text{GenerateSequence}(T_{\text{sax}}, (\text{tagName}, \text{type}));\)
\(\text{return} (L, N);\)
end

procedure \(\text{GenerateSequence}(T_{\text{sax}}, (\text{tagName}, \text{type}))\)
begin
\(\text{numChildren} \leftarrow 0;\) // keeps track of the number of children of \(\text{tagName}\)
while \(\text{true}\) do
\(\text{if} T_{\text{sax}}.\text{eof}() \text{ then} \text{return};\)
\((\text{tagNameNext}, \text{typeNext}) \leftarrow T_{\text{sax}}.\text{next}();\)
\(\text{if} \text{typeNext} = \text{END} \text{ then} \text{break};\) // end tag encountered
\(\text{GenerateSequence}(T_{\text{sax}}, (\text{tagNameNext}, \text{typeNext}));\) // recursively traverse the tree
\(\text{Stk}.\text{push}($\text{nodeID}$);\) // store the node’s postorder number
\(\text{nodeID} \leftarrow \text{nodeID} + 1;\)
\(\text{numChildren} \leftarrow \text{numChildren} + 1;\)
endw
// If the leaf node is reached, store it
\(\text{if} \text{numChildren} = 0 \text{ then}\)
\(\text{count} \leftarrow \text{count} + 1;\)
\(\text{store the leaf’s (hash(tagName), count) pair};\)
\(\text{return};\)
endif
// Once a non-leaf node is numbered, update entries in the
// LPS/NPS corresponding to the deletion of the node’s children
\(\text{for} \ k = 1 \text{ to numChildren do}\)
\(\text{N}[\text{Stk}.\text{top}()] \leftarrow \text{count} + 1;\)
\(\text{L}[\text{Stk}.\text{top}()] \leftarrow \text{hash(tagName)};\)
\(\text{Stk}.\text{pop}();\)
endfor
\(\text{count} \leftarrow \text{count} + 1;\)
\(\text{return};\)
end
rooted at the first child of the root to disk followed by a blank entry that would be filled when the root is assigned a postorder number. This process can be repeated for the other subtrees that follow. Once the root is numbered, the blank entries in the sequences are filled by performing random disk I/O. However, this may not always be necessary. PRIX can be used to index both a collection of XML documents as well as a single large XML document (e.g., DBLP [74]). In the case of a single large XML document, we need not construct a single Prüfer sequence. Rather we could split the document tree at the root and construct sequences for the subtrees rooted at the child nodes of the root. These sequences can then be indexed.

In order to support fast subsequence matching, the LPS’s are indexed by building a virtual trie using B+-trees. The nodes in the trie are assigned number ranges to support containment queries. Note that suffixes of the LPS’s need not be stored since all subsequences can be found using Algorithm 1. For each unique label, a B+-tree is built to store the number ranges of all the instances of that label in the virtual trie. The NPS’s and the leaf nodes (if used) are stored in the database (e.g., as records in a heap file).

3.7.3 Querying Engine

The LPS and NPS for an XPath query are constructed. All subsequences that match the query’s LPS in the data sequences are found by searching the virtual trie. The document identifiers of the data trees that have a matching subsequence are also determined. Based on the nature of the query, the querying engine can either choose the RPIndex or the EPIndex for the subsequence match phase. Post-processing is performed on the matching subsequences to discard non-matches (i.e., false alarms). The NPS’s corresponding to the matching document identifiers are fetched from the database and are used for the refinement-by-connectedness and refinement-by-structure phases. At the end of the refinement phases, all occurrences of the query twig are output.
3.8 Experimental Evaluation

In my experiments, I compared the query performance of PRIX, TwigStackXB, and TSGeneric+ [37] for a set of high selectivity queries. I implemented PRIX, and TwigStack/TwigStackXB in C++, and used the $B^+$-tree implementation of GiST [34] for all their indexes. The implementation of TSGeneric+ algorithm was obtained from Jiang et al. Since their code was developed on the Microsoft Windows platform, I compared PRIX and TSGeneric+ in terms of disk I/O. Note that the disk I/O for all these algorithms is random in nature (Section 3.6.5). PRIX was compared with ViST earlier [60] and outperformed it considerably.

PRIX and TwigStackXB/TSGeneric+ are suited for two different application domains. PRIX supports ordered twig pattern matching inherently. TwigStackXB and TSGeneric+ support unordered pattern matching. These algorithms can be adapted for ordered pattern matching by performing a post processing step to verify order among siblings. On the other hand, PRIX can be adapted for unordered pattern matching by testing the different twig pattern configurations that result from different sibling orders.

3.8.1 Experimental Setup

I ran my experiments for PRIX and TwigStack/TwigStackXB on 1.8GHz Pentium IV processor with 512 MB RAM running Solaris 8. A 120GB EIDE disk drive was used to store the data and indexes. The code was compiled using the GNU g++ compiler version 2.95.3. Direct I/O feature available on Solaris was enabled to avoid operating system’s cache effects. For TSGeneric+, the code was compiled using Microsoft Visual C++ compiler version 6.0. The experiments were run on the Microsoft Windows XP platform. For all the experiments, the buffer pool size was fixed at 2000 pages. The page size of 8 KB was used. For PRIX, 4-byte number ranges were used to label the nodes in the virtual trie. For TwigStack/TwigStackXB and TSGeneric+, the same 4-byte number ranges were used to label the nodes in the XML document trees.
Table 3.2. Datasets used for evaluation

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Size (in MB)</th>
<th># of Elements</th>
<th># of Attributes</th>
<th>Maximum Depth</th>
<th># of Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWISSPROT</td>
<td>115</td>
<td>2977031</td>
<td>2189859</td>
<td>5</td>
<td>50000</td>
</tr>
<tr>
<td>TREEBANK</td>
<td>86</td>
<td>2437666</td>
<td>1</td>
<td>36</td>
<td>56385</td>
</tr>
<tr>
<td>DBLP</td>
<td>134</td>
<td>3332130</td>
<td>404276</td>
<td>6</td>
<td>328858</td>
</tr>
</tbody>
</table>

Table 3.3. XPath Queries for SWISSPROT Dataset

Datasets I experimented with the datasets shown in Table 3.2. These datasets were obtained from the University of Washington XML repository [74]. I chose these three datasets since each had a different characteristic. The document trees in the SWISSPROT dataset were bushy and shallow. The document trees in the DBLP dataset had high similarity in structure and were shallow. The document trees in the TREEBANK dataset were narrow and had deep recursion of element names. Table 3.2 provides additional information such as the maximum depth, number of elements for the datasets. PRIX can be used to index and query a collection of XML documents or a single large XML document. In the case of a single large XML document tree (e.g., SWISSPROT) we remove the root element and transform the collection of subtrees into sequences. The sequences are then indexed using a disk-based virtual trie. Table 3.2 also shows the number of sequences constructed for each dataset.
The twig queries used for my experiments are shown as XPath expressions in Tables 3.3, 3.4, and 3.5. To avoid the frequent use of the axes like `following-sibling` in these tables, I assume that the order between the siblings in a twig query follows the left-to-right order in the corresponding XPath expression. For example, `//phdthesis[year][number]` indicates that `year` is followed by `number`. The listed queries have different characteristics in terms of selectivity, presence of values and twig structure. For the TREEBANK dataset, since the values were encrypted, I chose queries without values (character data). Tables 3.3, 3.4, and 3.5 also show the number of twig matches for each query. In my work, I focus on ordered twig pattern matching that is useful for applications that require...
the query nodes to follow the document order in XML. Note that the query processing time for PRIX, TwigStackXB and TSGeneric+ increases as the number of matches increases. For fairness of comparison, the number of ordered and unordered twig matches for each twig query were the same, i.e., only one query twig configuration was present in the data trees.

3.8.2 Performance Analysis

In this section, I analyze the query performance of PRIX, TwigStackXB and TSGeneric+ algorithms for high selectivity queries listed above.

The TwigStack algorithm examines every node in the sorted input stream. On the other hand, TwigStackXB uses XB-Trees to skip nodes in the sorted input stream. The effectiveness of XB-Trees depends on the distribution of possible twig matches in the input streams. If the possible solutions are localized in certain regions of the input streams, then XB-Trees are effective in skipping large portions of the input streams. However, if the possible solutions are scattered in the input streams then TwigStackXB is forced to drill down frequently to the lower regions of the XB-Trees to identify true matches. As mentioned in Section 2.1.2, TwigStack algorithms are sub-optimal for query patterns with parent/child relationships.

The TSGeneric+ algorithm (partly motivated by TwigStack algorithms) was proposed by [37] to further improve the skipping of elements by using available indexes on the elements. The algorithm uses XR-trees [36] that is based on the concept of interval trees. Using XR-trees, both ancestors and descendants of a given element can be determined efficiently. The TSGeneric+ algorithm takes advantage of high join selectivity edges in the query pattern to skip elements in the input lists. The algorithm uses different strategies for picking edges so that the skipping of elements in other edges of the query pattern can be maximized. In my experiments, I compared PRIX with the version of TSGeneric+ strategy that incurred the minimum I/O cost for fairness. Similar to TwigStack algorithms, TSGeneric+ is sub-optimal for query patterns with parent/child relationships.
Table 3.6. SWISSPROT - I/O cost and total time - TwigStack vs TwigStackXB

<table>
<thead>
<tr>
<th>Query</th>
<th>TwigStack</th>
<th>TwigStackXB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total time</td>
<td>Disk IO</td>
</tr>
<tr>
<td>$Q_1$</td>
<td>26.53 secs</td>
<td>4543 pages</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>35.39 secs</td>
<td>5563 pages</td>
</tr>
<tr>
<td>$Q_8$</td>
<td>12.88 secs</td>
<td>1715 pages</td>
</tr>
</tbody>
</table>

The performance of PRIX is dominated by the I/O cost incurred during subsequence matching. The fewer paths are traversed in the virtual trie, the fewer disk pages are accessed. I have proposed optimizations (in Section 3.6.1) that can speed up the subsequence matching phase by reducing the number of range searches required to find all the subsequences. It should be noted that the query processing cost also depends on the number of NPS’s that are read from the database.

SWISSPROT Dataset

PRIX vs TwigStackXB - For the SWISSPROT dataset, I observed that TwigStack performed worse than TwigStackXB for queries in Table 3.3. The elapsed time and the disk I/O for some of the queries are shown in Table 3.6.

In Figure 3.10, the elapsed time for processing the queries in Table 3.3 using PRIX and TwigStackXB are plotted. PRIX used EPIndex to process the queries. For queries $Q_1$, $Q_4$, and $Q_8$,
Figure 3.11. SWISSPROT - I/O cost comparison - (PRIX vs TSGeneric+)

<table>
<thead>
<tr>
<th>Query</th>
<th>PRIX</th>
<th>TwigStackXB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total time</td>
<td>Disk IO</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>0.82 secs</td>
<td>84 pages</td>
</tr>
<tr>
<td>$Q_5$</td>
<td>1.22 secs</td>
<td>121 pages</td>
</tr>
<tr>
<td>$Q_6$</td>
<td>1.22 secs</td>
<td>127 pages</td>
</tr>
<tr>
<td>$Q_7$</td>
<td>3.04 secs</td>
<td>411 pages</td>
</tr>
</tbody>
</table>

Table 3.7. SWISSPROT - Total time - PRIX vs TwigStackXB

$Q_3$, $Q_8$ and $Q_9$, PRIX and TwigStackXB had comparable performance. PRIX performed subsequence matching beginning with nodes with high selectivity due to bottom-up transformation of the query twigs. TwigStackXB found all matches by skipping large sections of the sorted input streams using XB-Trees.

TwigStackXB was faster than PRIX for $Q_2$ since a large number of matches were clustered in certain regions in the sorted streams. As a result, XB-Trees were effective in reducing the I/O cost.

PRIX processed queries $Q_4$, $Q_5$, $Q_6$ and $Q_7$ faster than TwigStackXB. Table 3.7 shows the elapsed time and disk I/O for these queries. We shall first analyze the performance of queries $Q_4$ and $Q_5$. The value node “PF00304” was scattered in the input dataset. However about half the documents containing //Entry/PFAM[@prim_id=“PF00304”] did not have the pattern //DISULFID/Descr. However //DISULFID/Descr occurred
very frequently in the dataset and as a result TwigStack frequently drilled down to the lower regions of the XB-Trees to eliminate false matches for processing $Q_4$. This resulted in an increase in disk I/O. The situation was similar for query $Q_5$ due to the pattern //SIGNAL/Descr.

PRIX on the other hand was two to three times faster than TwigStackXB for queries $Q_4$ and $Q_5$. Recall that the subsequence matching phase depends on the number of candidate ranges that are searched in the virtual trie to find all the matching subsequences. For $Q_4$, the subsequence matching was performed beginning with the node “PF00304” due to bottom-up transformation of the query twig. The partial matching documents were eliminated during the filtering stage. For $Q_5$, Bi-directional subsequence matching was performed using “PF00304” as the pivot since the tag $\text{Org}$ had low selectivity. PRIX was slower processing $Q_5$ as compared to $Q_4$ due to the use of both the $L\text{Trie}$ and $R\text{Trie}$ which increased the disk I/O.

Next I shall analyze the performance of query $Q_6$. The patterns //Entry/Organelle=“Chloroplast” and //Entry[Species=“Vicia faba”][Org=“Vicia”] were scattered in the dataset. However, not all documents that had the pattern //Entry[Species=“Vicia faba”][Org=“Vicia”] had //Entry/Organelle=“Chloroplast”. Since these patterns occurred in nearby documents in the data, TwigStack had to access lower level regions of the XB-Trees frequently to skip such document resulting in an increase in disk I/O.

PRIX on the other hand eliminated such matches during subsequence matching by starting with high selectivity nodes due to the bottom-up transformation of the query pattern. As a result fewer paths were traversed in the virtual trie. Similar behavior was observed for query $Q_7$ and PRIX was nearly two times faster than TwigStackXB.

PRIX vs $\text{TSGeneric}^+$ - The performance results from PRIX and TSGeneric$^+$ in terms of disk I/O for the SWISSPROT dataset are plotted in Figure 3.11. Recall that TSGeneric$^+$ can efficiently process twig queries by skipping both ancestors and descendants if the join
selectivities of (at least one of) the twig pattern edges are high. For example, the edge
Author="Price S.R" in $Q_1$ had high join selectivity. On a similar note, PRIX can
also process queries efficiently if they have high selectivities since only a few paths in the
virtual trie are accessed. Since queries $Q_1$, $Q_2$, $Q_3$, $Q_8$ and $Q_9$ had the above characteristics,
both PRIX and TSGeneric$^+$ had comparable performances for them.

