XML Structured and Unstructured Query Processing

Jiang Li

School of Information and Communication Technology
Griffith University

A thesis submitted in fulfilment of the requirements of the degree of

Doctor of Philosophy

July, 2011
Acknowledgements

First and foremost I would like to express my gratitude to my supervisor Dr. Junhu Wang, for his continuous support throughout my Ph.D. studies. Without his excellent technical and editorial advice, I cannot complete this dissertation. He has taught me how to explore a research area and see research in its full depth. In addition, it has been an honor to be his first Ph.D. student.

Then I would like to thank Griffith University for its excellent facilities in ICT school which brought much convenience to my Ph.D. studies. With its financial support, I can fully dedicate myself to research.

Last but not least, I would like to thank my wife Jingjing Kang for her understanding and love during the past few years. She was always encouraging me when I met difficulties. Without her support, I cannot complete my Ph.D. studies. I also want to express my deepest gratitude to my parents for their love, dedication and regretless support during my growth.
Abstract

XML has become a standard for representing, storing and exchanging data, so effectively and efficiently retrieving data from XML data sources becomes increasingly important. In this thesis, we mainly study two types of XML queries: XML structured query and XML keyword search.

For XML structured query, we focus on twig pattern matching, which lies in the center of most XML query languages (e.g., XPath, XQuery). Existing twig pattern matching algorithms can be classified into two-phase algorithms and one-phase algorithms. We first propose two novel one-phase holistic twig matching algorithms, TwigMix and TwigFast, which combine the efficient selection of useful elements (introduced in TwigStack) with the simple lists for storing final solutions (introduced in TwigList). TwigMix simply introduces the element selection function getNext of TwigStack into TwigList to avoid manipulation of useless elements in the stack and lists. TwigFast further improves this by introducing some pointers in the lists to completely avoid the use of stacks. On the other hand, previous twig pattern matching algorithms may incur other redundant computation, so we propose two approaches, namely re-test checking and forward-to-end, which can reduce the redundant computation and can be easily applied to both holistic one-phase and two-phase algorithms.

Improving the effectiveness of XML keyword search remains an open problem. In this thesis, we first present XKMis, which divides the nodes into meaningful and self-containing information segments, called minimal information segments (MISs), and return MIS-subtrees which consist of MISs that are logically connected by the keywords. The MIS-subtrees are closer to what the user wants. XReal [1] utilizes the statistics of underlying data to resolve keyword ambiguity problems. However, we found their proposed
formula for inferring the search-for node type suffers from inconsistency and abnormality problems. Therefore, we propose a dynamic reduction factor scheme as well as a novel algorithm DynamicInfer to resolve these two problems. Then, we resolve the ambiguities of keywords by exploiting users’ typing habit in constructing keyword queries. We propose an approach which infers and ranks a set of likely search intentions. In a search intention, each keyword has a specific meaning. The result subtrees of the inferred likely search intentions are returned to users in clusters, which can significantly save users’ browsing time. Finally we explore the application of query suggestion in XML keyword search and propose a novel interactive XML query system XQSuggest, which mainly targets non-professional users who roughly know the contents of the database.

In summary, this thesis presents several novel algorithms to improve the efficiency of twig pattern matching. It also presents several approaches to resolve the ambiguity of keywords and improve the effectiveness of XML keyword search.
Statement of Originality

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signed:

Month/Year
List of Publications


## Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Background .......................... 1
   1.1.1 XML and XML Query Languages 1
   1.1.2 Schema of XML Data ................. 4
1.2 Research problems .................... 5
   1.2.1 XML Twig Pattern Matching ........ 5
   1.2.2 XML Keyword Search .............. 6
1.3 Approach Overview ................... 6
   1.3.1 XML Twig Pattern Matching ........ 6
   1.3.2 XML Keyword Search .............. 7
1.4 The Contributions .................... 8
1.5 Thesis outline ....................... 10

### 2 Related Works

2.1 Labelling schemes .................... 11
   2.1.1 Region-based Labelling Scheme ...... 11
   2.1.2 Prefix-based Labelling Scheme ...... 12
   2.1.3 Other Labelling Schemes .......... 13
2.2 XML Twig Pattern Matching .......... 14
2.3 XML Keyword Search .................. 19
4.2.2 Formal Definitions ........................................... 56
4.2.2.3 Partitioning a Data Tree into MISs ..................... 58
4.2.4 Further Ranking of Answers ................................. 59
4.2.3 Algorithm .................................................... 60
4.2.3.2 XKMis_Construct ........................................ 61
4.2.3.3 XKMis_Output ........................................... 64
4.2.4 Summary ..................................................... 65

4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search 65
4.3.1 Problem of XReal ............................................ 65
4.3.2 Background .................................................. 67
4.3.2.2 Overview of XReal ..................................... 68
4.3.3 Analysis of XReal’s Weaknesses .............................. 69
4.3.4 Inferring Search-for Node ................................... 72
4.3.4.1 Preliminary definitions .................................. 73
4.3.4.3 Algorithm ................................................ 75
4.3.5 Summary ..................................................... 78

4.4 Exploiting User’s Typing Habit and Data Statistics for Effective XML Keyword Search 78
4.4.1 Observation .................................................. 78
4.4.2 Inferring the Meaning of Two Adjacent Keywords ........ 81
4.4.2.2 Inferring the Word Types of Two Adjacent Keywords ... 83
4.4.3 Inferring Likely Search Intentions ............................ 85
4.4.3.3 Algorithms .............................................. 89
4.4.4 Generating Results .......................................... 94
4.4.5 Summary ..................................................... 97

4.5 XQSuggest: an Interactive XML Keyword Search System 97
4.5.1 Introduction ................................................ 97
CONTENTS

4.5.2 Query Suggestion ................................................. 98
  4.5.2.1 Semantic String ........................................ 98
  4.5.2.2 Boolean Operators .................................... 99
  4.5.2.3 Result of Query ...................................... 99
4.5.3 Algorithm ....................................................... 100
  4.5.3.1 Notation ............................................... 100
  4.5.3.2 The Stack-based Algorithm ......................... 101
  4.5.3.3 Optimization Techniques ........................... 101
4.5.4 Summary ....................................................... 103

4.6 Comparison of Approaches .................................. 103

4.7 Experiments ...................................................... 104
  4.7.1 Experiment Set-up ....................................... 104
  4.7.2 XKMis vs. XRank, SLCA and MaxMatch ................. 104
    4.7.2.1 Processing time ................................ 105
  4.7.2.2 Scalability ........................................ 107
    4.7.2.3 Effectiveness .................................. 107
  4.7.3 DynamicInfer vs. XReal ................................. 110
    4.7.3.1 Results of Inferring Search-for Node .......... 110
  4.7.4 XInfer vs. XReal, XBridge, MaxMatch, DynamicInfer 111
    4.7.4.1 Index Size ...................................... 113
    4.7.4.2 Search Quality ................................ 113
    4.7.4.3 Efficiency .................................... 120
  4.7.5 XQSuggest vs. MaxMatch, XInfer ..................... 121
    4.7.5.1 Search Quality ................................ 121
4.8 Summary ....................................................... 122

5 Conclusion and Future Work ................................ 125
  5.1 Thesis Contributions ........................................ 125
  5.2 Future Work ................................................. 127

References ............................................................ 129
# List of Figures

1.1 An example of XML document and XML data tree .......................... 3
1.2 DTD Example ......................................................... 4
1.3 example data tree $t$ and tree pattern $Q$ ................................. 5
1.4 A result candidate ................................................... 6
1.5 Overview of twig pattern matching ....................................... 7

2.1 Region-based labelling schemes .......................................... 12
2.2 DeweyID labelling schemes ............................................. 12
2.3 Example of TJEssential .................................................. 15
2.4 Example of TwigStack+ .................................................. 16
2.5 Example of sub-optimality of P-C edge .................................. 16
2.6 Example of SLCA and ELCA ............................................ 20
2.7 Example of MaxMatch ................................................... 21

3.1 An example to explain the basic ideas of TwigMix ....................... 28
3.2 An example to illustrate the basic ideas of TwigFast ..................... 32
3.3 example data tree $t$ and tree pattern $Q$ ................................ 36
3.4 Processing Time(ms) of TwigMix and TwigFast ............................ 47
3.5 Processing time of queries with different characteristics ................ 48
3.6 TwigFast vs TwigFast* ................................................ 50
3.7 TwigStack, TJEssential, TwigFast* vs TwigStack* ....................... 50

4.1 SigmodRecord data tree and minimal information segments ............ 52
4.2 DBLP data tree and partial search results ............................. 53
4.3 Example search results over the SigmodRecord data tree ............... 54
LIST OF FIGURES

4.4 Search results over the SigmodRecord data tree, each node in a result represents a MIS .......................... 57
4.5 Example to explain algorithms ........................................... 60
4.6 An XML data tree of SigmodRecord ............................... 60
4.7 Two Data Set .......................................................... 69
4.8 An XML data tree of customers ......................................... 70
4.9 A Sample XML data tree of SigmodRecord ...................... 78
4.10 Three Screenshots of Our System ................................. 80
4.11 An Example of Algorithm ............................................. 92
4.12 Two extreme cases .................................................... 93
4.13 Data tree t ............................................................ 97
4.14 A screen shot of query suggestion in XQSuggest .......... 98
4.15 Processing time: XKMis vs. XRank and SLCA ............... 106
4.16 Processing time with increasing document size ............... 107
4.17 Precision: XKMis vs. XRank, SLCA and MaxMatch .......... 108
4.18 Recall: XKMis vs. XRank, SLCA and MaxMatch ............... 109
4.19 Scalability of index .................................................. 113
4.20 Precision: XInfer vs. XReal, XBridge, MaxMatch and DynamicInfer 115
4.21 Recall: XInfer vs. XReal, XBridge, MaxMatch and DynamicInfer 115
4.22 Processing time of XInfer ............................................ 120
4.23 Scalability of efficiency of XInfer ................................. 121
4.24 Precision, Recall and F-Measure on Mondial Data Set ........ 121
4.25 Precision, Recall and F-Measure on SigmodRecord Data Set 122
4.26 Precision, Recall and F-Measure on WSU Data Set .......... 122
4.27 Precision, Recall and F-Measure on DBLP Data Set .......... 123
List of Tables