Of special interest is the performance of queries $Q_6$ and $Q_7$. Let us first analyze the
query $Q_7$. The pattern //Entry/Organelle="Chloroplast" was scattered in
the dataset. In addition, the patterns //Entry/Species="Glycine max" and
//Entry/Org="Glycine" occurred less frequently than //Entry/Organell-
e="Chloroplast". The edge join selectivities for $Q_7$ was overall lower than that
of queries like $Q_1$, $Q_2$, etc. As a result, the effectiveness of skipping elements using the
XR-tree indexes was diminished and more elements in the input lists were scanned. Overall
the I/O cost was significantly higher than that of PRIX. Similarly, for $Q_6$ PRIX was more
efficient than TSGeneric$^+$. We can draw the conclusion that the distribution of probable
solutions for different branches of the twig pattern can reduce the effectiveness of XR-trees
to skip elements in the input lists. As for $Q_5$, due to the presence of high join selectivities
in the edges (i.e., @prim_id="PF00304") of $Q_5$, the query was processed efficiently
by TSGeneric$^+$. In contrast, due to the use of Bi-directional subsequence matching, PRIX
did not take advantage of the high join selectivities and was outperformed by TSGeneric$^+$.

**TREEBANK Dataset**

**PRIX vs TwigStackXB** - For the TREEBANK dataset, I observed that TwigStack performed
worse than TwigStackXB for the queries in Table 3.4. Table 3.8 shows the elapsed time
and disk I/O for some of the queries. Note that the TREEBANK dataset had deep recursion
of element names.

In Figure 3.12 the elapsed time and disk I/O for the queries in Table 3.4 using PRIX
and TwigStackXB are plotted. For PRIX, RPIndex was used to process all the queries.
TwigStackXB and PRIX had comparable performance for queries $Q_{10}$, $Q_{11}$, $Q_{13}$, $Q_{14}$,
<table>
<thead>
<tr>
<th>Query</th>
<th>TwigStack</th>
<th>TwigStackXB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total time</td>
<td>Disk IO</td>
</tr>
<tr>
<td>$Q_{10}$</td>
<td>9.34 secs</td>
<td>1426 pages</td>
</tr>
<tr>
<td>$Q_{11}$</td>
<td>6.55 secs</td>
<td>1060 pages</td>
</tr>
<tr>
<td>$Q_{15}$</td>
<td>14.01 secs</td>
<td>3008 pages</td>
</tr>
</tbody>
</table>

**TABLE 3.8. TREEBANK - TwigStack vs TwigStackXB**

**Figure 3.12. TREEBANK - Total time (PRIX vs TwigStackXB)**

**Figure 3.13. TREEBANK - I/O cost comparison - (PRIX vs TSGeneric +)**
For query $Q_{13}$, Bi-directional subsequence matching was performed with ADVP-1 as the pivot. This resulted in speeding up the filtering process. Note that for query $Q_{15}$, PRIX was slower than TwigStackXB. Since the twig matches were localized in certain regions of the sorted input stream, TwigStackXB could skip large regions of data using XB-Trees.

As stated in Section 2.1.2, TwigStack algorithms are sub-optimal for parent-child relationships in the query twigs. The performance of queries $Q_{12}$, $Q_{16}$, $Q_{17}$, $Q_{18}$ and $Q_{19}$ demonstrate this behavior. I observed that PRIX was two to three times faster than TwigStackXB in many instances. In Table 3.9 the elapsed time and disk I/O are shown for these queries.

Let us first analyze the performance of query $Q_{12}$. In query $Q_{12}$ the tag $\mathtt{PRT}$ appeared several times and was scattered in the input stream. However only in two such documents, tags $\mathtt{S}$ and $\mathtt{PRT}$ shared a parent-child relationship. In many other documents, $\mathtt{S}$ was an ancestor of $\mathtt{PRT}$. As a result of sub-optimality, TwigStackXB first found all such matches and often drilled down to the leaf nodes of the XB-Trees since these matches were not localized in certain regions (but scattered) of the sorted input streams. This process increased the disk I/O. These false matches are discarded during the merge post-processing step.

PRIX on the other hand was faster than TwigStackXB and performed (optimized) subsequence matching using the high selectivity node $\mathtt{PRT}$ and used the MaxGap of the node $\mathtt{PRT}$ to discard those documents with $\mathtt{S}$ as an ancestor of $\mathtt{PRT}$. The MaxGap of $\mathtt{PRT}$ was 0 in this case. PRIX discarded the false matches early in the subsequence matching phase and fewer disk pages were accessed overall.

<table>
<thead>
<tr>
<th>Query</th>
<th>PRIX</th>
<th>TwigStackXB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total time</td>
<td>Disk IO</td>
</tr>
<tr>
<td>$Q_{12}$</td>
<td>1.82 secs</td>
<td>376 pages</td>
</tr>
<tr>
<td>$Q_{16}$</td>
<td>1.15 secs</td>
<td>168 pages</td>
</tr>
<tr>
<td>$Q_{17}$</td>
<td>1.16 secs</td>
<td>171 pages</td>
</tr>
<tr>
<td>$Q_{18}$</td>
<td>1.20 secs</td>
<td>174 pages</td>
</tr>
<tr>
<td>$Q_{19}$</td>
<td>2.59 secs</td>
<td>390 pages</td>
</tr>
</tbody>
</table>

**Table 3.9. TREEBANK - PRIX vs TwigStackXB**
Similar was the case for queries $Q_{16}$, $Q_{17}$, $Q_{18}$ and $Q_{19}$ and PRIX was faster than TwigStackXB. The tag \texttt{NEG} had high selectivity in these queries. Note that PRIX used Bi-directional subsequence matching for $Q_{19}$ with \texttt{NEG} as the pivot since the tag \texttt{RB} had low selectivity. The MaxGap values for \texttt{RB} and \texttt{NEG} were 0 and 2 respectively. As expected PRIX took more time to process query $Q_{19}$ as compared to $Q_{17}$ because to process $Q_{19}$ both the \texttt{LTrie} and \texttt{RTrie} indexes were used for subsequence matching.

**PRIX vs TSGeneric$^+$** - The I/O cost for PRIX and TSGeneric$^+$ for the TREEBANK dataset are shown in Figure 3.13. The queries $Q_{10}$, $Q_{11}$, $Q_{13}$, $Q_{14}$, $Q_{15}$, $Q_{16}$, $Q_{20}$ and $Q_{21}$ had comparable performance for both PRIX and TSGeneric$^+$. This was because for these queries, some edges had high join selectivities. For example, in query $Q_{10}$ the edge \texttt{S/SYM} had high join selectivity. As a result, TSGeneric$^+$ was able to effectively skip elements using the XR-tree indexes. Similarly, PRIX used high selectivity nodes for subsequence matching and processed these queries efficiently.

Similar to TwigStackXB, TSGeneric$^+$ suffers from sub-optimality for parent-child edges in the queries. TSGeneric$^+$ first found all the ancestor-descendant matches and the non-matches that do not satisfy parent-child relationship were eliminated during postprocessing. However, since the dataset contained many such non-matches, TSGeneric$^+$ incurred considerably more I/O than PRIX to process queries $Q_{12}$, $Q_{17}$, $Q_{18}$. On the other hand, PRIX performed optimized subsequence matching using the MaxGap values for query nodes and was able to prune out the non-matches early during query processing. Thus PRIX outperformed TSGeneric$^+$ for these queries.

**DBLP Dataset**

**PRIX vs TwigStackXB** - As with the other datasets, I observed that TwigStack performed worse than TwigStackXB for the queries in Table 3.5. Table 3.10 summarizes the performance for some of the queries in the DBLP dataset. Since the queries had high selectivity,
TwigStackXB was effective in skipping a lot of the data in the input streams, thereby resulting in fewer disk accesses.

In Figure 3.14, the elapsed time for processing the queries using PRIX and TwigStackXB are plotted. PRIX used EPIndex for answering all the queries in Table 3.5. For queries $Q_{22}$, $Q_{23}$, $Q_{24}$, and $Q_{28}$, PRIX and TwigStackXB had comparable performance. As expected, TwigStackXB processed these queries efficiently, because the solutions for those queries were distributed in certain regions of the data and the XB-Trees were effective in skipping nodes in the input streams. On the other hand, PRIX also processed these queries efficiently by performing subsequence matching beginning with the high selectivity element tags/values. The bottom-up transformation of the query twig caused higher selectivity nodes to appear in beginning of its LPS.

For queries $Q_{25}$ and $Q_{29}$ TwigStackXB was faster than PRIX. For $Q_{25}$, most solutions were clustered in small regions of the input stream allowing TwigStackXB to find all the matches faster as compared to PRIX.

<table>
<thead>
<tr>
<th>Query</th>
<th>TwigStack</th>
<th>TwigStackXB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total time</td>
<td>Disk IO</td>
</tr>
<tr>
<td>$Q_{22}$</td>
<td>26.77 secs</td>
<td>6409 pages</td>
</tr>
<tr>
<td>$Q_{26}$</td>
<td>15.50 secs</td>
<td>3036 pages</td>
</tr>
</tbody>
</table>

**TABLE 3.10. DBLP - I/O cost and total time - TwigStack vs TwigStackXB**

**FIGURE 3.14. DBLP - Total time - (PRIX vs TwigStackXB)**
For query Q29, the Bi-directional subsequence matching was performed using the LTree and RTree with value “Jim Gray” as the pivot since the element key occurred in every document in the DBLP dataset. This increased the total IO as compared to processing Q28 which also had the same number of twig matches in the data.

For queries Q26 and Q27 PRIX was two to three times faster than TwigStackXB. The tag phdthesis was scattered in the sorted input stream. Also the tags year, series and number appeared frequently in other nearby documents that did not contain tag phdthesis. In order to eliminate such matches TwigStack frequently drilled down to lower regions of the XB-Trees. This increased the disk I/O. In PRIX, the cost of subsequence matching phase depends on the number of candidate ranges that are searched. Since PRIX used Bi-directional subsequence matching for processing these queries using tag phdthesis as the pivot, all matches were found by performing fewer disk I/O. Note that the tag phdthesis had high selectivity as compared to other nodes in the queries.
$Q_{26}$ and $Q_{27}$.

**PRIX vs TSGeneric$^+$** - The performance of PRIX and TSGeneric$^+$ for queries $Q_{22}$ through $Q_{29}$ are shown in Figure 3.15. The plot compares the disk I/O incurred by the PRIX and TSGeneric$^+$.

For queries $Q_{22}$ and $Q_{25}$, both PRIX and TSGeneric$^+$ had comparable performance. For example, in $Q_{22}$, PRIX used the high selectivity node ‘Antonin Guttman’ to search for matching subsequences by traversing few paths in the virtual trie. On the other hand, the high join selectivity edge author=‘Antonin Guttman’ proved useful for TSGeneric$^+$ to skip many elements in the input lists.

Next I shall analyze the performance of queries $Q_{23}$ and $Q_{24}$. It is interesting to note that PRIX only incurred about half the I/O as compared to TSGeneric$^+$. The subsequence matching approach used by PRIX was far more effective than using the XR-trees indexes even though a high join selectivity edge (i.e., author=‘C. J. Date’) was present in both queries. It is also interesting to note the performance of queries $Q_{26}$ and $Q_{27}$. PRIX processed the queries efficiently using Bi-directional subsequence matching with phdthesis as the pivot. TSGeneric$^+$ also processed the queries efficiently due to the presence of high join selectivity edges in these queries. Finally for queries $Q_{28}$ and $Q_{29}$, I observed an increase in the disk I/O using PRIX for $Q_{29}$ as compared to $Q_{28}$ due to Bi-directional subsequence matching. Note that the performance trends for $Q_{26}$ and $Q_{27}$ were similar for TwigStackXB and TSGeneric$^+$.

3.8.3 Evaluation of Bi-directional Subsequence Matching

To provide an insight to the effectiveness of Bi-directional Subsequence Matching, I compared Bi-directional subsequence matching and naive subsequence matching (from left to right) in terms of the disk I/O required to process queries $Q_5$, $Q_{13}$, $Q_{19}$, $Q_{26}$, $Q_{27}$ and $Q_{29}$ as shown in Tables 3.3, 3.4 and 3.5. The results are provided in Table 3.12. Note that the number of buffer pages was fixed at 2000.
TABLE 3.12. Improvements in disk I/O during subsequence matching using Bi-directional subsequence matching

<table>
<thead>
<tr>
<th>Query</th>
<th>Bi-directional</th>
<th>Naive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_5$</td>
<td>118 pages</td>
<td>98,645 pages</td>
</tr>
<tr>
<td>$Q_{13}$</td>
<td>130 pages</td>
<td>1,062 pages</td>
</tr>
<tr>
<td>$Q_{19}$</td>
<td>380 pages</td>
<td>444 pages</td>
</tr>
<tr>
<td>$Q_{26}$</td>
<td>19 pages</td>
<td>2,465 pages</td>
</tr>
<tr>
<td>$Q_{27}$</td>
<td>17 pages</td>
<td>2,540 pages</td>
</tr>
<tr>
<td>$Q_{29}$</td>
<td>325 pages</td>
<td>609,102 pages</td>
</tr>
</tbody>
</table>

TABLE 3.13. Maximum CPU costs incurred by the refinement phases

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>CPU Time</th>
<th>% of Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWISSPROT</td>
<td>$Q_7$</td>
<td>0.27 secs</td>
<td>8.86%</td>
</tr>
<tr>
<td>TREEBANK</td>
<td>$Q_{19}$</td>
<td>0.003 secs</td>
<td>0.10%</td>
</tr>
<tr>
<td>DBLP</td>
<td>$Q_{25}$</td>
<td>0.04 secs</td>
<td>4.70%</td>
</tr>
</tbody>
</table>

Overall, it was observed that the I/O cost during subsequence matching improved drastically when the Bi-directional subsequence matching was used. This clearly shows that it is essential to start with a node (pivot) with high selectivity. For example, in $Q_5$, the tag $\text{Org}$ had a very high frequency of 456,398 occurrences in the SWISSPORT dataset. Hence the number of paths explored in the virtual trie was very large. Similarly, the tag $\text{key}$ in $Q_{29}$ had a very high frequency of 328,858 occurrences in the DBLP dataset and showed similar trend. Overall I observed that Bi-directional subsequence matching improved the query processing performance of PRIX drastically.

3.8.4 CPU Costs For the Refinement Phases

I measured the CPU time for the queries listed in Tables 3.3, 3.4, and 3.5. For each dataset, I only show the query which had the maximum CPU cost in Table 3.13. The portion of the total time spent during the refinement phases is also shown in Table 3.13. I observed that the I/O cost dominated the total query processing time for the queries that I tested.
3.8.5 Summary of Performance Analysis

To summarize, I observed that PRIX yields good performance for processing queries with high selectivity. The query processing cost is dominated by the subsequence matching phase that is I/O bound. I observed that my proposed optimizations in Section 3.5 were effective in reducing the I/O cost during filtering. PRIX by virtue of the bottom-up tree transformation used high selectivity nodes to start searching the virtual trie for many queries. In some cases, PRIX used “Bi-directional subsequence matching” by choosing a pivot with high selectivity. As a result, few paths in the virtual trie were traversed thereby reducing the total processing time.