3.1 Limitation of TwigMix .......................................... 33
3.2 Example of redundant calls of getNext ......................... 37
3.3 Queries over TreeBank, DBLP and XMark ....................... 46
3.4 Number of elements pushed into S ................................ 47
3.5 Number of elements appended to result lists ................... 48
3.6 Queries over TreeBank, DBLP and XMark ....................... 49

4.1 \( C_{far}(T, q) \) for Inconsistency Problem (Query \( q \): \{article Karen\}) .................. 70
4.2 \( C_{far}(T, q) \) for Inconsistency Problem 2 ......................... 71
4.3 \( C_{far}(T, q) \) for Abnormality Problem .......................... 71
4.4 Sample Word Types .................................................. 52
4.5 Keyword queries ..................................................... 105
4.6 Comparison on F-Measure: XKMis vs. XRank, SLCA and MaxMatch ...... 110
4.7 Results on Resolving Inconsistency Problem 1 ..................... 111
4.8 Results on Resolving Inconsistency Problem 2 ..................... 112
4.9 Results on Resolving Abnormality Problem ....................... 112
4.10 Queries ................................................................. 114
4.11 Index size ............................................................. 116
4.12 Number of returned search intentions by XInfer .................. 116
4.13 Comparison on F-Measure: XInfer vs. XReal, XBridge, MaxMatch and DynamicInfer ........................................ 118
4.14 Ranking effectiveness of XInfer .................................... 119
4.15 Influencing factors .................................................... 119
1

Introduction

1.1 Background

1.1.1 XML and XML Query Languages

XML [2] stands for eXtensible Markup Language and is used for organizing structured and semi-structured information. The emergence of XML is mainly due to the limitations of other two markup languages: Hypertext Markup Language (HTML) and Standard Generalized Markup Language (SGML). HTML uses pre-defined tags (e.g., html, head, frame, br, etc) to instruct the browser how to display the texts or data on the screen, but they do not tell what the data really mean. SGML is too large and complex for most common desktop and web applications because of its complicated specifications. XML is regarded as an abbreviated version of SGML. The complex and less used parts are omitted.

An XML document mainly consists of two parts: mark up (tag) and content. A tag begins with “<” and ends with “>”. The texts that are not tags are content. There are three types of tags: start-tags (e.g., <DBLP>), end-tags (e.g., </DBLP>) and empty-element tags (e.g., <DBLP />). XML uses tags to describe data, which can make the information more meaningful and understandable. Additionally, the tags can be nested, which reflects the structure or hierarchy of data. Another two important concepts in XML are element and attribute. An element is a logical component which starts with a start-tag and ends with a matching end-tag or only consists of an empty-element tag. The texts between the start and end tag are the contents of an element.
1. INTRODUCTION

An attribute is a name/value pair that exists within a start-tag or empty-element tag. Figure 1.1(a) is an example of XML document of DBLP entries.

XML uses tree-structured data model, so the elements, attributes, and texts are modeled as nodes on a tree. With the edges between nodes, the structural relationships (e.g. Ancestor-Descendant, Parent-Child, siblings) are clearly represented. Additionally, XML data is order sensitive. Suppose a book is organized with XML format, the order of chapters or paragraphs is important. Wrong order may bring incorrect semantic of data. Figure 1.1(b) presents an XML data tree with DBLP bibliography entries. The ‘author’ nodes are order sensitive because people need to know who the first or second author of a paper is.

XML has become a popular standard for exchanging data between different applications or over the Internet without any conversion. At beginning, XML is mainly designed for documents, but currently it is widely used for representing arbitrary data structures. For example, more and more companies provide web and data services on the Internet, in which the request and response are encoded in XML. Everyday, huge amounts of XML data are generated and processed. How to store and efficiently query these data becomes an important research problem.

XPath [3] and XQuery [4] are two standard query languages for retrieving XML data. XQuery is derived from the Quilt [5] query language and borrows some features from XPath. Both XPath and XQuery use path expression as the main components to address parts of an XML document. XQuery can natively support XPath, which means XQuery can do everything that XPath can do, but XQuery also supports more complex functions. For example, we can express more complicated selection criteria with XQuery. Basically, the data search in XML documents not only includes value search, but also structure search. For example, an XPath query may look like this:

```
/DBLP/paper[year="2005"]/title
```

The query above selects the title of the papers published in 2005. It represents the paths from the root to the elements that we want to retrieve, so we may get more than one answer. Applying this query on the XML document in Figure 1.1(a), the elements “XML Query Processing” and “Keyword search in Relational Database” will be retrieved. An XPath query can be represented as a small tree (as known as twig pattern). Answering an XPath query is actually to find all matches of the twig pattern.
1.1 Background

(a) XML document

(b) XML data tree

Figure 1.1: An example of XML document and XML data tree
1. INTRODUCTION

This thesis presents approaches for accelerating the process of twig pattern matching. For unprofessional user, it is difficult to learn XQuery language to retrieve information from XML documents. The more feasible solution is to use keyword search (see Section 1.2.2). The system should only return useful information instead of the whole document according to the submitted keywords. In this thesis, we also present approaches to effectively answer keyword queries.

1.1.2 Schema of XML Data

An XML document can optionally have a schema to define its structure. Schema plays an important role in exchanging data between different parties. Applications can easily interpret the data and validate an XML document through checking whether this XML document conforms to its corresponding schema.

Document Type Definition (DTD) is a commonly used schema language, which can define document structure via element and attribute-list declarations. For each element, it specifies the names of its sub-elements and attributes using regular expressions with operators * (zero or more elements), + (one or more elements), ? (optional), and | (or). Figure 1.2 presents a sample DTD of the XML document in Figure 1.1(a).

XML Schema is another powerful schema language, which provides richer capabilities than DTD. The most important feature of XML Schema is that it is written in XML (XML-based), which means it is extensible for future additions. Additionally, XML Schema supports data types and name spaces.