3.9 Other Applications of Sequencing XML Documents and Twig Patterns

3.9.1 FiST: Scalable XML Document Filtering by Sequencing Twig Patterns

The publish-subscribe (pub-sub) systems play an important role in e-commerce and Internet applications by enabling selective dissemination of information. In a typical pub-sub system, whenever new content is produced, it is selectively delivered to interested subscribers. The pub-sub systems have enabled new services such as alerting and notification services for users interested in knowing about the latest products in the market, current affairs, stock price changes, etc on a variety of devices like mobile phones, PDAs and desktops. Such services necessitate the development of software systems that enable scalable and efficient matching of a large number of items to deliver against a potentially large number of subscribed users.

Nowadays, we come across many e-commerce sites that provide email notifications to subscribers about price changes and hot deals. For example, a recent service by Google, called Google Alerts, provides email updates of the latest based on topics of choice to subscribed users. Users can choose to receive notifications by selecting a topic and providing a list of search keywords. Another interesting example is the stock quote tracking service
FIGURE 3.16. XML-based publish-subscribe system

provided by Yahoo. Evidently, there is a growing use and demand for large-scale systems
for selective information dissemination.

The popularity of the XML (eXtensible Markup Language) as a standard for informa-
tion exchange has triggered several research efforts to build scalable XML filtering systems,
where subscribers’ interests are stored in their profiles typically expressed in the XPath lan-
guage [7].

Note that the problem of filtering XML documents is fundamentally different from the
problem of finding all occurrences of a twig pattern in an XML document. This is due
to the reversal in the roles of twig patterns and XML documents. Informally, the filtering
problem is stated as follows.

Given a set of XPath expressions, identify such XPath expressions that match
each of input XML documents to deliver.

As shown in Figure 3.16, in an XML-based pub-sub system, each incoming XML doc-
ument is examined against user profiles represented by XPath expressions to identify the
set of interested users. This is known as XML document filtering. The XML document is
then sent to users whose profiles matched it, which is called document dissemination.

One of the key challenges in building such a system is to effectively organize a large
number of profiles in order to minimize the filtering cost and achieve good scalability. In a
joint effort with research scholars from Seoul National University (Korea), we have developed a novel filtering system called **FiST** (*F*iltering by *S*equencing *T*wigs) that performs *holistic matching* of twig patterns with each incoming XML document. Matching is done *holistically* in the sense that FiST does not break a twig pattern into multiple root-to-leaf paths but rather matches the twig pattern as a whole unit.

FiST [42] adopts the idea of encoding XML documents and user profiles into Prüfer sequences. The Prüfer’s method provides a one-to-one correspondence between labeled trees and sequences [57]. It has been shown in the PRIX system [60, 63] that the tree-to-sequence encoding supports efficient twig pattern matching. A collection of sequences from user profiles are organized into a dynamic hash-based index, so that XML documents are filtered against the user profiles efficiently during the two basic phases: subsequence matching followed by refinement. In the subsequence matching phase, a superset of user profiles are identified that potentially match an incoming document. In the refinement phase, false matches are discarded by performing post-processing for branch nodes in the twig patterns.

To further improve the efficiency and scalability, FiST identifies user profiles with similar interests and optimizes the filtering process by enabling shared processing of user profiles with common patterns. As a result, both memory requirement and filtering time of FiST can be reduced. Our extensive experimental study shows that the holistic matching approach enables FiST to outperform the state-of-the-art YFilter system [80] by achieving superior scalability particularly when user profiles contain complex XPath expressions and XML documents are heavily recursive and deep.

### 3.9.2 SketchTree: Approximate Twig Pattern Counts over Streaming Labeled Trees

In recent years, the area of data stream processing has received much attention with key focus on developing online algorithms using a limited amount of memory. The algorithms are single-pass in nature in that every stream element is examined only once. Internet service providers, e-commerce companies and applications such as network monitoring and
sensor data collection, constantly gather and analyze a large amount of data to detect trends and/or anomalies in their systems. The volume of data generated by these applications obviates any traditional indexing and storing techniques. As a result, such applications necessitate efficient algorithms that can provide statistics or summaries on the data using a limited amount of memory.

Recent research in data streaming has focused on developing approximation algorithms with strong guarantees on the error bound. A popular approach has been to compute online synopsis on data streams in a limited space and use the synopsis for approximate query processing. Some of the key challenges that arise in the streaming environment are (a) to develop a synopsis data structure that requires space \( \log \text{or poly-logarithmic} \) in the length of the stream and (b) to compute the synopsis in a single pass over the stream by incurring a small per-element processing cost. Several theoretical and experimental studies have been conducted such as online computation of frequency moments [4], join size estimation [3, 23], online quantile computation [28, 30], and tracking frequent elements [18, 47].

The utility of tree structures spans across many areas such as modeling XML documents, representing phylogenies in biological applications, networks, web log analysis and so on. Today, the extensible markup language XML is a popular standard for information representation and exchange on the Internet [10]. Many emerging applications such as personalized news, stock quotes, and price alerts have become popular over the Internet. The rich data and query semantics provided by XML has triggered several research attempts to build selective information dissemination systems [5, 42], content-based routing systems [21, 68] and XQuery processors [40, 44] for streaming XML data. There is a growing interest in developing software systems for efficiently processing XML streams.

While finding all occurrences of a query pattern in tree structured data such as XML documents is one of the core operations on stored data (e.g., XISS [43], TwigStack [11], TSGeneric\(^+\) [37], PRIX [60]), it may not always be necessary to do so for the purpose of analyzing trends in the online activities. Rather it may be desired to count all match-
I have developed a new algorithm called SketchTree [61] for approximately counting all matching occurrences of a tree pattern in a stream of labeled trees. Counting is a fundamental operation for tasks such as selectivity estimation and data mining. This work attempts to address the problem of counting tree pattern matches over streaming labeled trees such as XML documents using a limited amount of memory. Informally, the tree pattern counting problem is stated as follows.

*Given a stream of labeled trees that are looked at only once in the fixed order in which they arrive, count all matching occurrences of a tree pattern in the stream so far.*

As shown in Figure 3.17, XML documents are processed in an online fashion and a synopsis is maintained in main memory. At any time, a COUNT query can be issued, and the system provides an approximate answer.

Note that this problem is fundamentally different from the problem of filtering for selective information dissemination [5, 42], where user profiles are represented as standing XPath queries. In the aforementioned tree pattern counting problem, there exist no standing queries to begin with, and any tree pattern can be thrown as a query at any moment in time during stream processing.

The key idea of SketchTree is to map tree patterns into Prüfer sequences and even-
tually map these sequences into one-dimensional integers. Prüfer sequence representation is adopted for both the data trees and query tree patterns. When a new data tree arrives in the stream, SketchTree enumerates all the tree patterns in this tree with one to \( k \) edges, which is a tunable parameter. For each tree pattern generated from the tree, the extended LPS and NPS for the pattern are constructed. Each pair of LPS and NPS is hashed into a one-dimensional integer. Thus a stream of trees is mapped into a stream of one-dimensional integer values. As a result, the problem of estimating tree pattern counts is reduced to that of approximately estimating the frequency of one-dimensional points in a stream. The stream of one-dimensional points is sketched using AMS sketches [4]. The use of AMS sketches enables SketchTree to provide probabilistic guarantees on the accuracy of the estimates. Given a query twig, it is mapped to Prüfer sequences and then to a one-dimensional point. Using the AMS sketches, the frequency of the query twig is estimated.

In my work, I show that SketchTree can estimate counts for a class of queries that includes both ordered and unordered tree patterns. Two strategies have been developed to reduce the memory requirement of SketchTree to guarantee a certain level of accuracy. In addition, an algorithm to efficiently enumerate all the tree patterns in a tree with at most \( k \) edges each has been proposed. The empirical evaluation of SketchTree shows that tree pattern counts can be estimated with relative errors of 10-15% using a limited amount of memory. SketchTree can be useful for tasks such as selectivity estimation over stored data, especially when the data is very large and multiple passes over the data is impractically expensive.

3.10 Conclusions

In this chapter, I have presented a new paradigm for XML pattern matching. XML documents are transformed into Prüfer sequences. Similarly, a query twig is also transformed into its Prüfer sequence. To find all occurrences of a query twig, subsequence matching
is performed on the set of sequences followed by a series of refinement phases. I have presented the theoretical background to show the correctness of my approach. Unlike most state-of-the-art techniques, my approach processes twig queries without breaking them into root-to-leaf paths and processing them individually. I have provided empirical results to demonstrate the efficient processing of twig queries by PRIX. The concept of sequencing XML documents and twig patterns has been successfully applied to the problem of XML document filtering and aggregating streaming XML. In conclusion, I have shown that a sequence representation of trees yields efficient solutions to handle large scale tree-structured data.
CHAPTER 4

THE \textit{psiX} SYSTEM

In this chapter, I present the design, implementation, and empirical evaluation of the \textit{psiX} system for indexing and querying XML documents stored in a peer-to-peer (P2P) system. I begin with an overview of \textit{psiX}. I propose a novel signature scheme to map XML documents and twig queries. I describe how a twig pattern match can be detected by using the signature representation of a document and a twig query. I present an indexing scheme for document signatures and how it can be maintained in a distributed way in a P2P environment. The correctness of these techniques is also provided. I describe the design of the core components of \textit{psiX} and some extensions. Finally, I present the empirical evaluation of \textit{psiX} to demonstrate its performance benefits.

4.1 System Overview

A high-level system design of \textit{psiX} and the description of its core components is shown in Figure 4.1. As shown in Figure 4.1(a), a typical peer runs an XQuery processor to pose XQuery queries and requests \textit{psiX} to locate all relevant XML documents and their publishers by best-effort, so that the rest of query processing can be done locally by the peer or can be shipped to a publisher. On each peer, \textit{psiX} runs atop the Chord system [19, 69], which is among distributed hashing frameworks commonly used for P2P networks.\footnote{DHT frameworks such as Pastry [65] can also be used.} Hence, \textit{psiX} inherits the scalability, robustness and load-balancing capabilities of Chord.

The core components of \textit{psiX} are shown in Figure 4.1(b). XML documents that are to be published are first mapped into algebraic signatures by the \textit{Signature Generator} component. The \textit{Publisher} component then inserts the document signatures into a distributed and
hierarchical signature index via the primitives provided by Chord.

When an XQuery query is presented by a peer, psiX performs a best-effort search to locate relevant documents and their publishers as described in the problem statement. During query processing, the twig pattern in an XQuery query is also mapped to an algebraic signature. The Document Locator component traverses the hierarchical signature index by using the primitives provided by Chord. The query signature is used to direct the search process. Once the identifiers of the documents and their publishers are identified, this information is passed back to the XQuery processor. The peer running the XQuery processor can retrieve the documents from their publishers and perform regular XML query processing or can ship the query to relevant publishers.

4.2 Signature Generation

In this section, I present a new method that summarizes XML documents and twig queries into algebraic signatures. I consider only the element tags and their structural relationship for signature generation. Subsequently, I propose a straightforward method to handle values in XML documents and queries.
4.2.1 Polynomials over Galois Field

Under the \( \psi X \) system, signatures of both documents and queries are constructed based on irreducible polynomials over a finite field \([6]\), so that algebraic operations can be performed on the signatures for indexing and query processing purposes. Similar to prime numbers in a set of integers, irreducible polynomials over a finite field are polynomials that cannot be represented by a product of two or more non-trivial polynomials of degree greater than zero. A finite field or Galois Field of modulo 2 (denoted by \( \text{GF}(2) \)) has two elements 0 and 1. In this field, the addition and multiplication operations are equivalent to bitwise XOR and AND operations, respectively. Namely, \( 0 + 0 = 1 + 1 = 0 \), \( 1 + 0 = 0 + 1 = 1 \), \( 0 \times 0 = 1 \times 0 = 0 \times 1 = 0 \), and \( 1 \times 1 = 1 \). An irreducible polynomial in \( \text{GF}(2) \) can be represented by a fixed-length bit string, each bit of which represents the coefficient of a polynomial term. Generally, a polynomial of degree \( n \) over a \( \text{GF}(2) \) is given by \( a_0 x^n + a_1 x^{n-1} + \ldots + a_n \), where each \( a_i \) is either 0 or 1. This polynomial can be represented by a string of \( \left\lceil \frac{n+1}{8} \right\rceil \) bytes.

Polynomials in a finite field have been well studied in the number theory discipline. Irreducible polynomials in a finite field have properties similar to those of integers. In particular, such operations as addition, multiplication, least common multiple (LCM) and greatest common divisors (GCD) can be applied to irreducible polynomials \([6]\). For example, addition of two polynomials in \( \text{GF}(2) \) is equivalent to performing a bitwise XOR of their corresponding bit string representations. For a given product of irreducible polynomials, the presence of any particular irreducible polynomial in the product can be tested by a polynomial division. This is because irreducible polynomials behave like prime factors. These convenient properties form the basis of my indexing and query processing strategy. For a given XML document and a given XQuery query of a twig pattern, the occurrence of the query pattern in the document can be determined by dividing the signature of the document by the signature of the query.
4.2.2 Benefits of Irreducible Polynomials

A polynomial signature is essentially a product of irreducible polynomials. One would argue truthfully that prime numbers could be used instead of irreducible polynomials to provide the equivalent properties. However, I have chosen irreducible polynomials over prime numbers for the following reasons. Irreducible polynomials can be easily generated [24]. Polynomials in GF(2) that are a product of irreducible polynomials can be represented and manipulated fairly easily by binary strings and binary operations (e.g., XOR). Long multiplication and division on polynomials of an arbitrarily high degree can be performed fairly easily. When prime numbers are multiplied together, there is no efficient algorithm for factorization of large numbers. However, factorization of polynomials in finite fields can be done efficiently using randomization [6]. I shall later show that factorization can be avoided by choosing irreducible polynomials of the same degree. I argue that representing a single polynomial is space efficient as compared to storing the irreducible polynomials as a list. (See Section 4.2.6.)

4.2.3 Signatures for XML Documents

To capture the structural summary of a document in a polynomial signature, I first define the structural summary graph (SSG) of a document as follows.

**Definition 4.2.1** (Structural Summary Graph). Given an XML document $T$, its structural summary graph $SSG(T)$ is a directed graph $G = (V, E)$, where the vertex set $V$ denotes a set of distinct tags in $T$, and a directed edge from $v_1$ to $v_2$ exists in $G$ if and only if the tag corresponding to $v_1$ is a parent of the tag corresponding to $v_2$ in $T$. The vertex corresponding to the root of $T$ has a dummy incoming edge.

Note that an SSG can be constructed for a collection of document trees that have the same root. A dummy incoming edge is added to deal with a special case where a document contains just a single node. For a document that has a schema (or DTD), its SSG can be constructed from the schema (or DTD). For a schema-less document, its SSG can be constructed by parsing the document itself. An SSG generated by parsing a document is a
Figure 4.2. Example XML documents and their structural summary graphs that they conform to

subset of the SSG that would be produced directly from the document’s schema (or DTD).