---

Figure 1.2: DTD Example

of this query from the XML data tree (i.e., twig pattern matching, see Section 1.2.1).

```xml
<!ELEMENT DBLP (paper*)>
<!ELEMENT paper (title,authors,conf,year)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (author+)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT conf (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```
1.2 Research problems

1.2.1 XML Twig Pattern Matching

Twig pattern matching, which is to find all matchings of a twig query from an XML data tree, is a core operation of XML query processing and has been widely studied (e.g., [7], [8], [9], [10], [11], [12], etc).

An XML twig query is a complex selection predicate on both structure and content of an XML document. Suppose every node in an XML data tree is labeled with a tag in \( \Sigma \) (\( \Sigma \) is a set of XML tags). Twig pattern is a small tree with every node labeled with a tag in \( \Sigma \), and every edge labeled / or //. Edges labeled / (resp. //) are called /-edges (resp. //-edges), and they represent parent-child (resp. ancestor-descendant) relationships between nodes. In what follows, we use \( N(t) \) to denote the node set of any tree \( t \), and \( \text{label}(v) \) to denote the label of node \( v \) in a data tree or twig pattern.

A matching of twig pattern \( Q \) in a data tree \( t \) is a mapping \( \delta \) from \( N(Q) \) to \( N(t) \) such that (1) \( \forall v \in N(Q), \text{label}(v) = \text{label}(\delta(v)) \), and (2) for every /-edge \( (v_1, v_2) \) in \( Q \), \( (\delta(v_1), \delta(v_2)) \) is an edge in \( t \), and for every //edge in \( Q \), there is a path from \( \delta(v_1) \) to \( \delta(v_2) \). Regarding the nodes in \( Q \) as a tuple, every matching of \( Q \) in \( t \) produces a tuple of nodes in \( t \). When there is no confusion, we simply refer to such a tuple as a matching of \( Q \) in \( t \). Figure 1.3 shows a data tree \( t \) and a twig pattern \( Q \), \((a_2, b_2, c_1)\) is the only matching of \( Q \) in \( t \).

Our first research problem is:

Given a twig pattern \( Q \) and an XML data tree \( t \), find all occurrences of \( Q \) in \( t \) efficiently.
1. INTRODUCTION

Keyword search has long been used to retrieve information from collections of text documents. Recently, keyword search in databases re-attracted attention of the research community because of the convenience it brings to users - there is no need for users to know the underlying database schema or complicated query languages.

A keyword query is a finite set of keywords $K = \{w_1, ..., w_n\}$. Given a keyword $w$ and a data tree $t$, the search of $w_i (1 \leq i \leq n)$ in $t$ will check both the labels of internal nodes and values of leaf nodes for possible occurrence of $w_i$. The subtrees that are are useful and desired to the user should be returned. For example, if the user submits the keyword query \{author Karen\} over the data tree in Figure 1.4, the subtree in Figure 1.4 is a good result candidate.

Our second research problem is:

*Given a keyword query $K$ and an XML data tree $t$, find result subtrees that contain the keywords in $K$ and are desired by the user.*

The difficulty of XML keyword search is how to determine a subtree is desired by the user or not.

1.3 Approach Overview

1.3.1 XML Twig Pattern Matching

Developing an XML twig pattern matching algorithm involves two major steps: (1) building positional information for each node in an XML data tree using a proper labelling scheme. (2) utilizing the positional information to find matches without travers-
1.3 Approach Overview

The original XML document. Figure 1.5 presents the basic steps of twig pattern matching. We can see that each node has a tuple which represents the position of this node in the data tree. Therefore, each keyword has an associated list which includes all of the positions of the nodes that contain this keyword. The twig pattern matching algorithm takes the lists of positions and twig pattern as the input for computation.

XML labelling scheme In XML twig pattern matching, the most commonly used labelling schemes are region-based labelling scheme and prefix-based labelling scheme. We introduce different types of labelling schemes in Section 2.1.

Overview of XML twig pattern matching There are mainly two types of XML twig pattern matching algorithms: two-phase and one-phase holistic twig pattern matching algorithms. TwigStack [8] is the most important two-phase holistic algorithm and has many extensions. In the first phase, TwigStack outputs matches for each root-to-leaf paths, and then in the second phase it merges these path matches together. When only //-edge appears in a twig pattern, every path match found in the first phase is part of final solutions. One-phase algorithms avoid the high cost of merging phase in the two-phase algorithms. It maintains the whole final solutions with a certain data structure and enumerates them efficiently. Twig2Stack [13] is a typical one-phase twig pattern matching algorithm, which stores final solutions in hierarchical stacks.

1.3.2 XML Keyword Search

Similar to XML twig pattern matching, developing XML keyword search also involves two major steps: (1) build positional information for each node in XML data tree; (2) identify the return nodes and result subtrees using a proper approach.
1. INTRODUCTION

XML labelling scheme The most commonly used labelling scheme is prefix-based labelling scheme because it is easy to find the Lowest Common Ancestor (LCA) of several nodes with prefix-based positional information.

Overview of XML keyword search To answer an XML keyword query, we first need to meaningfully connect the keyword occurrences together. Most existing algorithms uses LCA or its variants (ELCA [58], SLCA [59], VLCA [61], etc) to achieve this. The LCA nodes will act as the root candidates of result subtrees. After the LCA nodes are found, some systems identify the desired return nodes (XSeek [65], XReal [1], XBridge [67]) or remove irrelevant information from the result subtrees (MaxMatch [66], RTF [14], ValidMatch [15], etc).

1.4 The Contributions

The overall contribution of this thesis is that it proposes several novel approaches for efficient XML twig pattern matching and effective XML keyword search. Specifically, we make the following contributions:

Twig pattern matching

1. We present two novel one-phase holistic twig matching algorithms, TwigMix and TwigFast, which combine the efficient selection of useful elements (introduced in TwigStack) with the simple lists for storing final solutions (introduced in TwigList). TwigMix simply introduces the element selection function of TwigStack into TwigList to avoid manipulation of useless elements in the stack and lists. TwigFast further improves this by introducing some pointers in the lists to completely avoid the use of stacks. Our experiments show TwigMix significantly and consistently outperforms TwigList and HolisticTwigStack (up to several times faster), and TwigFast is up to two times faster than TwigMix.

2. We propose two approaches, namely re-test checking and forward-to-end, which can reduce the redundant computation and can be easily applied to both holistic one-phase and two-phase algorithms. We present a holistic one-phase algorithm TwigFast* and a holistic two-phase algorithm TwigStack* which extend TwigFast and TwigStack respectively by applying our proposed approach. The
1.4 The Contributions

experiments show that TwigFast* and TwigStack* can significantly improve the efficiency by avoiding the redundant computation.

XML keyword search

1. We present XKMis, a system for keyword search in XML documents. We divide the nodes into meaningful and self-contained information segments, called minimal information segments (MISs), and return MIS-subtrees which consist of MISs that are logically connected by the keywords. The MIS-subtrees are closer to what the user wants. The MIS-subtrees enable us to use the region code of XML trees to develop an algorithm for the search which is more efficient especially for large XML trees. We report our experiment results, which verify the better effectiveness and efficiency of our system.

2. We identify two problems (i.e., inconsistency and abnormality problems) of XReal in inferring SNTs. We propose a dynamic reduction factor scheme to resolve the identified problems. We provide algorithm DynamicInfer to incorporate this scheme to infer the SNT of a query. We conducted an extensive experimental study which verified the effectiveness of our approach.

3. We exploit the user’s typing habit in constructing queries to resolve the ambiguities of keywords. To achieve our goal, we design a formula to infer the desired meaning of a pair of adjacent keywords in a query without considering other keywords, which takes into account the statistics and structural properties of a keyword’s different meanings. We propose the Pair-wise Comparison Strategy (PCS) to infer a set of likely search intentions of a keyword query, which utilizes the inference results of pairs of adjacent keywords. We developed an XML keyword search system Xinfer that realizes the techniques we propose. We conducted an extensive experimental study which verified the better effectiveness of our approach.

4. We explore the application of query suggestion in XML keyword search and propose a novel interactive XML query system XQSuggest, which mainly targets non-professional users who roughly know the contents of the database. Our system extends conventional keyword search systems by instantly suggesting several understandable semantic strings after each keyword is typed in, so that the users can
1. INTRODUCTION

...easily select their desired semantic string, which represents a specific meaning of the keyword, to replace the ambiguous keyword. We provide a novel algorithm to compute the final results. Experimental results are provided to verify the better effectiveness of our system.

1.5 Thesis outline

The remainder of this thesis is organized as follows. We review related works in Chapter 2. Chapter 3 presents two one-phase algorithms (i.e., TwigMix and TwigFast) for efficiently filtering out useless elements and an approach for resolving the redundant computation in previous twig pattern matching algorithms. In Chapter 4 we study XML keyword search. We first propose a keyword search system XKMis which improves the effectiveness and efficiency by dividing an XML data tree into meaningful semantic segments. In order to improve the precision of inferring the search-for node type, we propose an approach and the corresponding algorithm DynamicInfer. Additionally we propose a keyword search system XInfer which exploits users’ typing habit and XML data statistics to improve the search effectiveness. Then we present an interactive keyword search system XQSuggest which improves the usability and effectiveness of keyword search. Finally, Chapter 5 concludes this thesis and shows some future research work.

Some of the material in this thesis appeared in our papers [16], [17], [18], [19], [20], [21].
Related Works

In this chapter, we first introduce the labelling schemes that are extensively used in twig pattern matching and XML keyword search. Then we discuss the existing work on twig pattern matching. Finally, the approaches on XML keyword search are presented.

2.1 Labelling schemes

Labelling schemes provide the approaches for representing the positional information of each node in an XML data tree. With positional information, the structural relationship between different nodes can be easily determined. Traditionally, the relationships (e.g., ancestor-descendant, parent-child, etc) between nodes can be determined by traversing the tree. However, this approach is too expensive to meet the requirements of query performance. Therefore, a number of XML labelling schemes were proposed.

In this section, region-based labelling scheme and prefix-based labelling scheme will be introduced. They have been extensively used in twig pattern matching and XML keyword search. We also present some other labelling schemes which are proposed to solve the re-labelling problem when updates happen.

2.1.1 Region-based Labelling Scheme

In region-based labelling scheme, each node $v$ is coded with a tuple of three values: $(v.\text{start}, v.\text{end}, v.\text{level})$. Such a labelling scheme has several useful properties: (1) ancestor-descendant and parent-child relationships can be identified in constant time: $\forall v_1, v_2 \in Nodes(t)$, $v_1$ is an ancestor of $v_2$ iff $v_1.\text{start} < v_2.\text{start} \leq v_2.\text{end} < v_1.\text{end}$,
2. RELATED WORKS

(2) $v_1$, $v_2$ do not have ancestor-descendant relationship, and $v_1$ lies in a path to the left of the path where $v_2$ lies iff $v_1.end < v_2.start$ (see Figure 2.1). The value of level of each node is straightforward. The start and end position can be computed in several ways. Dietz proposed a computing method that uses pre-order and post-order in tree’s traversal as the start and end positions respectively [22]. Another computing method uses the number of characters counted from the start of an XML document [23]. Figure 2.1 (a) and (b) shows examples of these two position computing method.

2.1.2 Prefix-based Labelling Scheme

Prefix-based labelling scheme is different from region-based labelling scheme. The structural relationship is determined through checking whether a code is a prefix of another. Obviously, this method is more expensive than just comparing numbers. DeweyID labelling scheme [24] is a typical prefix-based labelling scheme.
2.1 Labelling schemes

DeweyID labelling scheme comes from Dewey Decimal Classification [25]. Each code is assigned with a local order, for example, the $n^{th}$ child of a node will be assigned with $n$. Then the local order of a node will be concatenated to its parent’s code and the delimiter. Therefore, an ancestor’s code is the prefix of its descendants’ label. We can use this feature to determine the structural relationship between nodes. In addition, the precedence of sibling nodes is also kept. An example is shown in Figure 2.2.

2.1.3 Other Labelling Schemes

The coding scheme accelerates the determination of structural relationships between nodes, but it brings a serious problem which is called relabelling. As we know, an XML data source may not be static. Some new elements may be inserted and some old ones may be deleted. When the data is changed, the codes assigned before may become invalid. In order to keep the codes valid, the nodes have to be relabelled. However, this process is very time consuming if there are a large number of nodes need to be relabelled. Some researchers proposed new labelling schemes to avoid relabelling.

In order to avoid the relabelling problem of region-based labelling scheme, the extended region-based labelling scheme was proposed in [26]. This scheme also represents the label as a tuple of three values, but the difference is the start position uses an extended pre-order and the end position is replaced by the range of its descendants. Therefore, the label of a node is represented as (start, size, level). Another way of handling relabelling problem is to localize relabelling. When updates happen, just a little part of labels need to be re-computed. Based on this idea, [27] proposed the relative region-based labelling scheme. Each node’s start and end positions are relative offset to its parent’s position. The offset is calculated with the number of bytes counted from the beginning of parent node. Another region-based coding scheme called QRS (Quartering-Regions Scheme) utilizes the feature of float point number to obtain extra space for future insertions [28].

OrdPath [29] is a kind of prefix-based labelling scheme and can be regarded as an extension of DeweyID. It can resolve the relabelling problem of DeweyID to some extent. In the initial state, only odd numbers can be concatenated as labels. When insertion happens between two sibling nodes, the even numbers will work as caret followed by a new odd number.
2. RELATED WORKS

Tree-based labelling scheme is firstly introduced in [30]. Lee et al applied k-ary tree to represent the XML data. Each node is assigned an ID according to the tree traversal order. As we know, with the k-ary tree, there exists mapping functions between a parent and its child.

\[
\text{parent}(i) = \left\lfloor \frac{(i - 2)}{k} + 1 \right\rfloor \tag{2.1}
\]

\[
\text{child}(i, j) = k(i - 1) + j + 1 \tag{2.2}
\]

With the Formula (2.1) above, we can get the parent of a node whose ID is \(i\). If we want to get the \(j^{th}\) child of a node whose ID is \(i\), the Formula (2.2) can be used. [31] proposed an improved tree-based labelling scheme called PBiTree. It utilizes the perfect binary tree to represent XML data. Because each node just has two children, so a specific ‘binarization’ process is needed to transform an XML data tree to a binary tree. The biggest strength of this labelling scheme is sufficient structural information can be obtained from the properties of perfect binary tree. The formula is shown below:

\[
F(n_i, h_j) = 2^{h_j+1} \cdot \left\lfloor \frac{n_i}{2^{h_j+1}} \right\rfloor + 2^{h_j} \tag{2.3}
\]

With the formula above, given a node \(n_i\), its ancestor \(n_j\) at height \(h_j\) can be obtained.

The Prime Number Labelling Scheme [32] is based on the Divisibility property, which is integer \(A\) cannot be divisible by \(B\) if \(A\) has a prime factor and it is not a prime factor of \(B\). In this scheme, each node’s label is a multiplication of two parts: parent’s part and self part. Each node’s self-part is given a prime number. A node can only be divisible by its ancestors, so with this labelling scheme, the structural relationship can be determined easily.

2.2 XML Twig Pattern Matching

Over the last few years, many algorithms were proposed to perform twig pattern matching. Some work (e.g., [33], [34], [35], [31], [7]) first decomposes a twig pattern into a set of binary relationships (i.e., parent-child or ancestor-descendant relationships). Then the matches of these binary relationships are joined together. In [33], Zhang et al proposed a Multi-Predicate Merge Join (MPMGJN) algorithm which is different from
traditional merge join and the index nested-loop join algorithms and utilizes the inverted list of region codes (see Section 2.1). Al-Khalif et al [7] gave two structural join algorithm called tree-merge and stack-tree. In particular, stack-tree is optimal\footnote{Optimal means the worst-case I/O and CPU complexities are linear in the sum of the sizes of the input and output.} for a binary relationship. Different from the two algorithms above that are based on region-based labelling scheme, Wang et al [31] proposed a structural join algorithm which is based on PBiTree labelling scheme. The major problem of this binary structural join approach is that it may generate a large number of useless partial solutions and hence much time is wasted in the merging phase.

To overcome the problem above, Bruno et al [8] proposed a novel holistic twig pattern matching algorithm called TwigStack, which breaks the query tree into root-to-leaf paths, finds individual root-to-path solutions, and merges these partial solutions to get the final result. It is shown that when there are only //edges, every root-to-leaf path solutions returned by the algorithm will contribute to some final solutions. In addition, one vivid feature of TwigStack is the efficient filtering of useless elements through the use of function getNext. Because TwigStack acts as the basis of many other twig pattern matching algorithms, we will have a more detailed discussion about it in Chapter 3.

TwigStack may have redundant computation. The algorithm TJEssential [36] resolves the three deficiencies (i.e., self-nested suboptimal, order suboptimal and stream null suboptimal) in TwigStack that may bring redundant computation. We use an
2. RELATED WORKS

![Diagram](image1)

**Figure 2.4:** Example of TwigStack+

![Diagram](image2)

**Figure 2.5:** Example of sub-optimality of P-C edge

example to simply introduce these three deficiencies. In this example, the query in Figure 2.3(b) is submitted over the data tree in Figure 2.3(a). In the data tree, $a_1$, $a_2$, $a_3$ and $a_4$ are self-nested elements and there are no $b$, $c$, $d$ and $e$-elements between them. It is obvious that $a_2$, $a_3$ and $a_4$ will contribute to final solutions if $a_1$ does. However, after $a_1$ is pushed into the stack, TwigStack still makes calls of $getNext$ over the node $a$ (i.e., test whether $a_2$, $a_3$ and $a_4$ contribute to final solutions). These calls are redundant and this kind of cases are considered as self-nested suboptimal. For order suboptimal, sometimes, the order of calling $getNext$ over the child nodes will affect the total number of calls of $getNext$. For example, if the call of $getNext(b)$ is before $getNext(c)$, three redundant calls of $getNext(c)$ will be saved. However, TwigStack cannot dynamically adjust the calling order. For stream null suboptimal, it is observed that many calls of $getNext$ are redundant if some streams reach end. For instance, if $b_1$ is the last element in its corresponding stream, after $b_1$ is pushed into the stack, all of $d$-elements ($d_1$, $d_2$ and $d_3$) can be directly pushed into the corresponding stack without calling $getNext(a)$. TwigStack+ [37] reduces redundant computation by prevent $getNext$ from returning before a solution is found. Suppose the query in Figure 2.4(b) is submitted over the data tree in Figure 2.4(a). The third call of $getNext$ over the node $b$ will not return until $c_2$, $c_3$ and $c_4$ are skipped in $getNext$. 

TwigStack shows sub-optimality when $\neg$-edges appear in a twig pattern. For example, a query in Figure 2.5(b) is submitted over the data tree in Figure 2.5(a). There is
a /-edge in the query. TwigStack will output an intermediate path solution (a, b) even though this path solution does not exist in the final solution. In other words, TwigStack outputs a useless path solution. Several improvements of TwigStack were made to deal with /-edges. TwigStackList \[10\] introduces a simple buffering scheme to reduce the useless intermediate path solutions when /-edges exist. If /-edges do not appear under branching nodes, TwigStackList can guarantee no useless intermediate path solutions are produced. In the example above, TwigStackList will buffer all of the c elements that are the ancestors of e. Therefore, it will be easy to determine that the element e is useless and then skip it. This will prevent the path solution (a, b) from being returned. iTwigJoin \[9\] incorporates two streaming schemes, namely Tag + Level and Prefix Path Streaming (PPS), to enlarge the optimality of TwigStack. As we know, TwigStack is sub-optimal when parent-child edges exist. Via the re-organization of data streams, the elements that do not satisfy parent-child structural relationship will not be accessed, so the optimality can be maintained. However, Tag + Level and PPS cannot guarantee optimality in all cases. When just parent-child edges exist in a twig pattern, the optimality can be achieved. In BLAS \[38\], Chen et al proposed a bi-labelling scheme to improve the performance of processing /-edges. When there are only /-edges in the query, this method can achieve good performance. It should be noted that BLAS does not use holistic twig join strategy. \[39\] theoretically shows that when an /-edge is never followed by a /-edge downwards in a query, the algorithm can achieve time and space optimality.

Some algorithms make use of index structures to skip useless elements (e.g., \[40\], \[41\], \[11\], \[12\]). Chien et al \[40\] first proposed a structural join algorithm with B+-tree index, and introduced sibling pointers to improve the performance further. Then they use R-tree instead of B+-tree, and find that the structural join that utilizes B+-tree is more robust. TwigStack can achieve time and space optimality if only ancestor-descendant edges exist in a twig pattern. But according to the algorithm, every element in the stream is accessed, although some accesses are unnecessary. Therefore, \[8\] uses XB-tree to skip useless elements and presents a holistic algorithm TwigStackXB. TSGeneric+ \[11\] utilizes the index structure XR-Tree \[42\] to avoid such unnecessary accesses such that the I/O performance can be effectively improved. TwigOptimal \[12\] is another algorithm which uses B-tree and skips useless elements by forwarding nodes.
2. RELATED WORKS

...virtual pointers. According to [43], B+-tree, XB-tree and XR-tree achieve comparable performances when the data tree does not involve recursion. However, when the data tree is highly recursive, XB-tree yields the best performance.