An SSG can form a tree or a (cyclic or acyclic) graph depending on the characteristics of a document. For illustration, two SSGs $S_1$ and $S_2$ for document trees $T_1$ and $T_2$, respectively, are shown in Figure 4.2.

Next I shall describe how irreducible polynomials are used to map an XML document into a polynomial signature, which is essentially a product of irreducible polynomials. To begin with, I assume that each edge in an SSG is assigned a distinct polynomial. Later, in Section 4.4.1, I propose a method to overcome this.

Given a document tree $T$ and an SSG $S$, I use the notation $\text{sig}(T, S)$ to represent the signature of $T$ with respect to $S$, or $s$ for short whenever obvious. A valid signature is a non-zero bit string that represents a product of irreducible polynomials. If $T$ does not conform to $S$, then $\text{sig}(T, S) = 0$, because the signature of $T$ is invalid. Unless otherwise stated, all signatures that I refer to in subsequent discussions are valid.

The process of mapping an XML document into a polynomial signature is described in Algorithm 6. Assuming that an input document tree $T$ conforms to an SSG $S$, $T$ is traversed in preorder. A partial signature $s$ is initialized to a polynomial assigned to the dummy incoming edge in SSG $S$ (Line 1). If $T$ has only one node (i.e., $|T| = 1$), then the partial signature $s$ becomes its ultimate signature. During the preorder traversal of $T$,
each time an edge $e_T$ (associated with a parent-child pair) appears at a certain level (set by the depth of the parent tag in $T$) for the first time, the partial signature $s$ is updated by multiplying itself with the polynomial assigned to the directed edge $e_S$ in $S$ corresponding to $e_T$ (Lines 3 through 7). (Note that the source (tag) of $e_S$ is identical to the parent tag in $e_T$ and the sink (tag) of $e_S$ is identical to the child tag in $e_T$.) If $e_T$ appears at $k$ different levels in $T$, then the polynomial assigned to $e_S$ is multiplied with the partial signature $k$ times. By using the level information, signatures of XML documents with recursive tag names are made more precise.

**Algorithm 6: Polynomial Signature Generation**

```plaintext
procedure GenerateTreeSignature($T$, $S$) 
/* $T$ - XML document tree; $S$ - SSG of $T$ */
polynomial $s$ <- $p_{dum}$; /* given to the dummy edge in $S$ */
1: if $T$ has one node then return $s$; /* signature */
2: /* Traverse the document in preorder. */
3: for each edge $e_T$ in $T$ do
4:   let $l$ denote the level at which the parent tag in $e_T$ appears in $T$
5:   if edge $e_T$ appears at level $l$ for the first time then
6:     let $p$ be the polynomial assigned to the directed edge $e_S$ in $S$ corresponding to $e_T$
7:     $s$ <- $s \times p$
8:   endif
9: endfor
10: return $s$; /* signature of the document */
```

**Example 4.2.2.** We shall construct signatures for the two XML document trees shown in Figure 4.2(a) and 4.2(c). Edges of $S_1$ are assigned distinct polynomials $p_0$ to $p_4$, and edges of $S_2$ are assigned distinct polynomials $r_0$ to $r_6$. By applying Algorithm 6, we compute $\text{sig}(T_1, S_1) = p_0p_1p_2p_3p_4$, and $\text{sig}(T_2, S_2) = r_0r_1r_3^2r_5r_6$. Identical edges appearing at different levels from the root are accounted for by my algorithm. Hence $r_3^2$ appears in $\text{sig}(T_2, S_2)$. Note that $\text{sig}(T_1, S_2) = \text{sig}(T_2, S_1) = 0$ since $T_1$ does not conform to $S_2$, and $T_2$ does not conform to $S_1$.

Our signature scheme is flexible because the precision of document signatures can be improved by making the structural summaries more accurate. For instance, consider a document $D_1$ with a root $A$, and two siblings $B$, and each $B$ has a child $C$ and $D$ respectively.
Suppose another document $D_2$ with a root $A$, has a child $B$, which in turn has two children $C$ and $D$. Using Algorithm 6, $\text{sig}(D_1, S_1) = \text{sig}(D_2, S_1) = p_1 p_3 p_4$. However, if a polynomial $p_5$ is multiplied to the signature when $C$ and $D$ appear as siblings of $B$, then $\text{sig}(D_2, S_1)$ becomes more precise.

4.2.4 Signatures for Twig Queries

Recall that it is of our interest to find documents with structural matches that can be formulated by XPath expressions given in XQuery queries. Since an XPath expression can be mapped to one or more twig patterns, I directly deal with twig pattern queries. Essentially, a twig pattern has a defined structure on the nodes in the pattern and each node has a label associated with it. A twig pattern can be mapped to an equivalent XPath expression.

Twig Queries with Parent-Child Edges I shall first deal with twig patterns that have only parent-child edges and can occur anywhere in the document. Such a twig pattern can be represented as a labeled tree. Suppose we have a twig pattern $Q$ that is a labeled tree with more than one node (i.e., $|Q| > 1$). Assuming that $Q$ conforms to an SSG $S$, we can generate $\text{sig}(Q, S)$ by applying Algorithm 6 but with Line 1 in the algorithm replaced by the statement $s \leftarrow 1$. On the other hand, if a twig pattern $Q$ consists of only a single node (i.e., $|Q| = 1$), this query needs to be handled differently, because there is no edge involved in the query. The signature of a singleton query $Q$ is a list of irreducible polynomials assigned to the incoming SSG edges for the node in $Q$. If $Q$ does not conform with $S$, then $\text{sig}(Q, S) = 0$.

Example 4.2.3. Let us compute the signatures for the twig patterns shown in Figure 4.3(a) and 4.3(b) using the SSGs $S_1$ and $S_2$ in Figure 4.2. By applying Algorithm 6 to $Q_a$ and $Q_b$, we obtain $\text{sig}(Q_a, S_1) = p_1 p_2 p_3 p_4$, and $\text{sig}(Q_b, S_2) = r_3 r_4$. $Q_c$ has one node and its signature is a list i.e., $\text{sig}(Q_c, S_2) = \{ r_1, r_6 \}$.

Handling Twig Queries with ‘//’ and ‘*’ When $Q$ has ‘//’ and ‘*’, the query signature is a list of signatures. The basic idea to construct signatures for such a query is to resolve ‘//’ and ‘*’ by examining the SSG, but not the data. Suppose we need to compute the signature
of $A/B//C/D$. The partial signature for $A/B$ is computed. Then in the SSG, we determine each incoming edge to $C$ via which $C$ is reachable from $B$.\footnote{Depth-first search can be used to test reachability.} The polynomial assigned to such a qualifying incoming edge is multiplied with the partial signature of $A/B$ resulting in a list of signatures. Each signature in the list is then multiplied with the polynomial assigned to the edge from $C$ to $D$ in the SSG. To handle a query $A/B/*/C/D$, we only consider those qualifying incoming edges to $C$ that belong to (directed) paths of length 2 from $B$ to $C$. I show in Section 4.3 that my approach of considering only the incoming edges is enough instead of considering entire paths when resolving ‘//’ and ‘*’.

**Example 4.2.4.** Consider twig $Q_4$ in Figure 4.3(d). The partial signature of $M/O$ is $r_3$. To resolve ‘//’, the only incoming edge reachable from $O$ to $M$ is edge $QM$. Thus we have a list of signatures $\{r_3r_6\}$. Finally, we multiply each signature in this list with $r_3$. Thus, $\text{sig}(Q_4, S_2) = \{r_3^2r_6\}$.

**4.2.5 Time Complexity of Signature Construction**

In Algorithm 6, each tree edge is examined once and tested if it has been considered already or not. Thus for a tree of size $n$, it takes $O(n\log(n))$ time to compute all the irreducible polynomials to be multiplied together. The cost of multiplication depends on the degree of the irreducible polynomials and their total number. In Section 4.5, the cost of signature construction is measured for both documents and twig queries. It is seen that the average cost per signature is within a few milliseconds.
4.2.6 Representation of Signatures

A signature of an XML document or a twig query is essentially a product of irreducible polynomials. Three primitive algebraic operations and one auxiliary operation are performed over signatures: (a) division of two signatures, (b) LCM of two signatures, (c) GCD of two signatures, and (d) total number of irreducible polynomial factors in a signature. One could argue that a list of irreducible factors can be maintained instead of multiplying them to handle the above operations. However, it is space efficient to maintain them as a product. Let $d$ denote the degree of each irreducible polynomial. A signature with $r$ irreducible polynomials requires $\left\lfloor \frac{d \times r + 1}{8} \right\rfloor$ bytes. A list representation would require $\left\lfloor \frac{d + 1}{8} \right\rfloor \times r$ bytes, which is higher. For example, a signature with 200 irreducible polynomials, each of degree 17, requires 426 bytes. A list representation of the irreducible polynomials requires 600 bytes. (Note that a signature can have irreducible polynomials of different degrees starting from degree 1.) In terms of processing time, it is easier to compute LCM and GCD by using a list representation. LCM and GCD computations are done while publishing documents. I consider typical P2P workloads where the frequency of issuing queries is significantly higher than that of document publications. Also, I provide the computation costs of these operations in the experimental evaluation.

Suppose a signature is represented as a list. Since multiplication is avoided, a set of integers can be used to label the edges in the SSG. However, as described earlier, the use of fixed length words to represent the edge labels is not compact in terms of space when the signature is represented as a list. I argue that representing a signature as a product of labels is space efficient. Therefore, irreducible polynomials are used as the basis of the signature scheme.

4.3 Finding a Twig Match

Our goal is to determine the existence of a twig pattern in a document by examining the query and document signatures. Note that we are not interested in finding all occurrences
of a twig pattern in a document.

4.3.1 Divisibility Property of Signatures

I begin with a theorem that states a necessary condition for a twig pattern to have a match in an XML document by examining their corresponding signatures. First I consider the case when a twig pattern has more than one node, can appear anywhere in the document, and has only parent-child edges.

**Theorem 4.3.1.** Given an XML document tree $T$, and a twig query tree $Q$ ($|Q| > 1$), and suppose $T$ and $Q$ both conform to an SSG $S$. If $Q$ is a subgraph of $T$, then $\text{sig}(Q, S)$ divides $\text{sig}(T, S)$.

**Proof.** It is given that $Q$ has at least one edge. When an edge $e$ appears at level $l$ in $Q$, it matches only one directed edge in $S$ (Definition 4.2.1). Let us call it $e_S$. The irreducible polynomial assigned to $e_S$ is multiplied with the partial signature while computing $\text{sig}(Q, S)$, if it was not already seen in $Q$. Since $Q$ is a subgraph of $T$, the edge $e$ appears (at some level) in $T$ and matches the same directed edge $e_S$ in $S$. If it seen for the first time at that level, then the irreducible polynomial assigned to $e_S$ is multiplied with the partial signature while computing $\text{sig}(T, S)$. Therefore, every irreducible polynomial included for computing $\text{sig}(Q, S)$ is also included for computing $\text{sig}(T, S)$. Hence $\text{sig}(Q, S)$ divides $\text{sig}(T, S)$. \qed

As mentioned in Section 4.2.4, if $Q$ has exactly one node or contains ‘/’ or ‘*’, then $\text{sig}(Q, S)$ is a list of polynomial signatures.

**Theorem 4.3.2.** Given $\text{sig}(Q, S) = \{s_1, s_2, ..., s_n\}$, if $Q$ has a match in an XML document tree $T$, then for $(1 \leq i \leq n)$, $\exists i$, s.t. $s_i$ divides $\text{sig}(T, S)$.

**Proof.** Suppose $Q$ has a single node, then $\text{sig}(Q, S)$ is a list of signatures that denote the polynomials assigned to the incoming edges of the node in $Q$ in $S$. If the node in $Q$ appears in $T$ that conforms with $S$, then at least one polynomial assigned to the incoming edges of the node in $S$, appears in $\text{sig}(T, S)$. Hence, at least one signature in $\text{sig}(Q, S)$ divides $\text{sig}(T, S)$.
Suppose $Q$ has ‘//’ axis (or wildcard ‘*’). Then $\text{sig}(Q, S)$ is a list of signatures. Suppose ‘//’ is resolved with paths that are parent-child edges using the SSG $S$. Then $Q$’s signature contains the signatures of all such resolved tree patterns. Let us call them $\{Q_{r_1}, Q_{r_2}, \ldots\}$. If $Q$ has a match in $T$, then at least one resolved tree pattern $Q_{r_i}$ is a subgraph of $T$. By Theorem 4.3.1, $\text{sig}(Q_{r_i}, S)$ divides $\text{sig}(T, S)$. However, I avoid this method of resolving ‘//’ (or ‘*’) to have a finite list of signatures. Based on the approach mentioned in Section 4.2.4, while constructing $\text{sig}(Q, S)$, I consider only the polynomials assigned to the incoming edges of a sink node of ‘//’ (or ‘*’) via which it is reachable from the source node of ‘//’ (or ‘*’) in $S$. The intuition is that when a source and sink of ‘//’ appear in the document as an ancestor-descendant pair, then at least one incoming edge to the sink from the source in $S$ appears in the document. For every resolved tree pattern $Q_{r_i}$, there exists a signature $s_i \in \text{sig}(Q, S)$ that divides $\text{sig}(Q_{r_i}, S)$ since the edges considered during the construction of $s_i$ is a subset of the edges in $Q_{r_i}$. Since at least one resolved tree pattern $Q_{r_i}$ is a subgraph of $T$, at least one signature $s_i \in \text{sig}(Q, S)$ that divides $\text{sig}(T, S)$. 

In order to compute a signature for a query with ‘//’ (or ‘*’) such as $A//D$, it is sufficient to consider only the incoming edges to $D$ reachable from $A$ in the SSG. This is because if $D$ is a descendant of $A$ in a document, then the polynomial assigned to at least one of the incoming edges to $D$ (reachable from $A$) in the SSG will be present in the document signature. Thus, I do not consider all the distinct (minimal) paths from $A$ to reach $D$. Hence, testing for the existence of these polynomials (by division) is enough if $A//D$ appears in the document.

Since Theorems 4.3.1 and 4.3.2 is a necessary condition for a twig match, false matches may occur, but there are no false dismissals. I evaluated the precision of the polynomial signatures, and observed that my scheme yielded very high precision in all cases. (See Section 4.5.) Eliminating falsely matched documents can be performed by post-processing. In a P2P environment, this post-processing can be done by multiple peers storing XML documents or a peer where the query was issued.
4.3.2 Indexing the Document Signatures

Rather than examining every available document’s signature, the search space of signatures can be pruned effectively by organizing the signatures in a hierarchical index and by performing the divisibility test while the index is accessed. I first present the key concepts in a stand-alone environment. The signature index of $\psi X$ can be best explained by drawing an analogy with an R-tree index [33]. The key feature of an R-tree index is the containment property, which warrants that the minimum bounding rectangle (MBR) of a node minimally encloses the MBR’s of all its child nodes. What the containment property is to the R-tree index is the divisibility property to the signature index of $\psi X$. In a signature index, the polynomial signature of a node is divided by the polynomial signatures of all its child nodes. Thus, the polynomial signature of a node is computed by taking the least common multiple (LCM) of the polynomial signatures of all its child nodes.