All of the algorithms above are two-phase twig pattern matching algorithms. Chen et al [13] observed that the holistic two-phase algorithms still suffer from high merging costs, and they proposed a one-phase algorithm Twig2Stack, which avoids the merging phase by storing final solutions in hierarchical stacks. For a twig pattern \(Q(V, E)V = V_1, V_2, \ldots, V_n\), each query node \(V_i(i \leq i \leq n)\) has a hierarchical-stacks \(HS_{vi}\). \(HS_{vi}\) is a set of ordered stack trees \(ST_j(V_i)\). Each stack \(S_k(V_i)\) in a stack tree has zero or more elements of the node \(V_i\). The stacks in stack trees of different query node may be connected by pointers to present ancestor-descendant relationship. The process of constructing hierarchical-stacks is embedded in the post-order traversal of the XML data tree. It is claimed that Twig2Stack outperform TwigStack. Qin et al [44] proposed another one-phase algorithm, TwigList, which uses a much simpler data structure, a set of lists, to store the final solutions. Due to the simpler data structure and hence the reduction in random memory access, TwigList achieves better performance than Twig2Stack. Twig2Stack and TwigList can avoid the high cost of the merging phase, but they lose an important ability of the holistic approach, which is efficiently locating twig occurrences and discarding useless elements. More recently Jiang et al [45] proposed a one-phase holistic twig pattern matching algorithm called HolisticTwigStack, which maintains the overall solutions in linked stacks. However, a considerable amount of time is taken to maintain the linked stacks. Most recently, Grimsmo et al [46] proposes two optimal twig pattern algorithms TJStrictPre and TJStrictPost, but they still introduce much redundant computation because of more complicated data structures and lots of tests for /-edges.

In order to reduce the input data, virtual streams for internal nodes are proposed. The basic idea is that the positional information of a node can be inferred from its descendants. Virtual Cursors [47] implements virtual streams using DeweyID and path summaries. TJFast [48] is another algorithm uses the concept of virtual streams and proposes Extended Dewey labelling scheme to represent the position of the elements and accelerate the determination of structural relationships. Given an XML data tree and a DTD, the labels are produced via a module function. Each tag name is mapped to an integer using this module function, and then the self integer is concatenated
with the labels of its parent. Obviously, a label of an element also contains all of its ancestors’ labels (position information), so TJFast only accesses the data streams of the leaf nodes in a twig pattern. This significantly reduces the I/O complexity and improves the overall performance.

Some research focuses on order-based queries ([49], [50], [51]). Vagena et al [49] proposed holistic algorithms for twig queries with order-based forward and backward axes of the same type. In [50], they proposed an approach for handling positional predicates within XML query processing. More recently, Lu et al [51] provided an holistic algorithm called OrderedTJ for order-based twig queries. Their approach is based on the new concept of Ordered Children Extension (OCE).

In order to handle the twig queries with OR-predicates, Jiang et al [52] proposed a holistic algorithm which does not decompose a twig query with OR-predicates into several twig queries without OR-predicates. For the Not-predicates, Jiao et al [53] proposed a holistic algorithm to answer the path query with Not-predicates. However, it cannot process the twig query with Not-predicates. To solve this problem, Yu et al [54] proposed an algorithm called TwigStackList, which can guarantee I/O optimality when all the positive edges below branching nodes are ancestor-descendant relationships.

To avoid the join in previous twig pattern matching algorithms, some sequence matching algorithms were proposed ([55], [56], [57]). In ViST [55], both XML data tree and query tree are transformed to structure-encoded sequences. Then, the problem of twig pattern matching is converted to the problem of sequence matching. However, ViST may suffer from the problem of false alarm. Different from ViST, [56] transforms XML data tree and query tree into Prüfer sequences. Compared with structure-encoded sequences, Prüfer sequences consume less space for storing indexes and avoid false alarms. More recently, Wang et al [57] proposed a performance-oriented principle to guide the sequencing of tree structure.

2.3 XML Keyword Search

In this section, we review some important literature on XML keyword query processing.

Connecting keyword matches A lot of work has been done to connect the keyword matches in a meaningful way (e.g., [58], [59], [60], [61]). Most previously proposed
2. RELATED WORKS

XML keyword search systems use the concept of lowest common ancestor (LCA), or its variant, to connect the XML nodes which contain the keywords. XRank [58] proposed the excluding semantics (ELCA) to connect keyword matches, which connects keyword matches by the LCA nodes that contain at least one occurrence of all keywords after excluding the occurrences of keywords in their descendants that already contain all keywords. XKSearch [59] proposed the notion of Smallest LCA (SLCA) to connect keyword matches. A SLCA is the root of a subtree which contain all the keywords, and any subtree rooted at its descendants does not contain all the keywords. Because ELCA and SLCA are widely used in existing XML keyword search systems, we provide an example to illustrate the difference between them. Suppose the user submits a keyword query (b, c) over the data tree in Figure 2.6. The ELCA and SLCA are respectively \((a_1, a_2, a_3, a_5, a_6)\) and \((a_2, a_3, a_6)\). \(a_1\) and \(a_5\) are not SLCA because they have descendants (i.e., \(a_2\) and \(a_6\)) that contain all of the keywords. Li et al [60] proposed the concept of Meaningful LCA (MLCA) to connect keyword matches. A set of keyword matches are considered to be meaningfully related if every pair of the matches is meaningfully related. Two keyword matches are considered meaningfully related if they can be linked with a SLCA. The LCA-based approaches have a common inherent problem, which is that they may link irrelevant XML nodes together and return large amounts of useless information to the user. This problem is called false positive problem in [61]. In order to solve the false positive problem, Li et al [61] proposed the concept of valuable LCA (VLCA). Suppose there are two XML nodes which have the same label on the paths \(w\) to \(u\) and \(w\) to \(v\), \(u\) and \(v\) are considered to be Heterogenous and will not be linked together. Different from the LCA-based approaches, XSEarch [62] introduces the concept of interconnection to connected keyword match nodes. Two matches can be interconnected if there are no distinct nodes with the same tag name on the path between them.
Generating results} After keyword matches are meaningfully connected, it is important to determine the way of displaying the results. Most LCA-based approaches return the whole subtree rooted at the LCA or its variants (e.g., MLCA, SLCA, etc). Sometimes, the returned subtrees are too large for users to find the needed information. Therefore, \[63\] returns paths instead of subtrees. Hristidis et al in \[64\] introduced the concept of minimum connecting trees (MCTs) to exclude the subtrees rooted at the LCAs that do not contain keywords. This approach makes the results more compact. To make the answers displayed more meaningful, XSeek \[65\] tried to recognize the possible entities and attributes in the data tree, distinguish between search predicates and return specifications in the keywords, and return nodes based on the analysis of both XML data structures and keyword match patterns.

Relevancy and ranking} Due to the ambiguity of keywords, the returned answers are often irrelevant to users’ expectations. In order to make the results more relevant to the user, Liu et al \[66\] proposed an axiomatic framework, which includes two important properties that a search engine should satisfy: monotonicity and consistency. Based on this framework, they developed an algorithm MaxMatch and proposed a new concept contributor. A subtree under a result subtree will be considered as a contributor if it contains richer information (i.e., contains more specified keywords) than its siblings. MaxMatch can prune the subtrees that are not contributors. For example, the user submits a keyword query (b, d, e) over the data tree in Figure 2.7. Besides the subtrees rooted at a₁ and a₂ are considered as the results, the subtrees rooted at c₂ and c₃ will be pruned. This is because c₂ and c₃’s siblings (i.e., c₁ and c₄) contain more specified keywords than them. MaxMatch cannot filter out all the uninteresting nodes, so Kong et al proposed the concept of Relaxed Tightest Fragments (RTF) and valid contributor, which can resolve the problems of false positive and redundancy in MaxMatch. More recently, Bao et al \[1\] introduces relevance oriented ranking into XML keyword search.
2. RELATED WORKS

They first infer the search-for node type by computing the confidence score of a node type to be desired by the user. Then they utilize a novel XML TF*IDF ranking strategy to rank the result subtrees rooted at the search-for node type. In order to infer better search-for node type or result type, [67] proposed an estimation-based approach to compute the promising result types. According to the experimental results in [67], XBridge achieves better precision than XReal. Additionally, some approaches (e.g., [68], [69]) rank the results according to the distance between keyword match nodes. Termehchy et al [70], [71] proposed coherency ranking (CR), which is a database design-independent ranking method for XML keyword query processing. The results of query remain invariant when the schema is reorganized.

**Entity-based query processing** Xu et al [72] proposed to use minimum information unit (MIU) in place of nodes and return the lowest common ancestor MIUs. This can ensure that the results are informative. However, this still suffers from the problems of LCA, such as false positive. Similarly, Bao et al proposed object-level semantics called Interested Single Object (ISO) and Interested Related Objects (IRO) to answer keyword queries [73]. It should be noted that although XSeek also identify entity nodes, it does not treat entity as the smallest unit during processing.

**Keyword search over XML document with ID-References** Some research work focuses on finding matches from the XML document with ID-References which can be modelled as a directed graph. In [74], Chen et al proposed Tree+IDREF model to resolve the keyword search on the XML document with ID-References. To improve the effectiveness of keyword search, Zhou et al proposed an effective semantics called Meaningful Connected Network (MCN) [75]. He et al proposed BLINKS which introduces bi-level index to prune and accelerate the top-k keyword search over graphs [76]. In [77], Golenberg et al proposed a keyword search engine that does not miss relevant results, is efficient and generates the answers in an order that is highly correlated with the desired ranking. A framework for describing semantic relationships among nodes in the XML document with ID-Referenc es was presented in [78]. Within this framework, interconnection semantics is introduced.

**Snippet generation** Additionally, in order to help the user quickly assess the relevancy of the XML query results, Huang et al provide a snippet generation system called
eXtract in [79]. They proposed the concept of feature dominance and devised a formula to compute the dominance score of a feature to help generate snippets. Then, Liu et al [80] extends eXtract by introducing prominent features and the distribution of dominant features.

**Efficiency** The efficiency of SLCA evaluation was studied in [59], [81] and [82]. [81] proposed multiway-SLCA approach to evaluate keyword queries according to SLCA semantics. [82] proposed a hash-based approach to answer SLCA-based keyword queries. The efficiency of ELCA evaluation was studied in [83] and [84]. In [83], Xu et al proposed an algorithm called Indexed Stack to process keyword queries based on ELCA semantics. More recently, Zhou et al proposed an algorithm called Hash Count to compute ELCA nodes. The basic idea is to use hash table to record the number of keyword occurrences appearing under a node. Then it will be easy to judge whether a node dominates all of the keywords.

**Others** Lu et al [85] proposed a useful tool called XClean which suggests alternative queries when user’s queries contain spelling errors. Li et al studied top-k keyword search over probabilistic XML data and proposed two algorithms PrStack and EagerTopK in [86]. Query segmentation is the process of dividing a query into individual phrases (or concepts), which is widely used in web search engines to improve document-retrieval precision [87] [88]. There are mainly two types of approaches for query segmentation. The first one is based on the mutual information (MI) between pairs of keywords. If the MI value between two keywords is smaller than a threshold, a segment separator will be inserted at this position. This approach does not consider the relationship among more than two keywords and it heavily relies on the statistics of the corpus. The second approach uses supervised learning to help decide whether to insert a separator at a position, and considers the keywords within a small window at that position. In addition, query logs are frequently used in query segmentation techniques. Due to some similarity between query segmentation and our approach, we discuss the differences between them in Chapter 4.

Petkova et al [89] considers different meanings of a keyword. They proposed an algorithm which can transform a keyword query into a set of content-and-structure queries and then these queries are ranked according to information theoretic measure
2. RELATED WORKS

information gain (KL-divergence). Sometimes, for this approach, it is difficult to generate a complete set of possible queries with the transformation operators they provide, particularly when a query contains many keywords or the keywords have many different meanings. It is also not easy to always generate good structured queries. As a result, some important queries may be missing. This means the user will not get the desired results. In addition, the ranking scheme they use only considers the frequency of a keyword’s different meanings. Some important structural factors (e.g., the distance between keywords, etc) are not considered. Therefore, the desired results may obtain a very low rank. For example, if the user intends to retrieve the papers about book digitization from DBLP dataset and submits the query “book digitization”, their system will preferentially return books about digitization because most occurrences of “book” appear as the element name book.
3

XML Twig Pattern Matching

3.1 Introduction

In the past few years, many algorithms of twig pattern matching were proposed (e.g., [7], [8], [13], [44], [10], etc). In particular, the one-phase algorithm TwigList [44] accelerates Twig2Stack [13] by utilizing simple lists instead of hierarchical stacks. Due to the simpler data structure and hence the reduction in random memory access, TwigList achieves better performance than Twig2Stack. However, both TwigList and Twig2Stack still suffer from pushing/popping-up a large amount of useless elements into/out of the stack. Therefore, if we can find a solution to avoid these unnecessary stack operations and efficiently skip useless elements, the performance can be significantly improved. In our work, we use the core function getNext in TwigStack [8] to achieve this goal and modify the data structure for storing final solutions to completely avoid using the stack. Furthermore, we find that getNext may introduce much redundant computation. Therefore, we propose two approaches called re-test checking and forward-to-end to resolve this problem.

In this chapter, we first present two one-phase holistic algorithms TwigMix and TwigFast, which utilize getNext in TwigStack for efficiently filtering useless elements and introduce a new data structure for storing final solutions, in Section 3.3. Then we reduce redundant computation in previous algorithms in Section 3.4. Finally, we present the experimental results in Section 3.5.
3. XML TWIG PATTERN MATCHING

Algorithm 1 getNext(q) [8]
1: if (isLeaf(q)) return q
2: for qi ∈ children(q) do
3:    ni = getNext(qi)
4:    if (ni ≠ qi) return ni
5: nmin = minarg ni, nextL(Tni)
6: nmax = maxarg ni, nextL(Tni)
7: while (nextR(Tni) < nextL(Tnmax)) do
8:    Advance(Tni)
9: if (nextL(Tni) < nextL(Tnmin)) return q else return nmin

3.2 Background and Notation

3.2.1 Notation

Below, we will use elements to refer to nodes in a data tree, and nodes to refer to nodes in a twig pattern. We will also use x-child (resp. x-descendant, x-element) to refer to a child (resp. descendant, element) labeled x. As in TwigStack, for each node n, there is a stream, Tn, consisting of all elements with the same label as n arranged in ascending order of their start values in the region codes. Note that an element may appear in several streams if there are nodes with identical labels in Q. For each stream Tn, there exists a pointer PTn pointing to the current element in Tn. The function Advance(Tn) moves the pointer PTn to the next element in Tn. The function getElement(Tn) retrieves the current element of Tn. The function isEnd(Tn) judges whether PTn points to the position after the last element in Tn. In addition, for node n, the functions isRoot(n) (resp. isLeaf(n)) checks whether node n is the root (resp. leaf), and parent(n) (resp. children(n)) returns the parent (resp. set of children) of n.

3.2.2 TwigStack and TwigList

To facilitate our explanation, we briefly recall the major features of TwigStack and TwigList here.

As mentioned earlier, TwigStack uses a function getNext(q) to efficiently filter useless elements. For self-containment, we copy the function into Algorithm 1. In the function, nextL(Tn) and nextR(Tn) return getElement(Tn).start and getElement(Tn).end respectively. The function has the following properties: if q is root(Q) (the root of Q), then getNext(q) always returns a node n that has a minimal descendant extension, i.e., (1) for each child n’ of n, the current element of Tn has a descendant which is
the current element of $T_{n'}$, and each child of $n$ recursively has this property; (2) the current element of $n$ has the minimum start value among all nodes that have property (1). The function also moves the pointer $PT(T_{n_i})$ when the current element in $T_{n_i}$ no longer has descendants in $T_{n_j}$, for some of child $n_j$ of $n_i$ (lines 7,8).

**TwigList** is based on the following observation [44]: for each $a$-element $v$, its $b$-descendants can be arranged in a minimal interval, such that every $b$-descendant of $v$ falls into this interval, and $b$-elements that are not descendants of $v$ do not fall into the interval. As a consequence, we can use a pair of position values, $v_{\text{start}_b}$ and $v_{\text{end}_b}$, to specify the interval for all $b$-descendants of $v$. For example, for the data tree shown in Figure 3.1 (a), all descendants of the $a$-nodes can be arranged in a list $b_1, b_2, b_3, b_5, b_4$, and $a_{1\text{start}_b} = a_{1\text{end}_b} = 1$, $a_{2\text{start}_b} = 2$, $a_{2\text{end}_b} = 3$, $a_{3\text{start}_b} = 4$ and $a_{3\text{end}_b} = 5$ will tell us the $b$-descendants of each $a$-element. The data structure used in **TwigList** is thus a set of lists, one list, $L_n$, for each node $n$ in $Q$. Each element $v$ in $L_n$ has pairs of start and end pointers pointing to the start and end positions of descendant intervals (one interval for each child of $n$). These lists are used to store the final solutions. For instance, for the data tree and query in Figure 3.1 (a), (b), the lists built by **TwigList** are shown in Figure 3.1 (d). In the figure, $a_1$, $a_2$ are not put into list $L_n$ because they do not have $c$-descendants. The main algorithm of **TwigList** is a procedure to construct the lists, once this is done, it uses another procedure **TwigList-Enumerate** to efficiently enumerate the final solutions. To construct the lists, **TwigList** uses a stack, $S$. Elements are pushed into the stack in pre-order, and $\text{top}(S)$ is popped up when a non-descendant of $\text{top}(S)$ arrives, and it is then checked to see whether it should be appended to the corresponding list.

### 3.3 Accelerating One-phase Twig Pattern Matching Algorithm

In this section, we present two one-phase holistic twig matching algorithms, **TwigMix** and **TwigFast**, which combine the efficient selection of useful elements introduced in [8] with the simple data structure for storing final solutions introduced in [44]. **TwigMix** simply introduces the getNext function of **TwigStack** into **TwigList** to avoid manipulation of useless elements in the stack and lists. **TwigFast** further improves this by introducing some pointers in the lists to completely avoid the use of stacks, based on the
3. XML TWIG PATTERN MATCHING

![Diagram of XML twig patterns](image)

Figure 3.1: An example to explain the basic ideas of TwigMix

observation that the overhead of maintaining the pointers is generally negligible compared with the pushing/popping-up of elements into/from the stack. We conducted extensive experiments with both real and synthetic data. Our experiments show that (1) TwigMix significantly and consistently outperforms TwigList and HolisticTwigStack (up to several times faster), and TwigFast performs even better (up to two times faster) than TwigMix; (2) compared with TwigList, TwigMix saves an average of 75.93% of elements from being pushed into stack and an average of 70.19% of elements from being appended into the result lists. Since the result lists built by our algorithms are far shorter than those built by TwigList, our algorithms relieve the problem of memory consumption.

3.3.1 TwigMix: Introducing Efficient Element Filtering into TwigList

3.3.1.1 Overview of TwigMix

We explain the basic ideas used in TwigMix using the example in Figure 3.1. TwigMix uses the same data structure as TwigList, but it introduces the getNext() function to avoid pushing useless elements into the stack \( S \) and appending useless elements into the lists. In Figure 3.1 if we apply the TwigList algorithm, all of the elements will be pushed into \( S \). When the elements are popped up from the stack, the algorithm will determine whether to append them to the result lists. For this example, \( a_1 \) and \( a_2 \) are not appended to the result lists because they cannot find their \( c \)-descendants. However, \( b_1, b_2 \) and \( b_3 \) are still appended to the result list although they do not contribute to the final solutions. Figure 3.1(d) shows the structure of the final lists constructed by TwigList. For TwigMix, due to the introduction of getNext(), \( a_1 \) and \( a_2 \) can be directly abandoned and will not be pushed into \( S \). The elements \( b_1, b_2, b_3 \) will not be pushed into \( S \) either because they cannot find their ancestors in \( S \). The final result lists are
3.3 Accelerating One-phase Twig Pattern Matching Algorithm

Algorithm 2 TwigMix-Construct(Q)

1: initialize stack \( S \) as empty;
2: initialize the list \( L_{n_j} \) as empty, \( n_j.counter \) as 0, for all nodes \( n_j \in Nodes(Q) \);
3: while \( \neg \text{end}(Q) \) do
4: \( n_{\text{act}} = \text{getNext}(\text{root}(Q)) \)
5: \( v_{\text{act}} = \text{getElement}(n_{\text{act}}) \)
6: toList(S, region(\( v_{\text{act}} \))) // region(\( v \)) denotes the interval \( (v.start, v.end) \)
7: if \( \text{isRoot}(n_{\text{act}}) \) OR parent(\( n_{\text{act}} \)).counter > 0 then
8: for \( n_k \in \text{children}_{n_{\text{act}}} \) do
9: \( v_{\text{act}}.start_n_k = \text{length}(L_{n_k}) + 1 \)
10: push(S, \( v_{\text{act}} \))
11: \( n_{\text{act}}.counter + + \)
12: Advance(T_{n_{\text{act}}})
13: toList(S, (\( \infty, \infty \)))
14: procedure end(q)
15: return \( \forall n_i \in Nodes(q) : \text{isLeaf}(n_i) \Rightarrow \text{isEnd}(T_{n_i}) \)
16: procedure toList(S, r)
17: while \( S \neq \emptyset \) AND \( r \not\in \text{reg}(\text{top}(S)) \) do
18: \( v_j = \text{pop}(S) \)
19: let \( v_j \)'s type be \( n_j \) // the type \( n_j \) is memorized when \( v_j \) is pushed into \( S \)