In the signature index, each node contains entries of the form $(\text{sig}, \text{ptr})$. In a non-leaf node, $\text{ptr}$ is a pointer to a child node in the index and $\text{sig}$ is the LCM of all the signatures in the child nodes. In a leaf node, $\text{ptr}$ denotes a docid, and $\text{sig}$ is a document signature.

**Remark 4.3.3.** Let $s_{\text{parent}}$ and $s_{\text{child}}$ denote the signatures of a node and its child. If a query signature $s_q$ divides $s_{\text{child}}$, then $s_q$ divides $s_{\text{parent}}$ by the divisibility property.

Remark 4.3.3 provides strong leverage to the pruning strategy for query processing, because a query signature $s_q$ not dividing the signature $s_{\text{parent}}$ of a node implies that no signatures of its child nodes are divided by $s_q$.

**Finding Relevant Documents** Given a query signature $s_q$, the goal is essentially to find all document signatures in the index that are divisible by $s_q$. The steps involved are analogous to those for searching a key in an R-tree.

Based on Remark 4.3.3, if $s_q$ does not divide the signature stored in a node entry, then none of the document signatures stored in the leaf nodes that are reachable via the entry’s $\text{ptr}$ are divisible by $s_q$. This property allows us to prune the search space of document signatures. Just as the MBR’s stored in an R-tree node can overlap each other, the polynomial
signatures stored in a node of the signature index can “overlap” if they contain common irreducible polynomials. Thus the search process may traverse along multiple paths in the index to find all the relevant document signatures.

The search begins by fetching the root node of the signature index. For each entry in the node, if its signature is divided by a query signature $s_q$, then the child node of the entry is fetched and the divisibility test is performed again for the signatures stored in this child. This process continues recursively until a leaf node is reached. For each $(\text{docsign, docid})$ pair stored in the leaf node, if $s_q$ divides $\text{docsign}$, then $\text{docid}$ is returned as a candidate.

Inserting a Document Signature A document signature is inserted into a signature index by traversing the index from the root. At each non-leaf node, an entry whose signature has the highest similarity with the input document signature is chosen as the best child. Consequently, similar document signatures are placed close to each other in the signature index, and the corresponding documents are expected to contain common twig patterns.

The similarity of signatures is measured by the $PSim$ function defined in Section 4.3.2. In a P2P environment, this clustering property plays a vital role, since similar document signatures end up being stored across a few peers and thereby minimizing the time required to locate relevant XML documents.

Measuring Similarity between Signatures Given a pair of document signatures, we can estimate the similarity between the two corresponding documents using the algebraic properties of the signatures. The intuition is that XML documents with a similar structure and tags would share common edges from their SSG, and their signatures would share common irreducible polynomials assigned to the edges in the SSG. Therefore, counting the number of irreducible polynomials common to a pair of signatures provides a sense of similarity between their document structures. Below, I define a similarity measure $PSim$ for a pair of signatures.

**Definition 4.3.4** ($PSim$). **Given polynomial signatures** $s_a$ and $s_b$, **let** $g$ and $l$ **denote the number of irreducible polynomials in** $GCD(s_a, s_b)$ and $LCM(s_a, s_b)$, **respectively**. $PSim(s_a, s_b)$
is defined as the ratio $\frac{g}{l}$.

**Example 4.3.5.** Let $p_0$, $p_1$, and $p_2$ be irreducible polynomials. Suppose $s_a = p_0p_1^2p_2^3$ and $s_b = p_1p_2^2$. Then, $GCD(s_a, s_b)$ has 3 irreducible polynomials (i.e., $p_1p_2^2$), and $LCM(s_a, s_b)$ has 6 irreducible polynomials (i.e., $p_0p_1^2p_2^3$). Thus, $PSim(s_a, s_b) = 0.5$.

Based on Definition 4.3.4, a higher value of $PSim$ denotes a higher similarity between two signatures. Note that $PSim$ resembles Jaccard index for measuring similarity of sets. A signature can be mapped into a set by treating identical irreducible polynomials as different by numbering them with a subscript from 1 to $k$ if an irreducible polynomial has a power $k$ in the signature. Then the $LCM$ between the two signatures is denoted by the product of the irreducible polynomials in the union of these two sets. And the $GCD$ between the two signatures is denoted by the product of the irreducible polynomials in the intersection of these two sets.

**Remark 4.3.6.** Given two signatures $s_a$ and $s_b$, and if all the irreducible polynomials of $s_a$ and $s_b$ have the same degree, then $PSim(s_a, s_b)$ is computed by $\frac{\deg(GCD(s_a, s_b))}{\deg(LCM(s_a, s_b))}$.

This remark provides a significant opportunity of optimization for determining document similarity. That is, if all the irreducible polynomials assigned to edges in an SSG are of the same degree, then $PSim$ can be computed, without incurring a potentially costly factorization step, by computing the polynomial degrees.

Such an assignment of polynomials is feasible because the number of irreducible polynomials of degree $n$, denoted as $C(n)$, is about $\frac{1}{n}$th the number of all polynomials of degree $n$ [6]. For example, $C(17) = 7710$, which is enough for an SSG with a modest number of distinct edges.

4.4 Coping with a P2P Environment

In this section, I present the main design principles of $psiX$ to cope with a peer-to-peer environment. The current implementation of $psiX$ is built on top of the Chord system [69], which is one of the popular distributed hash table (DHT) frameworks, and hence inherits
its benefits (e.g., scalability, load balancing, and robustness). I frequently use the terms a node of an index and a peer.

4.4.1 Distributed Signature Index

The signature index is stored using a distributed hash table maintained by Chord. Each Chord peer, and key is mapped to a 160-bit identifier. The identifier space forms a ring called the Chord identifier ring. Chord supports lookup operations to retrieve the value of a key and insert operations to store key-value pairs. Chord allows multiple inserts for the same key and appends the new value to the existing values. In psiX, the identifier of an index node is treated as a key, and the entire content of the index node as a value. This key-value pair is stored in the DHT. The signature index is then maintained and traversed by insert and lookup operations.

**Example 4.4.1.** A signature index is shown in Figure 4.4(a). Each node in the index is assigned a unique identifier. The root node has id “1.1,” and its first, second and third child nodes have ids “2.1,” “2.2,” and “2.3” respectively. The ptr field of each entry in a non-leaf node is a node id of a child node. Each index node is then stored in Chord as a key-value pair with its node id as the key and the entire node content as the value. Figure 4.4(b) shows the Chord identifier ring, where the dark dots denote the 160-bit identifiers of the peers, and the light dots denote the 160-bit identifiers of the signature index node ids. According to the Chord protocol, a key is stored on the first peer that has an identifier equal to or follows the identifier of the key in the Chord ring. In Figure 4.4(b), the nodes
with id “1.1” and “2.3” are stored on peer A, the node with id “2.1” is stored on peer C, and the node with id “2.2” is stored on peer B. The signature index can be traversed by issuing a lookup operation for node id “1.1”. A child node’s id can be obtained by reading the ptr field of an entry in the root, and another lookup can be issued.

Signature Generation In order to assign irreducible polynomials to the edges in an SSG, each peer uses the same algorithm \(^3\) to generate all the irreducible polynomials of degree \(n\), whose value may have been chosen by the bootstrap (first) peer or is publically known for that P2P environment. Then, given an SSG \(S\), it is traversed in depth-first order. For each encountered edge \(e\) in \(S\), a path from the dummy tag up to the sink tag of \(e\) is constructed. This path is hashed to an offset \(i \in [1, C(n)]\) by using, for example, a combination of Rabin’s fingerprinting method [58] and the universal hash functions [12]. Recall \(C(n)\) is the total number of irreducible polynomials of degree \(n\) in GF(2). (See Section 4.3.2.) The polynomial at offset \(i\) in the list of irreducible polynomials is then assigned to edge \(e\). The signature can be computed using Algorithm 6.

An SSG can be constructed from a DTD or by parsing the document once if it is schema-less. If a DTD is public (e.g., dblp.dtd), then a peer can construct signatures for XML documents that conform to the DTD or for twig queries on such XML documents. However, if the DTD or the SSG of the data is not public, then it should be advertised so that other peers can issue queries to locate such data. The incentive for doing so is that other peers can now search its data. One way of achieving this is to let peers publish such information (much smaller than the actual data) as values of well-known keys in the DHT framework. A peer can occasionally lookup the well-known key and learn about the DTDs/SSGs of other peers. Thus, given a query, a peer constructs signatures based on the DTDs/SSGs that are relevant to the query. (The signature for the query can be a list.) For example, a query `//author/name` may be relevant to `dblp.dtd`, `books.dtd`, and an SSG of a schema-less document. I think that the scenario when a peer poses a query

\(^3\)In the current implementation of psiX, we use an irreducible polynomial generator obtained from the Combinatorial Object Server [24].
with absolutely no knowledge about the data (i.e., its DTD or SSG) may not be a common case.

Assigning Unique Identifiers to Nodes It is essential that the index nodes are assigned unique node ids. One way of dynamically achieving this in a P2P environment is by using simple Farey fractions \[16\]. Suppose a node is assigned an interval \((\frac{n_l}{d_l}, \frac{n_r}{d_r})\) such that \(\frac{n_l}{d_l} < \frac{n_r}{d_r}\), where \(n_l\) and \(n_r\) are non-negative integers, and \(d_l\) and \(d_r\) are positive integers. Let \(l\) be its level. In my design, each index node has a header that stores its level and assigned interval. Then the node’s id is computed by Chord’s hash function \(h("l, n_l, d_l")\) that maps a key to a 160-bit identifier with a very low probability of collision. When a node at level \(l\) is split, its interval is split into two non-overlapping intervals \((\frac{n_l}{d_l}, \frac{n_r+n_l}{d_r+d_l})\) and \((\frac{n_r+n_l}{d_r+d_l}, \frac{n_r}{d_r})\) – a nice property of Farey fractions. The first interval is retained by the original node, and the second interval \((\frac{n_r+n_l}{d_r+d_l}, \frac{n_r}{d_r})\) is assigned to the newly created node whose id is computed by \(h("l, n_r + n_l, d_r + d_l")\). At any level, all the node intervals are contained in the interval \((0, \frac{1}{1})\). By design, once a non-root node is created at level \(l\), it remains at level \(l\). Also the left boundary of a node’s interval never changes. In the signature index, the leaf node is at level 0, and the root node has the highest level. The root node is special, and its id is fixed, and publically known. Each time the root splits, its level increases by one, and two child nodes at the previous level are created. The root’s interval is fixed at \((0, \frac{1}{1})\), and its child nodes are assigned intervals \((0, \frac{1}{2})\) and \((\frac{1}{2}, \frac{1}{1})\) respectively, whose node ids are computed by \(h(\cdot)\).

4.4.2 Publisher

A peer publishes an XML document by inserting the document’s signature and identifier into the signature index. Note that a document resides at the publisher’s host. I assume that this document identifier contains the IP address of the peer. When a peer issues a query, it receives a set of document identifiers of relevant documents, and then contacts the publishers for the documents by using the IP addresses. As new documents are published,
the signature index grows, and node splits occur when required. Due to the decentralized nature of a P2P network, the DHT framework does not provide guaranteed consistency for multiple-writers, which is essential when peers concurrently update and split index nodes. To overcome this limitation, we propose an approach that allows peers to synchronize among themselves during node splits, and this approach ensures that the nodes in the index do not go missing due to concurrent operations by peers. Therefore, successfully inserted document signatures are not lost. Our assumption is that a peer does not explicitly delete document signatures.\footnote{Chord does not support explicit deletes.} So the signature index never shrinks – once created, the nodes are never deleted, and they are never merged together. I shall frequently refer to a node with id $k$ as node $k$ in subsequent discussions.

For the ease of exposition, we assume that the DHT operations such as lookup and insert issued to Chord complete successfully, and peer failures are handled by Chord via replication of keys. (See Section 4.4.5.) I also assume that the peers are non-malicious and adhere to the protocols described below for insertion and node split. I consider a typical P2P environment where queries from users are more frequent than document publications.

**Insertion Protocol** In psiX, we provide peers with two new operations called $\text{insert}_U$ and $\text{insert}_S$ on a node id that are variants of the basic $\text{insert}$ operation provided by the DHT. These two operations are invoked by a peer as part of the insertion protocol. Using an $\text{insert}_U$ operation, a peer can update a signature in a node entry to preserve the divisibility property, as well as adding a new $(\text{sig, ptr})$ entry to a node. Using an $\text{insert}_S$ operation, a peer can split a node. The $\text{insert}_S$ operation allows multiple peers to synchronize among themselves when splitting the same full node.

I first consider the basic case when there is enough room in the index to accommodate a document signature $s$ with docid $d$. To insert the pair $(s, d)$, the signature index is traversed from the root by $\text{lookup}(\text{rootid})$. Once a node’s content is fetched, the best child node is selected for insertion. At a non-leaf node, the $(\text{sig, ptr})$ entry that has the highest similarity
with the input document signature, measured by $PSim(s, sig)$, is chosen. (In case of ties, the entry whose signature has the smallest polynomial degree is chosen. Further ties can be broken by randomly choosing one of them.) By issuing $lookup(ptr)$, a child node is fetched from the DHT. When a leaf node is fetched, and it is not full, the following operations are done: $(s, d)$ is inserted into the leaf node, and the signatures in selected entries of the nodes from the parent of the leaf to the root are updated to preserve the divisibility property.

An $insert_U$ operation is used to update the index nodes on the selected path to reflect the insertion of a new signature. Recall that an index node is stored as a key-value pair, where the key is the nodeid, and the value is a list of $(sig, ptr)$ ordered pairs. When an $insert_U$ operation is issued with the arguments $k$ and $(s, p)$, then the peer storing node $k$ will atomically replace the pair $(r, p)$ in the corresponding value list with the pair $(\text{LCM}(r, s), p)$. If such a pair $(r, p)$ does not exist in this value list, then $(s, p)$ is appended to this list.

A peer issues $insert_U$ on the leaf node to add the pair $(s, d)$, and issues $insert_U$ on the non-leaf nodes starting from the parent of the leaf up to the root to update the signatures in the node entries while backtracking. This is analogous to adjusting the MBRs in the internal nodes of an R-tree during insertion. Note that multiple peers can issue an $insert_U$ operation on the same node entry whose signature is $r$. Since LCM is an associative operation, regardless of the order in which their requests arrive at a peer storing the node, eventually the signature in the node entry is the LCM of $r$ and the signatures passed as arguments to $insert_U$ by the peers. Thus as peers insert new signatures, they can be eventually searched by other peers by traversing the index.