20: \( n_j.counter - - \)
21: for \( n_k \in \text{children}_{n_{\text{act}}} \) do
22: \( v_j.end_n_k = \text{length}(L_{n_k}) \)
23: append \( v_j \) into list \( L_{n_j} \)

shown in Figure 3.1(c). Therefore, TwigMix does not waste time in pushing/popping-up \( b_1 \), \( b_2 \), and \( b_3 \) into/from stack and appending them to result list \( L_b \). It also saves memory because \( b_1 \), \( b_2 \) and \( b_3 \) do not need to be stored in the lists. If the data tree is large, the savings of time and space will be quite significant (see Section 3.5 for examples).

3.3.1.2 TwigMix

TwigMix differs from TwigList in its way of constructing the final result lists. Once the lists are constructed, it uses the same procedure TwigList-Enumerate in [44] to enumerate all final solutions.

Our new algorithm for building the result lists, TwigMix-Construct, is shown in Algorithm 2. Like TwigList, we use a stack \( S \) to achieve bottom-up processing of elements. For each node \( n_i \in Nodes(Q) \), we use a counter \( n_i.counter \) to record the number of elements in stack \( S \) for that query node. In Algorithm 2 after initialization, the function \( \text{getNext}(q) \) is repeatedly called (lines 3,4) to get the query node which
has a minimal descendant extension (see Section 3.2.2). The loop will stop until there are no elements not processed for any of the leaf nodes (see the end(q) function). Line 7 is particularly important. If the returned query node $n_{act}$ is the root, its current element is directly pushed into the stack $S$. However, if it is not the root, the counter of $parent(n_{act})$ is checked to see whether any elements of $parent(n_{act})$ are in the stack. We push the current element of $T_{n_{act}}$ into $S$ only when there are elements of $parent(n_{act})$ in $S$ (this is why the elements $b_1$, $b_2$ and $b_3$ in Figure 3.1(a) are not pushed into stack). The counters are maintained at line 11 and line 20, when an element is pushed into or popped up from $S$. When an element is pushed into $S$, the start positions of its descendant intervals are set (lines 8,9). In the sub-procedure $toList(S,r)$, we check whether the current element in the node returned by $getNext(root(Q))$ is a descendant of $top(S)$, if not, we pop up $top(S)$, set the end positions of its descendant intervals, and append it directly to the corresponding list. Note that, unlike the procedure in TwigList, we do not need to check whether $top(S)$ can be appended to list because all elements pushed into the stack are guaranteed to appear in some final solution (provided $Q$ has no /-edges). At the end of the algorithm, we apply an infinite interval to $toList$ in order to pop up all elements from $S$.

**Example** Consider the twig pattern and the data tree in Figure 3.1. Initially, the current elements of the query nodes are $(a_1, b_1, c_1)$. All the first three calls of $getNext(a)$ return node $b$. Because the counter of $b$’s parent $a$ is 0, the elements $b_1, b_2, b_3$ are not pushed into the stack $S$. The fourth call of $getNext(a)$ returns node $a$. Node $a$ is the root of the query tree, so $a_3$ is directly pushed into $S$ and the start positions of its descendant intervals are recorded. The counter of node $a$ increases by 1. The next two calls of $getNext(a)$ return node $b$. Because the counter of node $a$ is 1, the elements $b_5, b_4$ are pushed into the S stack. Next, $getNext(a)$ returns node $c$. The coming of $c_1$ results in $b_5$ and $b_4$ being popped up and appended to $L_b$. Finally, the range $(\infty, \infty)$ makes $c_1$ and $a_3$ pop up and they are appended to $L_c$ and $L_a$ respectively. When $a_3$ is appended to $L_a$, the end positions of its descendant intervals are recorded. ⊥

### 3.3.1.3 Analysis of TwigMix

In this section, we show the correctness of TwigMix and analyze its time and space complexity. We prove the following lemma first.
Lemma 1 Suppose $Q$ has no $/$-edges. TwigMix pushes an element into stack $S$ iff the element contributes to some final solutions.

Proof [sketch] (only if) If $\text{getNext}(\text{root}(Q))$ returns $q_{\text{act}}$, then $q_{\text{act}}$ has a minimal descendant extension (see Section 3.2.2). Therefore, the current element $v_{\text{act}}$ of $T_{q_{\text{act}}}$ (line 5) has a descendant in $T_{n_i}$ for each child $n_i$ of $q_{\text{act}}$. Line 7 and line 10 make sure that only if $q_N$ is $\text{root}(Q)$ or $S$ contains an element of type $\text{parent}(q_{\text{act}})$ do we push $v_{\text{act}}$ into $S$. In both cases, $v_{\text{act}}$ participates in at least one final solution, since we assume there are only $$/$-edges in $Q$.

(if) If an element $v$ of type $n$ participates in some final solution, $\text{getNext}(\text{root}(Q))$ will return $n$ when the current element of $T_n$ is $v$. If $n$ is the root, $v$ will be pushed into $S$ directly. Otherwise, the stack $S$ will contain at least one element of type $\text{parent}(n)$ when $v$ is returned in line 5, because elements are pushed into stack in pre-order, and an element will be popped up from $S$ after its descendants have been popped-up (line 17-18). Hence $v$ will also be pushed into $S$. \(\square\)

Theorem 1 Given a twig pattern (that has $$/$-edges only) and an XML data tree, TwigMix correctly builds up the final result lists.

Proof [sketch] We only need to show that (1) elements contributing to final solutions will be appended to the lists, and (2) for each element in the list, its descendant intervals are correctly set. (1) is true because of Lemma 1 and the fact that every element pushed in the stack $S$ is appended in the result list. (2) is true because, for any element $v_{\text{act}}$ satisfying the condition in line 7, it is pushed into $S$ before any of its descendants are pushed into $S$. Therefore, the lists of the children nodes of $n_{\text{act}}$ has no descendants of $v_{\text{act}}$ before $v_{\text{act}}$ is pushed into $S$ at line 10. However, after $v_{\text{act}}$ is pushed into $S$, the next element in $T_{n_i}$ pushed into $S$ for any child $n_i$ of $n$ must be a descendant of $v_{\text{act}}$. Therefore, line 9 correctly sets the start positions of the descendant intervals for $v_{\text{act}}$. Furthermore, $v_j$ is popped up from $S$ only when all of its descendants have been popped up and appended to lists. Therefore, line 21-22 correctly sets the end positions of $v_j$’s descendant intervals. \(\square\)

Complexity analysis Algorithm 2 scans each stream $T_n$ from start to end once, through the functions $\text{getNext}()$ and $\text{Advance}(T_{n_{\text{act}}})$ at line 12. For each element in $T_n$ it may push it into stack, pop it up from stack, append it to list, and set its start
and end positions for its descendants. Suppose $d$ is the maximum degree of nodes in $Q$. For each element appended to result lists, at most $d$ intervals need to be recorded and recording an interval needs constant time. Pushing/popping-up an element into/from the stack $S$ can be finished in constant time. Therefore, the worst-case time complexity is $O(d \cdot N)$ ($N$ is the sum of the sizes of the input streams), which is linear in $N$. The worst-case space complexity is linear in the sum of the sizes of the occurrences of the twig pattern (the sum of the sizes of the final lists).

**Considerations of */-edges*:** The $get\text{Next}(q)$ function does not guarantee the returned node can be expanded to a solution when */-edges* exist. Therefore, Algorithm 2 does not guarantee all of the elements moved into the stack $S$ and result lists will appear in final solutions when */-edges* exist. To make sure the final results enumerated are still correct, we need to modify the enumeration algorithm so that it checks the satisfaction of parent-child relationship, for */-edges*, when outputting final solutions.

To improve the efficiency of enumeration, one can use the strategy of adding sibling links as in [44]. This strategy cannot prevent useless elements from being pushed into the stack $S$ and appended into the result lists. To reduce the manipulation of useless elements, we can incorporate the $get\text{Next}(q)$ function of algorithms that try to reduce the useless intermediate path solutions when */-edges* exist (e.g. TwigStackList [10], iTwigJoin [9], etc). However, these algorithms may result in the elements of the query nodes returned by $get\text{Next}(q)$ are not in pre-order. Therefore, TwigMix-Construct needs to be adjusted.
3.3 Accelerating One-phase Twig Pattern Matching Algorithm

3.3.2 TwigFast: Avoiding Manipulation of Elements in Stacks

3.3.2.1 Limitations of TwigMix

TwigMix integrates the holistic approach into TwigList, so only potentially useful elements are pushed into stack \( S \) and result lists. The time taken by pushing/popping-up elements into/from stack will become significant for large data trees. In order to get a glimpse of the number of elements that pass through \( S \), we implemented TwigMix and did some experiments over the DBLP data set. The selected queries are listed in Table 3.1. As shown in the table, for all three queries, the number of elements pushed into \( S \) is very large. Therefore, if we can directly build up the final lists without using the stack, the performance can be significantly improved.

<table>
<thead>
<tr>
<th>Query</th>
<th>Number of elements pushed into S</th>
</tr>
</thead>
<tbody>
<tr>
<td>//dblp//inproceedings[/title]/author</td>
<td>915,856</td>
</tr>
<tr>
<td>//dblp//article[/author][/title]/year</td>
<td>553,062</td>
</tr>
<tr>
<td>//dblp//inproceedings[/cite][/title]/author</td>
<td>149,015</td>
</tr>
</tbody>
</table>

Table 3.1: Limitation of TwigMix

3.3.2.2 TwigFast

TwigFast uses a data structure that is essentially the same as that of TwigMix, but to avoid the use of stack \( S \), it adds some pointers in the lists. More specifically, each element appended to the result list has a pointer, \( cancestor \), that points to its closest ancestor in the same list. With these pointers, the elements on the same path can be linked together. For example, in Figure 3.2(f), the element \( a_3 \) has a pointer pointing to its closest ancestor \( a_1 \). For each result list, a \( tail \) pointer is also maintained to point to the last element that still has potential descendants in the future. Together with the pointers that point to closest ancestors, we can easily maintain a list of elements which still have potential descendants, and these elements must be on the same path. For example, in Figure 3.2(f), with the pointers, we can easily find \( a_3 \) and \( a_1 \) still have potential descendants, but \( a_2 \) will not contribute to any new solutions in the future, so it is skipped by the pointer.

The purpose of the \( cancestor \) and \( tail \) pointers is to make it possible to correctly set descendant intervals for each element. When an element \( e \) is about to be appended
Algorithm 3 TwigFast(Q)
1: initialize the list $L_n$, as empty, and set $n_i.tail = 0$, for all $n_i \in Nodes(Q)$;
2: while $\neg$end(Q) do
3: $n_{act} = \text{getNext(root(Q))}$
4: $v_{act} = \text{getElement}(n_{act})$
5: if $\neg$isRoot($n_{act}$) then
6: SetEndPointers(parent($n_{act}$), $v_{act}.start$)
7: if isRoot($n_{act}$) ∨ parent($n_{act}$).tail ≠ 0 then
8: if $\neg$isLeaf($n_{act}$) then
9: SetEndPointers($n_{act}$, $v_{act}.start$)
10: for $n_k \in \text{children}(n_{act})$ do
11: $v_{act}.start_{n_k} = length(L_{n_k}) + 1$
12: $v_{act}.cancestor = n_{act}.tail$
13: $n_{act}.tail = length(L_{n_{act}}) + 1$
14: append $v_{act}$ into list $L_{n_{act}}$
15: Advance($T_{n_{act}}$)
16: SetRestEndPointers(Q, ∞)
17: procedure SetEndPointers($n$, actL)
18: while $n$.tail ≠ 0 do
19: $v_n = \text{element}(n$.tail$)$
20: if $v_n.end < actL$ then
21: for $n_k \in \text{children}(n)$ do
22: $v_n.end_{n_k} = length(L_{n_k})$
23: $n$.tail = $v_n$.cancestor
24: else
25: break
26: procedure SetRestEndPointers($n$, actL)
27: if $\neg$isLeaf($n$) then
28: SetEndPointers($n$, actL)
29: for $q_i$ in children($n$) do
30: SetRestEndPointers($n_i$, actL)