**Node Splitting Protocol** If a leaf node chosen for insertion is full, then the node splitting protocol is invoked by a peer. As new nodes are created and logical links to them are updated during splitting, the splitting protocol ensures that index nodes are not lost due to concurrent invocations of node splits by peers. In the subsequent discussions, I emphasize that the signature index remains well-linked under concurrent operations by peers, and state theorems that establish the correctness of this scheme.
Definition 4.4.2 (Well-linked). A signature index is well-linked if the logical links to every index node of the signature index including any newly created node can be found by traversing the signature index from the root.

When a leaf node that is fetched by a peer is full, it determines the set of full nodes (that it has fetched) from the leaf to a non-leaf node that need to be split. The peer issues node splits in the top-down fashion starting from the top-most non-leaf node to the leaf to prepare space before inserting the pair \((s, d)\). A reinsertion of the pair \((s, d)\) is done from the parent of the highest-level node that split. The splitting is done this way so that when multiple peers fetch the same full node, they consider the same set of child nodes while partitioning the node entries. If we proceed in the bottom-up fashion, then while splitting an internal node, two different peers may have different newly created child node entries to consider while partitioning the entries in the full node. Hence, the top-down approach simplifies the splitting algorithm.

First, I shall consider the case when only one peer invokes the splitting protocol on a node. Analogous to splitting a node in an R-tree, a peer (locally) creates a partition of the node entries of a full node by first picking two seed entries, one for each partition, that have the most dissimilar signatures. Then each remaining \((\text{sig}, \text{ptr})\) entry is added to a partition that has the highest similarity between \text{sig} and the LCM of the signatures in the partition. (Ties can be broken by choosing the smaller partition. Further ties can be broken by randomly choosing a partition.)

As described in Section 4.4.1, the level and interval information in the header of a full node is used to determine the level, interval, and id of its new sibling node. A peer invokes an \(\text{insert}_S\) operation as part of the node splitting protocol. The \(\text{insert}_S\) operation is invoked by this (calling) peer on a node id (key), along with its current header value, a new header value for it, and a list of \((\text{sig}, \text{ptr})\) entries. When the peer responsible for storing that node id gets the \(\text{insert}_S\) request, it atomically does the following two steps: (a) If the node id does not exist, it creates the node and initializes the node’s header and content with the new header value and the \((\text{sig}, \text{ptr})\) entries passed by the calling peer,
**Algorithm 7**: Protocol for Splitting a Non-root Node

/* This procedure is called by a peer to split a node */

```plaintext
proc splitNode(k; V)
  /* k - key or node id; V - value list or node contents; */
  partition V into V₁ and V₂

  (k', hₖ', hₖₙₑₓ) ← f(hₖ) /* new node id, and headers */

  let s be the LCM of signatures in V₂

  status ← insertₛ(k', 0, hₖ', V₂)

  if status = first then
    insertₚ(parent(k), (s, k')) /* create link */
    insertₛ(k, hₖ, hₖₙₑₓ, V₂) /* update hdr */
  else
    await(k's header ! = hₖ) /* wait for hdr to change */
  endif
endproc
```

/* This procedure is executed by the peer storing r */

```plaintext
proc insertₛ(r, hₙₑₓ, hₙₑₙاء, Vᵣ)
  /* r - key or node id; hₙₑₓ, hₙₑₙاء - old and new headers; */
  /* Vᵣ - list of entries */

  if r does not exist then
    create a node with id r, header hₙₑₙاء, and contents Vᵣ return first; /* first peer */
  else if r's header = hₙₑₓ then
    remove Vᵣ from the value list of r set hₙₑₙاء as r's header
  endif
  return first; /* not the first peer */
endproc
```

respectively. (b) Otherwise, it checks if the header of the node (that it has) is identical to the current header value passed by the calling peer. If so, it removes the \((\text{sig, ptr})\) entries from the node and updates the header with the new header provided by the calling peer. Note that Step (a) enables the creation of a new sibling node, and Step (b) enables the removal of entries from a node that split. Conceptually similar to splitting a node in an R-tree, a calling peer invokes the \(\text{insertₚ} \) and \(\text{insertₛ} \) operations to complete the process of a node split in the P2P environment. This is illustrated in Figure 4.5. The signature index remains well-linked after the calling peer successfully executes the splitting protocol.
To cope with the lack of concurrency control mechanism in the underlying DHT, the node splitting protocol of psiX (a) exploits the header information of a node, and (b) extends the \textit{insert}_S operation to pass back a handle on a node. (The handle is a Boolean value which is \textit{true} when valid.) As a result, peers can synchronize among themselves, and only one peer performs the critical splitting tasks, while others wait for this peer to finish. Recall that a node’s header is changed via an \textit{insert}_S operation after it splits. In addition, a peer that is responsible for storing a node returns a valid handle to a calling peer whose \textit{insert}_S request arrives first provided that the node was created during this request. A calling peer whose request arrives after the node has already been created is returned an invalid handle.

Let us analyze the case when more than one peer concurrently issue the splitting protocol on the same node \( k \) illustrated in Figure 4.5(a). When a calling peer decides to split a full node \( k \), it invokes \textit{insert}_S on the new node \( k' \) (split from \( k \)) and checks the handle. If the handle is valid, then the calling peer’s request created the node \( k' \) (Figure 4.5(b)), and it proceeds to update the link from the parent of \( k \) to the node \( k' \) using an \textit{insert}_U operation (Figure 4.5(c)). Finally, the node contents of the node \( k \) are updated and its header is changed using an \textit{insert}_S operation (Figure 4.5(d)). If a calling peer receives an invalid handle, it awaits for the header of the full node \( k \) to change. During this process, the peer issues \textit{lookup}(\( k \)) using an exponential backoff strategy [54]. When the header changes, this peer realizes that the splitting process has been completed by some other peer, and successfully finishes splitting node \( k \). Algorithm 7 describes the node splitting protocol. The function \( f \) outputs the id of a new node \( k' \) split from \( k \) along with new headers for them. The procedure \textit{insert}_S returns a handle on the node. When the handle is \textit{first}, then the peer continues to perform the critical splitting operations. Otherwise it waits for the header of the node that split to change.

In the splitting protocol, only one peer performs the critical splitting operations on a node, and the remaining peers wait for completion of the splitting operations. By using a handle on a node, and awaiting the change of the node’s header, peers synchronize among themselves to split a full node, and thus retain the signature index in a \textit{well-linked} state. I
state the following theorem.

**Theorem 4.4.3.** Given a signature index that is well-linked, suppose \( n \) peers independently and concurrently decide to split the same nodes starting from level \( l_1 \) to level \( l_2 \). Then if each peer issues a node split and finishes successfully, then the index remains well-linked.

**Proof.** I show that starting with a well-linked index, the index remains well-linked after a full node split at a level \( l \) by \( n \) peers. Then by using the top-most level split as the base case, we prove by induction that the index remains well-linked after splitting the same set of nodes from a level \( l_1 \) to \( l_2 \) by \( n \) peers. Given a well-linked index, we shall consider three cases while splitting a node at a level \( l \) in the signature index.

**Case (1): All calling peers have a copy of the node \( k \) with \( h_k \) as the header and \( V \) as the node content.**

When Algorithm 7 is applied, each peer creates the same partitions \( V_1 \) and \( V_2 \). Also, they compute the same new node id \( k' \), and headers for \( k \) and \( k' \). The first peer to receive a valid handle creates \( k' \) with \( V_2 \), updates the link from the parent of \( k \), and then finally removes entries in \( V_2 \) from node \( k \). It then changes the header of \( k \). The remaining calling peers lookup \( k \) and finish once the header has changed. As a result, the index remains well-linked.

**Case (2): All calling peers have a copy of the node \( k \) with \( h_k \) as the header but different node contents w.r.t. the signatures in the node entries.**

When Algorithm 7 is applied, each peer computes the same new node id \( k' \), and headers to be created, but different partitions of \( V \). However, the logical links in \( k \) are identical. Hence after partitioning \( V \), all the logical links are still present. Since only one peer receives a valid handle, its partitioning of the node entries is adopted during the split, and it completes the critical splitting tasks that leaves the index well-linked. Once the header is changed, all the other calling peers finish successfully.

**Case (3): All calling peers have a copy of the node \( k \) but with different headers.**

When the headers are different, this implies that \( k \) has already split after some calling peers fetched \( k \). When such peers try to invoke a split, then the new node \( k' \) that they attempt to create exists already and they do not receive a valid handle. When they check the header of
Before splitting $k$

Create a sibling $k'$

Update the link to $k'$

Remove entries from $k$

**Figure 4.5.** Steps involved during a node split operation

$k$, it is already different from what they have and they finish successfully. One calling peer that has the latest header of $k$ will perform the splitting tasks and finish successfully. Thus the index remains well-linked.

Now I shall apply induction to prove the theorem.

Let $P(t)$ denote the hypothesis “the signature index remains well-linked after top-down splits at $t$ consecutive levels (by $n$ peers) starting from a non-root node”.

$P(1)$: the signature index remains well-linked after one split of a non-root node (by $n$ peers).

Basis: $P(1)$ is true based on Cases (1), (2), and (3).

Inductive step: Assume $P(t)$ is true. To prove that $P(t + 1)$ is true. Before the index splits the $(t + 1)$'th time, by inductive hypothesis, it remains well-linked after the $t$'th split. Using Cases (1), (2), and (3), it remains well-linked after the $(t + 1)$'th split. Hence $P(t+1)$ is true. By the principle of mathematical induction, the index remains well-linked after $t$ ($\geq 1$) splits of the same set of full nodes starting from a non-leaf node to lower level nodes.

$\Box$
The splitting of a root node is special, and the root id never changes. To split a root shown in Figure 4.6(a), a peer (locally) creates the partition of the entries in the root. The level of the two child nodes is obtained from the root header, and their intervals are assigned based on the scheme described in Section 4.4.1. Let \( c_1 \) and \( c_2 \) denote the two new child ids to be created. The calling peer invokes \( \text{insert}_S \) on child \( c_1 \). If it receives a valid handle, it continues to create child \( c_2 \) (Figure 4.6(b)), removes entries from the root using an \( \text{insert}_S \) operation, creates logical links from the root to \( c_1 \) and \( c_2 \) using an \( \text{insert}_U \) operation, and finally updates the header of the root (Figure 4.6(c)). Note that the level of the root has increased by one. If a calling peer receives an invalid handle for node \( c_1 \), then it awaits for the root’s header to change using an exponential backoff strategy while issuing \( \text{lookup} \) on the root. When the header change for the root is detected, the peer finishes successfully.

Algorithm 8 describes the steps involved in splitting the root node. I state the following

**Algorithm 8: Protocol for Splitting the Root**

```plaintext
proc \( \text{splitRoot}() \)
1: let \( R, h_R, h_{R_{\text{new}}}, V_R \) denote the id, old header, new header and contents of the root fetched by a peer
2: partition \( V_R \) into \( V_{R_1} \) and \( V_{R_2} \)
3: let \( l_1 \) and \( l_2 \) be the LCMs of signatures in \( V_{R_1} \) and \( V_{R_2} \) respectively
4: let \( k_1 \) and \( k_2 \) denote the node ids of the two new child nodes
5: let \( h_{k_1} \) and \( h_{k_2} \) denote the headers of \( k_1 \) and \( k_2 \)
6: \( \text{status} \leftarrow \text{insert}_S(k_1, \emptyset, h_{k_1}, V_{R_1}) \) /* create left child */
7: if \( \text{status} = \text{first} \) then
8: \( \text{insert}_S(k_2, \emptyset, h_{k_2}, V_{R_2}) \) /* create right child */
9: \( \text{insert}_S(R, h_R, h_{R_{\text{new}}}, V_R) \) /* remove contents */
10: \( \text{insert}_U(R, (l_1, k_1)) \) /* link to left child */
11: \( \text{insert}_U(R, (l_2, k_2)) \) /* link to right child */
12: \( \text{insert}_S(R, h_R, h_{R_{\text{new}}}, \emptyset) \) /* update root’s header */
else
13: \( \text{await}(R' \text{'s header} ! = h_R) \) /* wait for a hdr change */
endif
endproc
```

Theorem 4.4.4. The signature index remains well-linked under successful concurrent in-
vocations of a root split by peers.

Proof. Similar to the proof for Theorem 4.4.3, there are a few cases to consider.

Case (1): All calling peers have a copy of the root \( r \) with \( h_r \) as the header and \( V_R \) as the node content. When Algorithm 8 is applied, each peer creates the same partitions \( V_{R_1} \) and \( V_{R_2} \). Also, they compute the same child node ids \( k_1 \) and \( k_2 \), and headers \( h_{k_1} \) and \( h_{k_2} \) for the split. The first peer to receive a valid handle creates \( k_1 \) and \( k_2 \), removes entries from \( r \), updates the links to \( k_1 \) and \( k_2 \) from \( r \), and finally updates the header of \( r \). The remaining calling peers lookup \( r \) and finish once the header has changed. As a result, the index remains well-linked.

Case (2): All calling peers have a copy of the root \( r \) with \( h_r \) as the header but different node contents w.r.t. the signatures in the node entries. When Algorithm 8 is applied, each peer computes the same child node ids \( k_1 \) and \( k_2 \), and headers \( h_{k_1} \) and \( h_{k_2} \), but different partitions of \( V_R \). However, the logical links in \( r \) are identical. Hence after partitioning \( V_R \), all the logical links are still present. Since only one peer receives a valid handle, its partitioning of the node entries is adopted during the split, and it completes the critical splitting tasks that leaves the index well-linked. Once the header is changed, all the other calling peers finish successfully.

Case (3): All calling peers have a copy of the root \( r \) but with different headers. When the headers are different, this implies that \( r \) has already split after some calling peers fetched \( r \). When such peers try to invoke a split, then the new nodes \( k_1 \) and \( k_2 \) that they attempt to create exists already and they do not receive a valid handle. When they check
the header of \( r \), it is already different from what they have and they finish successfully. One calling peer that has the latest header of \( r \) will perform the splitting tasks and finish successfully. Thus the index remains well-linked.

\[ \square \]

**Remark 4.4.5.** Once a document signature is successfully inserted into a leaf node of the signature index, it remains in the index because the index remains well-linked under concurrent operations by peers.

**Extensions** The signature index always remains well-linked, but the divisibility property may not hold in the nodes that were involved in the splitting process. Our extensions aim to restore the divisibility property in a graceful manner. Recall that I consider a typical P2P environment where queries from users are more frequent than document publications.

When a node is split, the signature of its parent entry is not updated by a calling peer. This is because when this peer is in the process of splitting a node, another peer could choose the same entry in the parent to add a signature, and update it. But this calling peer is oblivious to such inserts unless it fetches its parent again. As a result, the signature in the parent may not be tight. To handle this, a peer responsible for storing a non-leaf node \( k \) maintains a timer for it. (A non-leaf node have level greater than zero which is stored in its header.) When this timer expires, it performs a maintenance task atomically: for each entry \((s, p)\) in \( k \), fetch the child \( p \), recompute the LCM of all its signatures, and update the signature \( s \) in the entry. The timer for node \( k \) is initialized when this node (1) is created using \( \text{insert}_S \), or (2) is modified via \( \text{insert}_U \) by the addition of a new node entry (e.g., adding a link from the parent to a new child). The timer interval is doubled each time the timer expires i.e., an exponential backoff strategy. The intuition is that a node that does not satisfy conditions (1) and (2) will rarely perform the maintenance task over time. The timer interval can be stored in the node so that if a new peer becomes responsible for that node, it can continue the maintenance task by initializing the timer with the interval stored in the node.
In another scenario, suppose a peer inserts a signature into a leaf and suppose its parent splits (due to other peers) before it updates the entry chosen for insertion using $\text{insert}_U$. Thus the chosen entry in the parent could have been moved to another node after splitting. In that case, $\text{insert}_U$ on the parent would result in two logical links to the leaf. To avoid this, a calling peer issues a stricter version of $\text{insert}_U$ on the non-leaf nodes when updating the signatures during insertion. Thus when the stricter version of $\text{insert}_S$ is invoked on a node with an argument $(s, p)$, the peer responsible for storing that node atomically updates the signature only if some pair $(r, p)$ already exists in the node. Otherwise, the node is unaffected, and the entry in the new parent chosen for insertion is not updated. Fortunately, the maintenance task on the new parent will update this entry.

Suppose a non-leaf node is fetched by a peer and requires a split as it is full. Suppose others update the signatures in the node entries before this peer issues a split. Then as a result of the splitting protocol, the updates are lost in the newly created sibling. A similar situation arises when two peers fetch a full node with the same header but with different signatures in the entries. A splitting of the node may result in lost updates if the peer with a stale copy receives a valid handle. Fortunately, the aforementioned maintenance task will recover such lost updates. A peer may read a node that is not full, but by the time its $\text{insert}_U$ request reaches the peer that stores the node, the node may become full due to inserts from other peers. Such a situation is handled by the peer storing the node by forcing that $\text{insert}_U$ to fail. The calling peer then initiates a node split.

4.4.3 Document Locator

Given a query signature $s_q$, a peer traverses the signature index starting from the root node using the $\text{lookup}$ operation. The $\langle \text{sig}, \text{ptr} \rangle$ entries are tested for divisibility with $s_q$ to determine the need for further traversal. If the node is a leaf and if $s_q$ divides $\text{sig}$, then $\text{ptr}$ is output as a matching global docid. If the node is a non-leaf node and $s_q$ divides $\text{sig}$, then the child node is fetched by issuing $\text{lookup}(\text{ptr})$ and the divisibility test is repeated as before. If the divisibility test fails, then the child node is not searched further. If a twig
query has ‘/’ or ‘*’, then the process of locating documents is the same as above except that the query signature is a list of signatures and Theorem 4.3.2 is used for the divisibility test.

4.4.4 Characteristics of the Distributed Signature Index

A P2P system is faced with four challenges, namely, (a) data distribution, (b) dynamism, (c) data evolution, and (d) decentralization [26]. The psiX system, via the signature index, addresses these challenges in the following manner.

- Data Distribution - The signature index in psiX is stored in the DHT using the index node ids as the keys. Since the index is balanced and has fixed node fanout, the document signatures are distributed across the peers. As a result, while query processing, the load is distributed across peers.

- Dynamism - When peers join the system, then they become responsible for storing some index nodes based on the DHT protocol. When peers leave the system, the nodes that they stored are moved over to other peers.

- Data Evolution - As more document signatures are indexed, the signature index grows in size and remains balanced. New index nodes are stored in the DHT, and thus data continues to be distributed across peers.

- Decentralization - There is no central site that controls the processing of queries and publishing documents. When locating relevant documents, a peer issues key lookups over the DHT and retrieves the relevant portions of the signature index. There is no notion of super-peers in psiX, and the peers execute the indexing protocols without any central control.
4.4.5 Failure of Peers and Load Balancing

The $psiX$ system relies on the DHT for managing failure of peers. For instance, Chord replicates a key (and its value) in some $r$ immediate successors of the peer that is responsible for storing the key. If this peer fails, then a replica is fetched from one of its $r$ successors. The effectiveness of replication has been well studied and evaluated [69, 19].

Balancing the load among the peers is essential for building an efficient and scalable P2P system. The $psiX$ system relies on Chord’s location protocol for balancing the distribution of keys across the peers. The hierarchical index maintained by $psiX$ has skewed index node access patterns. The top-level nodes are accessed more often than lower-level nodes. This increases the load on the peers that store the top-level nodes of the signature index. Also certain parts of the data can be frequently queried by peers. In this regard, I suggest the two independent caching approaches that can be combined – one that can be incorporated into $psiX$ and the other that has been proposed already for Chord. In typical P2P systems, queries from users are more frequent than publishing documents. In such environments, during query processing, the temporal locality of index node accesses can be exploited by caching. We trade off freshness of results (i.e., newly published relevant documents may not be returned immediately) for query performance.

The first approach is a simple but effective caching strategy employed by each peer while locating documents. Each peer maintains an LRU cache of key-value pairs (i.e., $nodeid$ and node contents). Similar to the buffer management for traditional indexes, the key-value pairs being accessed are pinned in the cache and unpinned when they are no longer required. Only unpinned key-value pairs that are least-recently used are evicted from the cache. As a result, the root node can be kept in the cache most of the time. In Section 4.5, I present the evaluation results of the caching strategy and show that caching significantly reduces the lookup cost during query processing.

The second approach is to employ the techniques proposed in the Cooperative File System (CFS) [19], a P2P read-only storage system built over Chord. CFS caches (LRU
based) file blocks along the lookup path in the Chord ring. Hence a popular block can be found within a few hops along the lookup path without having to contact the peer that is responsible for storing it. Experimental evaluation of CFS [19] has shown that caching is indeed effective in balancing the load on the peers. The psiX system can adopt the aforementioned technique to deal with skewed access patterns of the index nodes.

Over a period of time, due to new documents being published by other peers, the key-value pairs in the cache become inconsistent. A peer can associate expiration times with each key-value pair in the cache. Once a key-value pair expires in the cache, it is removed from the cache. The next lookup for the key will be served by the peer responsible for storing it. In future, I would like to investigate how the techniques proposed in the web-caching community can be adopted by psiX.

4.4.6 Joining and Leaving of Peers

The psiX system relies on the Chord protocol to handle the leaving and joining of peers. The effectiveness of these schemes has already been studied [69]. When a node joins the system, it contacts a bootstrap peer according to the Chord protocol and becomes part of the Chord ring. During this process, the new peer can be informed about the degree of irreducible polynomials to use. Now the peer can start publishing and issuing queries to locate relevant XML documents.

When a peer leaves the P2P system, it does not delete the documents it has published so far. The peer follows the Chord protocol where the key-value pairs it stores are moved to its successor. When a peer rejoins, it can still access the documents it has previously published. I do not evaluate psiX’s performance under different levels of churn i.e., peers continuously joining and leaving. I believe that the various methods proposed to handle churn in DHTs can be directly exploited by psiX.
Alternate Indexing Method for Signatures

The document signatures can also be stored in a P2P environment as a distributed index based on inverted lists. This indexing scheme, which will be called Inverted-index, has been designed following the basic ideas proposed by Galanis et al. [25], but my scheme differs in that signatures of XML documents are stored in the distributed hash table instead of paths in the documents. Our scheme allows holistic processing of twig queries since the entire signature of a twig is tested against a document signature. This scheme is better than a path-based indexing scheme (e.g., [25, 9]) where twigs are processed by breaking them into paths and processing them individually by issuing separate DHT lookups. For example, under the scheme proposed by Galanis et al., a twig query with three paths (branches) requires three separate DHT lookups.

Using the Inverted-index, an XML document tree $T$ (with an SSG $S$ and a global docid $d_G$) is published by issuing an insert $(tag, (sig(T, S), d_G))$ operation to Chord for each unique tag in the XML document. To locate relevant XML documents for a given query $Q$, a peer chooses the target node $t$ of $Q$ and performs lookup($t$). For each signature-docid pair $(s, d)$ returned as a value item for the lookup, if $s_q$ divides $s$, then the docid $d$ is returned as a match. Recall that in Chord, a key lookup requires hops that is logarithmic in the number of peers. Since Inverted-index and psiX use signatures for documents and queries, the set of global docids returned for a twig query by Inverted-index is identical to the set of global docids that would be returned by psiX for the same query. The Inverted-index system has been developed this way in an attempt to provide a setting for fair comparisons in performance evaluation. A local LRU cache can be maintained by each peer running Inverted-index (similar to psiX) so that temporal locality of tag references can be exploited during query processing.

An inherent limitation of Inverted-index is that an insert operation is done for each unique tag in the XML document while psiX requires one insert operation per document. If the same tag appears in XML documents that are unrelated (e.g., generated from differ-
ent DTDs), then the list of signatures stored for that tag will include all such (unrelated) XML documents. Thus the list of signatures for a tag can grow large, thereby reducing the pruning power of Inverted-index. Our idea of organizing documents signatures in a hierarchical index can be incorporated into Inverted-index too. A peer running Inverted-index can build an index over the signatures it stores for each tag. Then the process of locating relevant document signatures for a twig query can be processed by the peer storing the target node rather than the peer that issued the query. The relevant document identifiers can then be returned to the requesting peer. Note that this scheme aims at reducing the computation cost of searching a list of signatures, but does not reduce the number of lookup operations issued by a peer during query processing.

4.4.8 Handling Values in XML Documents

The psiX system can be extended to support values in XML documents and twig queries in a straightforward way. Recall that each key in the DHT has a list of value pairs associated with it. I introduce an extension to the lookup operation that returns the value pairs for key \(k\) which satisfy an input predicate. Note that this predicate is over the two attributes in a value pair. Thus the peer that stores \(k\) only returns those value pairs that satisfy the predicate to the caller. (A peer can index the value pairs for a key to efficiently evaluate predicates over them.)

Suppose value \(v\) appears with a tag \(C\) in a document with id \(d\). Let \(B\) be the parent of \(C\) in the document. Let \(p\) be the polynomial assigned to the directed edge from \(B\) to \(C\) in the document’s SSG. Then using \(p\) as the key, \((v, d)\) is stored as a value pair in the DHT. Suppose we want to process a query \(A/B[C > 100]\). The signature is constructed for \(A/B/C\), and the relevant docids are found as before. Let us denote these docids by \(D_{sig}\). Since the predicate ‘\(> 100\)’ is associated with \(C\) whose parent is \(B\), the polynomial assigned to the directed edge from \(B\) to \(C\) in the SSG (i.e., \(p\)) is used as a key, and all the value pairs whose first attribute satisfies the predicate ‘\(> 100\)’ are returned. Let \(D_{pred}\) denote these docids. Then \(D_{sig} \cap D_{pred}\) will denote the final result set. Without caching,
one extra DHT lookup is required per predicate in a query.

The extra DHT lookup per predicate can be avoided if the values (both text and numeric data) in a document are summarized and stored along with its signature. Suppose there are $k$ integer values $v_1, v_2, ..., v_k$ corresponding to $k$ different instances of tag $C$ in the document. Let each instance of $C$ have a parent $B$ such that $p$ is the irreducible polynomial assigned to the directed edge from $B$ to $C$ in the SSG. Then a value histogram $H$ is constructed over the values $v_1, v_2, ..., v_k$ and is stored as a pair $(p, H)$ along with the polynomial signature of the document. Each set of values for a tag can be summarized similarly. Text values can be summarized using Bloom histograms. Now a signature representation of a document contains the a polynomial signature and a list of $(p, H)$ pairs. These are stored in the leaf nodes of the signature index. While processing a twig query with predicates, first the polynomial signature of a document in the leaf node is tested for divisibility. On success, for each predicate the corresponding histogram is used to perform the predicate evaluation. On successful evaluation, the document id is returned.

4.5 Performance Evaluation

This section presents performance evaluation of the $psiX$ system. Due to the lack of comparable XML P2P systems readily available to compare with, I compared $psiX$ with $Inverted-index$ both with the polynomial signatures built atop the Chord system [19, 69]. The goal of this experimental evaluation is to test the performance of the proposed indexing techniques rather than evaluating the efficacy of the underlying DHT for scalability, robustness, and load-balancing.

4.5.1 Implementation Details

I implemented both $psiX$ and $Inverted-index$ in C++, and ran all the experiments on a Linux cluster with 12 processing nodes, each with a 2.4GHz Pentium IV processor, 1GB RAM and two 40GB EIDE disks. A P2P network with a varying number of peers was simulated
on this Linux cluster by instantiating a number of logical peers on each processing node using the underlying Chord system. Each logical peer is identified by an IP address and a port number.

4.5.2 Setup for the Experiments

Performance evaluation for both psiX and Inverted-index was carried out by first publishing all the documents from an insertion workload (denoted as \( I\bar{W} \)) into the P2P system. \( I\bar{W} \) was generated from a collection of DTDs published on the Internet. The document signatures in \( I\bar{W} \) were inserted by a set of peers, where one peer was responsible for inserting the document signatures corresponding to one DTD. Twig queries were generated using the same DTDs and a query workload (denoted as \( Q\bar{W} \)) was prepared by randomly mixing the queries based on different DTDs. One peer then issued queries from \( Q\bar{W} \), one at a time. A query was considered complete, once the global identifiers of the relevant documents were retrieved.

**Evaluation Metrics** I compared the effectiveness of psiX and Inverted-index for locating documents for different local cache sizes.\(^5\) I assumed a typical network delay time of 50 ms for each hop in the P2P network. I measured (a) the average number of hops per query, and (b) the query throughput. To compare the publication cost, I measured the total number of insert and lookup operations performed by psiX and Inverted-index to publish all documents in \( I\bar{W} \). I also measured the precision of the signature scheme, and evaluated the scalability of psiX by varying the number of peers.

**Datasets and Document Signatures** I obtained 37 Document Type Definitions (DTD) (e.g., Treebank, DBLP) published on the Internet [73, 74, 79]. (See Appendix). For each DTD, I generated a set of documents using IBM’s synthetic XML data generator. As a result, I generated a rich set of heterogeneous XML documents with different characteristics such as depth, fanout, recursive structure etc. The structural summary graphs were generated

---

\(^5\)The caching along the lookup path peers was not used.
I experimented with three insert workloads by choosing different number of documents for each DTD based on a normal distribution. I chose a different standard deviation ($\sigma$) for each normal distribution, and denote them by $\sigma = 10$, $\sigma = 15$, and $\sigma = \infty$ (i.e., uniform distribution). Table 4.1 shows the maximum document size, average document size, maximum signature size and average signature size for the three $I\bar{W}$ workloads. Figure 4.7(a) shows the number of documents for each DTD in $I\bar{W}$ for the three $\sigma$ values. Note that for DTDs with id 7 and 8, their datasets had 92 and 41 documents respectively. The total number of documents in $I\bar{W}$ for $\sigma = 10$, $\sigma = 15$, and $\sigma = \infty$ was 4606, 3847, and 3633 respectively.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$I\bar{W}$ ($\sigma = 10$)</td>
<td>1,051.42</td>
<td>9,095</td>
<td>50.23</td>
<td>412</td>
</tr>
<tr>
<td>$I\bar{W}$ ($\sigma = 15$)</td>
<td>1,051.30</td>
<td>9,095</td>
<td>50.65</td>
<td>412</td>
</tr>
<tr>
<td>$I\bar{W}$ ($\sigma = \infty$)</td>
<td>1,051.77</td>
<td>9,095</td>
<td>51.01</td>
<td>412</td>
</tr>
<tr>
<td>$Q\bar{W}$</td>
<td>NA</td>
<td>NA</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

**Table 4.1. Insert and query workloads used for evaluation**

using these DTDs and the edges were assigned irreducible polynomials of degree 17. In all, 6338 distinct irreducible polynomials were used.

**Query Sets and Query Signatures** For performing the task of locating documents, I generated a workload of XPath queries for the collected DTDs using the XPath generator provided by the YFilter system [80]. The XPath queries with a maximum query depth of 3 were generated and contained nested path expressions, predicates, wildcard ‘*’ and ‘//’ axis.

I generated a query workload $Q\bar{W}$ containing 3,314 queries using the same set of DTDs. Queries with ‘//’ and ‘*’ had a maximum of 5 signatures per query. The average and maximum sizes of individual polynomial signatures are shown in Table 4.1. For each query, I measured the number of documents that were matched in the P2P network, and plot them in sorted order for the three $I\bar{W}$ datasets. (See Figure 4.7(b)). Some queries had no matching
documents.

4.5.3 Experimental Results

\[ \sigma = \infty \]

\[ \sigma = 10 \]

\[ \sigma = 15 \]

**Figure 4.7.** The characteristics of the insert and query workloads $IW$ and $QW$ in terms of the number of documents published per DTD and the number of documents located for queries

\[ \sigma = 10 \]

\[ \sigma = 15 \]

\[ \sigma = \infty \]

**Figure 4.8.** Cost of publishing the documents in the workload $IW$

I present the experimental results for a P2P network of 240 peers setup on a cluster of 12 processing nodes, each with 20 peers. Later in Section 4.5.3, I examine the performance of $psiX$ when the number of peers are increased. I considered a typical P2P environment where queries from users are more frequent than document publications.
**Figure 4.9.** Evaluation of psiX in terms of average number of hops, throughput, scalability, and precision of signatures

*Insertion Workload* The documents in $I_W$ were published by a set of peers into the P2P system. The psiX system performed lookup operations to fetch the index nodes, and insert operations to update the nodes, split full nodes etc. Inverted-index required only insert operations during publishing as it stores a document signature in the DHT for every distinct tag in the document.

Figure 4.8 summarizes the processing cost of psiX and Inverted-index in terms of the number of lookup and insert operations for the three $I_W$ workloads. In all cases, I observed that psiX required only 31% of the DHT operations as compared to Inverted-index. This is because Inverted-index does one insert operation per distinct tag in the XML document, while psiX inserts one signature per document. Hence, psiX’s model of storing document
signatures in a hierarchical index is more efficient than *Inverted-index*. The publishing cost decreased as $\sigma$ was increased, because the total number of documents published decreased.

*Query Workload* A peer issued queries from $Q$, one query at a time, to locate relevant documents published in the P2P network. The query processing performance of $\psi X$ and *Inverted-index* was measured with a varying size of local cache at the peer from 100,000 bytes to 300,000 bytes.

The plots in Figure 4.9(a) show the average number of hops per query required by $\psi X$ and *Inverted-index* for insert workloads with $\sigma = 10$, $\sigma = 15$, and $\sigma = \infty$ respectively. While the average number of hops required by *Inverted-index* decreased by a small margin as the local cache size increased, the average number of hops required by $\psi X$ decreased very quickly. Except when $\sigma = 10$ and cache size was 100,000 bytes, $\psi X$ yielded similar or significantly better performance as compared to *Inverted-index* for all the three different workloads. This shows that $\psi X$’s model of organizing document signatures achieves better clustering of signatures and cache locality.

For the workload with $\sigma = 10$, the average number of hops required by $\psi X$ was 34% and 65% less than that of *Inverted-index* for cache sizes 200,000 and 250,000 bytes, respectively. For $\sigma = 15$, $\psi X$ required 1.56 hops as compared to 2.89 hops required by *Inverted-index* for cache size of 150,000 bytes. Similar trend was observed for $\sigma = \infty$.

Note that the index node fanout for $\psi X$ was fixed for all the three insert workloads. As more documents from a particular DTD were published, more documents were matched by queries in $Q$, and hence more hops were required. *Inverted-index*, on the other hand, performed similarly for the three workloads, since the hops were determined by lookup operations of the leaf tags in the query patterns. For the same cache size, the cache hits improved marginally as $\sigma$ was increased. However, the value list for each tag can potentially grow quickly. This can be avoided by controlling the size of the value list and distributing the value list, but this would increase the number of hops during query processing.

Figure 4.9(b) shows the query throughput for $\sigma = 10$, $\sigma = 15$, and $\sigma = \infty$ respectively.
To compute the total time required to process $QW$, I assumed a network delay of 50 ms per hop. The cost of polynomial division was also included. I observed that with increase in cache size, $\psi X$ outperformed Inverted-index significantly due to superior cache locality. For instance, when $\sigma = 15$ and cache size was 150,000 bytes, $\psi X$ yielded almost twice the throughput of Inverted-index. The throughput of Inverted-index remained almost flat in all cases.

Cost of Divisibility Tests The cost of dividing polynomial signatures performed during query processing by both $\psi X$ and Inverted-index was small compared to the typical network delays. For $\psi X$ and Inverted-index, the average cost was about 9 ms per query and 4 ms per query respectively. Since $\psi X$ performed signature division at each level in the index, it incurred a higher cost as compared to Inverted-index.

Scalability Test To evaluate the scalability of $\psi X$, I measured the average number of hops per query required by $\psi X$, as the number of peers was increased. Generally, on a Chord DHT with $N$ peers, the number of hops per key lookup is $O \left( \log(N) \right)$. Figure 4.9(c) shows the trend in the average number of hops for a representative case, when the local cache size was fixed at 150,000 bytes. For each insert workload, the average number of hops grew slowly as the number of peers was increased by a factor of two (i.e., 240, 480 and 960 peers). This demonstrates that $\psi X$ fully exploits the scalability of the underlying DHT.

Precision of Polynomial Signatures I evaluated the effectiveness of the polynomial signature scheme proposed in Section 4.2 by measuring the precision of the signatures. For each set of documents generated from a DTD, I computed the number of falsely matched documents whose signatures were divisible by a query signature, but did not actually contain the query pattern. I chose a larger set of documents and queries for each DTD. On an average, 800 documents and 175 queries were used for each DTD. The corresponding precision values for each DTD are shown in Figure 4.9(d). For 29 of the DTDs, the precision was 1.0, which indicated that the signature scheme precisely captured the structural summary of the
documents. The lowest precision value among the DTDs was 0.95. In future, I would like to investigate methods to further improve the precision of the signature scheme.

Signature Construction Cost The average cost of constructing the signatures for documents and twig queries is shown in Figure 4.10. Algorithm 6 describes the process of constructing one single polynomial signature for a document or query. As expected, overall the cost of constructing signatures for documents was higher than that for queries due to the higher number of tree nodes. It is evident that the cost of signature construction is small compared to the network delays experienced in P2P systems.

4.6 Other Applications of Polynomial Signatures and the Signature Index

In this dissertation, the signature scheme has been used for a P2P environment where the existence of a twig pattern is tested and XML data is heterogeneous in nature. However, the signature scheme can be applied to a centralized environment when XML data is heterogeneous. The signature index can be built over document signatures except that each entry in an index node has a pointer to the actual signature that is stored as a variable length record. This is because each index node is of fixed page size (e.g., 4K, 8K). The cost of signature operations such as division now become noticeable.
4.7 Conclusions

I have presented a new system called psiX, built atop a DHT framework, for efficiently publishing and locating XML data in a peer-to-peer environment. Each XML document is mapped into an algebraic signature that captures the structural summary of the document. This document signature is published by a peer into the network. The participating peers in the network collectively maintain a distributed and hierarchical index over the signatures. By virtue of the signature scheme, document signatures of structurally similar documents tend to be stored at the same peer. As a result, fewer peers are contacted to locate relevant documents for user queries. Our experimental evaluation shows that psiX provides an efficient location service for XML data in typical P2P environments where user queries are more frequent than document publications. In this work, I have so far assumed that peers do not behave maliciously. I would like to address security concerns in psiX as part of future work.
CHAPTER 5
CONCLUSIONS

In this chapter, I summarize the contributions of this dissertation and provide directions for future research.

5.1 Contributions

In this dissertation, I have addressed the problem of XML indexing and query processing in two environments, namely (a) a traditional environment where XML data is centrally stored, and (b) a growingly popular P2P environment. For these environments, I have developed two systems called PRIX [60, 63, 56] and psiX [62]. At the core of each system is an efficient twig pattern matching technique using a novel XML indexing method suited for the challenges arising in the target environment. The aforementioned environments have different models of storing data, i.e., centralized vs distributed. Moreover, the query processing requirements are different, i.e., finding all occurrences of a twig pattern in a traditional environment vs testing the existence of a twig pattern in a P2P environment.

5.1.1 The PRIX System

I propose a new way of indexing XML documents and processing twig patterns in an XML database. Every XML document in the database can be transformed into a sequence of labels by Prüfer’s method that constructs a one-to-one correspondence between trees and sequences. During query processing, a twig pattern is also transformed into its Prüfer sequence. By performing subsequence matching on the set of sequences in the database, and performing a series of refinement phases that I have developed, all the occurrences of a twig pattern can be found in the database. This approach allows holistic processing of a twig pattern without decomposing the twig into root-to-leaf paths and processing these
paths individually. Furthermore, I show that all correct answers are found without any false dismissals or false alarms. Experimental results demonstrate the performance benefits of PRIX.

Further, I have successfully applied this sequencing approach to two other data-intensive problems namely (a) XML document filtering [42] and (b) XML streaming [61].

5.1.2 The $psi$X System

I have developed a new system called $psi$X that runs on top of an existing distributed hashing framework. Under my system, each XML document is mapped into an algebraic signature that captures the structural summary of the document. The participating peers in the network collectively maintain a distributed and hierarchical index over the signatures. By virtue of the signature index, the signatures of documents with similar structural characteristics tend to be stored together at the same peer, and a search for document sources is resolved quickly. The experimental study demonstrates that $psi$X provides an effective and scalable location service for a wide variety of XML documents in the P2P environments. Note that the proposed signature scheme can also be applied to indexing and querying heterogeneous XML data.

5.2 Directions for Future Research

Tree-structured data is common in domains such as biology, life sciences, computer networks, and the World Wide Web. An important problem is to mine frequent tree patterns in the data. Mining sequences and subsequences has been well studied by the data mining community. I would like to explore the challenge of mining frequent tree patterns by using existing solutions for mining sequences and subsequences, once tree-structured data is represented as a sequence by Prüfer’s method.

As vast amounts of information on the Internet are being marked up in XML by independent data sources, users could potentially query heterogeneous XML repositories in
a similar way to that of searching textual content on the Internet via search engines such as Google. However, the existing twig query processing techniques were not particularly developed for this purpose. I plan to investigate the challenges that arise when indexing, storing, and query processing heterogeneous XML content, while still providing effective and intuitive mechanisms for users to search them. In my opinion, a key challenge is to effectively capture the heterogeneity of the documents that would enable the pruning of large amounts of irrelevant documents early on during query processing. I would like to explore the applicability of the polynomial signature scheme developed as part of this dissertation work to capture the heterogeneity and summarize XML documents. By indexing the signatures for documents and searching them during query processing, I believe that irrelevant documents can be pruned away early on.

Although handheld devices such as PDA’s and smartphones are limited in their storage and computing capacities, the computing mobility and convenience that they offer has resulted in a dramatic rise in their usage. I believe that data summarization and filtering are attractive schemes for presenting large amounts of useful and relevant information to the users of such devices, as they can provide sufficiently rich information to users despite the computing limitations of these devices. I would like to develop general techniques that enable users to tune the level of information richness that they desire depending on their computing models.

Multicore processors are becoming common and provide opportunities for parallelizing existing software for uniprocessors. I would like to develop parallel algorithms for CPU intensive tasks over large-scale data such as document filtering and data summarization.

As large volumes of information are being generated on the Internet, developing scalable information systems has become a challenge. Although successful search engines such as Google and Yahoo allow users to implicitly publish their content on the Internet, users lack easy and efficient mechanisms for large-scale dissemination of their content to interested parties. Existing publish-subscribe systems for large scale XML data dissemination do not lend themselves well as the number of publishers increases, because routing state
that needs to be maintained grows linearly with the number of publishers.

Jointly with the faculty at the University of Arizona, we are investigating a peer-to-peer based approach for large scale XML data dissemination – *any user can publish content as well as subscribe to any content* – that provides decentralization, scalability, and adapts to the dynamism of users joining and leaving the network [59]. We are exploring how the distributed indexing scheme in \( \psi X \) can be adapted to index user profiles. A major challenge is to manage the increased rate of index updates due to the changing user interests. Security and privacy threats hinder the usage of P2P applications in practice. I would like to investigate techniques to overcome these threats in P2P environments.

Indexing and querying graph-structured data is gaining importance due to its use in domains such as biology, life sciences, social networking, ontologies etc. The polynomial signature scheme used in \( \psi X \) can summarize graphs and quickly test for existence of patterns. I would like to investigate how the signature scheme can be useful for indexing and querying graphs.

Social networking on the Internet has become common. In P2P systems, verifying the authenticity of the published data is important due to their decentralized nature and lack of any trusted authority. Can social networking help identify trusted publishers? I would like to extend \( \psi X \) to incorporate social networking information to rank data publishers who have been identified during the first phase of query processing \( i.e., \) locating relevant XML documents. During the second phase of actually evaluating the query over matching documents, \( \psi X \) can choose these documents based on the ranking of its publisher and network characteristics such as delays. The goal is to provide a user with the most trustworthy results early on and quickly. A challenging task is to perform joins across the publishers of XML data. I would like to investigate the possibility of bipartite graph matching to determine an optimal evaluation order.
REFERENCES