to $L_E$, the start positions of intervals are determined (line 10 to 11). For each child $C_i$ of query node $E$, the start position is equal to $\text{length}(L_{C_i}) + 1$. The end positions of an element can be determined when the element will not have any new descendants coming in the future (line 9). For each child $C_i$ of query node $E$, the end position is equal to $\text{length}(L_{C_i})$. For example in Figure 3.2(f), the coming of $a_3$ indicates $a_2$ will not have any new descendants in the future, so the end positions of $a_2$ are determined.

Example Consider the data tree and twig pattern shown in Figure 3.2. The first call of $\text{getNext()}$ returns $a$, with $a_1$ being the current element ($v_{act}$) of $T_a$. Since $a$ is the root of $Q$, and $a$ is not the leaf, the procedure $\text{SetEndPointers}(a, v_{act}.start)$ is called but it does nothing since $a.tail = 0$. Now the start positions of $a_1$’s descendant intervals are
set to 1, and $a1.ancestor = 0$, $a.tail = 1$, and $a1$ is appended to list $L_a$, and current element of $T_a$ is set to $a2$ (Figure 3.2 (c)). The second call of $getNext$ also returns $a$, and $SetEndPointsers(a, a2.start)$ is called. Since $a.tail \neq 0$, and $a1.end \geq a2.start$ (i.e., $a2$ is a descendant of $a1$), the procedure finishes with nothing done. Now lines 10 to 15 sets the start positions of $a2$’s descendant intervals as 1, and $a2.ancestor = 1$, $a.tail = 2$, appends $a2$ to $L_a$ (Figure 3.2 (d)), and advances $T_a$ to $a3$. The next call of $getNext()$ returns $b$, which is a leaf node. The current element of $T_b$ is $b1$. Therefore, $SetEndPointsers(a, b1.start)$ is called. Since $a.tail \neq 0$, and $a2.end > b1.start$, we set $c1$ to $c2$, and make $PT(T_c)$ point to $c2$ (Figure 3.2 (e)). The next call of $getNext()$ returns $a$ with $v_{act} = a3$. $SetEndPointsers(a, a3.start)$ is called. Since $a2.end < a3.start$, i.e., $a2$ no longer has $b$-descendants or $c$-descendants, we set the end positions of $a2$ as 1 and 1 for $b$ and $c$. We then set the start positions of $a3$ as 2 and 2, $a3.ancestor = 1$ (pointing to $a1$), $a.tail = 3$, append $a3$ to $L_a$ and advance $T_a$ (Figure 3.2 (f)). The next two calls of $getNext()$ return $b$ and $c$ respectively, so we append $b2$ and $c2$ to $L_b$ and $L_c$ respectively, and advance $T_b$ and $T_c$ (Figure 3.2 (f)). Now we use the infinite value to set the remaining end positions. That is, the end positions of $a3$ to 2. The final lists are shown in Figure 3.2 (h).

Correctness and complexity Both the correctness of TwigFast and the linear time and space complexity of Algorithm 3 can be established, in a way similar to TwigMix.

Considerations of /-edges For TwigFast, the strategy of adding sibling links [44] can also be applied. But one thing should be noted. TwigFast directly builds up the final solutions into result lists, so ancestors are always appended to result lists before their descendants. Therefore, when we set end pointers for an element, if it cannot find its children, it should be marked as useless. The enumeration algorithm will skip this element.

3.3.3 Summary

In this section, we presented two novel one-phase twig pattern matching algorithms that efficiently find twig pattern occurrences. TwigMix introduces holistic ideas into the original bottom-up approach, such that the elements that do not contribute to final
3. XML TWIG PATTERN MATCHING

Figure 3.3: example data tree t and tree pattern Q

solutions are not moved into the stack and result lists. TwigFast directly builds up final solutions without pushing/popping-up elements into/from the stack. The better overall performance of our algorithms has been substantiated in our experiments (see Section 3.5). Since the result lists built by our algorithms are far shorter than those built by TwigList, our algorithms relieve the problem of memory consumption.

3.4 Reducing Redundant Computation in Twig Pattern Matching

In Section 3.3, we use the function getNext to efficiently skip useless elements. Many one-phase and two-phase twig pattern matching algorithms also use getNext as a core function for filtering (e.g., HolisticTwigStack [15], TwigStackList [10], etc). However, getNext may incur other redundant computation (see Section 3.4.1 for details). Li et al [36] try to resolve the redundant computation and propose TJEssential, but their approach involves much overhead and cannot avoid some important types of redundant computation. Therefore, we propose a different approach to reduce redundant computation, and this approach imposes less overheads and can be easily applied to both holistic one-phase and two-phase twig pattern matching algorithms that are based on TwigStack. We present the algorithms TwigFast* and TwigStack* which extend TwigFast (one-phase) and TwigStack (two-phase) respectively by applying our proposed approach.

3.4.1 Deficiencies in Previous Algorithms

In this section, we explain the redundant computation of getNext. For ease of understanding, we use the query and data tree in Figure 3.3 to exemplify the redundant
3.4 Reducing Redundant Computation in Twig Pattern Matching

<table>
<thead>
<tr>
<th>Step</th>
<th>getNext(a)</th>
<th>getNext(b)</th>
<th>getNext(c)</th>
<th>getNext(d)</th>
<th>getNext(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a(a_1)</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>a(a_2)</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>a(a_3)</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>4</td>
<td>d(d_1)</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>e(e_1)</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>e(e_2)</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>7</td>
<td>e(e_3)</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>8</td>
<td>e(e_4)</td>
<td>b</td>
<td>c</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>9</td>
<td>b(b_1)</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>c(c_1)</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c(c_2)</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>b(b_2)</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>13</td>
<td>c(c_3)</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Example of redundant calls of getNext

Computation of getNext. We present each step of calling getNext over the root of the query tree (i.e., a) in Table 3.2. As shown in this table, each step includes the recursive calls of getNext over the nodes in the query tree, and the return node of each recursive call is listed. The recursive calls in each step are listed in the order of being called (i.e., the pre-order of the query nodes). In the second column, besides the return node of getNext(a), we also list the current element of this return node.

Basically, the redundant computation mainly comes from the following redundant test and late end.

**Redundant test** redundant test is making redundant calls of getNext over some nodes in the query tree. The current elements of these nodes did not change in the previous step.

Consider the data tree t and query Q in Figure 3.3, a_1-a_3 are self-nested nodes. After we found a_1 has a solution extension in step 1, it is unnecessary to call getNext over the query trees rooted at the nodes b and d when testing whether a_2 has a solution extension. This is mainly because the current elements of the nodes b, c, d and e do not change during and after step 1. We just need to check whether a_2 dominates the current elements of the nodes b and c (i.e., b_1 and c_1). This also happens on a_3 when checking if a_3 has a solution extension. Therefore, the calls of getNext over the nodes b, c, d and e in step 2-3 are redundant, and they are grayed in Table 3.2. getNext(a) returns node a in step 3 as shown in Table 3.2. This means only the current element of

---

4In this paper, a step is a call of getNext over the root of a query tree including the recursive calls.
a will be changed (i.e., the pointer $PT_a$ of stream $T_a$ will be forwarded) and the current elements of the other nodes will keep unchanged. Therefore, it is unnecessary to call \texttt{getNext} over the nodes $b$, $c$, $d$ and $e$ in step 4. This also happens in step 9 and 12, and the calls of \texttt{getNext} over $d$ and $e$ are redundant. Furthermore, during step 5-7, the calls of \texttt{getNext} over the subtree rooted at node $b$ are redundant because the current elements of the nodes in the subtree rooted at $b$ do not change.

**Late end** is wasting time on the elements that will not contribute to any solution when some cursors of the streams reach end.

Suppose there are no elements to be processed in the stream of node $q$, it is possible to skip all the rest of the elements in the streams of nodes \texttt{ancestors($q$)} and \texttt{descendants($q$)}. This can avoid some calls of \texttt{getNext} and the time spent on scanning the elements in some streams. For the example above, when there are no elements left in the stream $T_d$ after step 4, we can directly set $PT_a$ to the end because the remaining elements in stream $T_a$ will not contribute to any solutions. Then, when we found the rest of the elements in the streams of \texttt{descendant($q$)} will not contribute any solutions, we can set the $PT$ pointers of these streams to the ends. In Table 3.2, calls of \texttt{getNext} in step 8 and 13 are redundant and can be pruned.

### 3.4.2 Approach for Avoiding Redundant Computation

In this section, we explain our way of avoiding redundant computation, namely \texttt{re-test checking} and \texttt{forward-to-end}.

#### 3.4.2.1 Re-test checking

Redundant test is calling \texttt{getNext} over the nodes that have been processed by \texttt{getNext} before and there is no need to do that again in the current step. For example, in Table 3.2 there is no need to call \texttt{getNext} over the nodes $b$, $c$, $d$ and $e$ in steps 2-3 because \texttt{getNext} has been called over these nodes before and the current elements of them do not change after the calls. It is also unnecessary to call \texttt{getNext} over the nodes $b$ and $c$ in steps 5-7 for the same reason. Therefore, the key point for solving this problem is to determine whether it is necessary to call \texttt{getNext} over a query node again in the current step. Our solution for this problem is called \texttt{re-test checking} and is mainly based on the following observation:
3.4 Reducing Redundant Computation in Twig Pattern Matching

Algorithm 4 getNext*(q)

1: if isLeaf(q) then
2: return q
3: for qi ∈ children(q) do
4: if qi.retest = true then
5: ni = getNext*(qi)
6: if ni ≠ qi then
7: qi.retest = true
8: return ni
9: nmin = min argqj∈children(q) nextL(Tqj)
10: nmax = max argqj∈children(q) nextL(Tqj)
11: while (nextR(Tq) < nextL(Tnmax)) do
12: Advance(Tq)
13: if nextL(Tq) < nextL(Tnmin) then
14: q.retest = false
15: return q
16: else
17: nmin.retest = true
18: return nmin

Observation getNext(n) is used for testing whether a solution extension can be found for the current element of node n. Suppose getNext has been called over the node n before. If the current element of any node in the query tree rooted at n changes, it is necessary to call getNext(n) again for re-testing. Otherwise, getNext(n) does not need to be called.

Consider a query tree rooted at Q. In a step, if the return node of getNext(Q) is n, a solution extension for the current element of node n has been found and the current element of n will be changed to the next one in stream Tn. According to the observation above, we need to call getNext(n) again in the next step to test whether another solution extension can be found for the new current element of node n. Besides the node n, we also need to call getNext over the ancestors of node n again to test whether solution extensions can be found for these nodes because the current element of one of their descendants (i.e., n) has changed.

To solve the problem, we introduce an extra value retest for each query node to record whether getNext need to be called on this node in the next step. The initial value of retest is true which means all the nodes need to be tested by getNext, and this value is dynamic during computation.

The new version of getNext is presented in Algorithm 4. As shown in this algorithm, getNext*(q) will first check whether qi needs a re-test before calling getNext*(qi) (line
Algorithm 5 Forward-to-end

1: procedure ForwardAnstoEnd(n)
2:   for each p in ancestors(n) do
3:     ForwardtoEnd(T_p)
4: procedure ForwardDestoEnd(n)
5:   for each d in descendants(n) do
6:     ForwardtoEnd(T_d)

4). If a re-test is needed (i.e., q_i.retest = true), getNext*(q_i) will be called (line 5). If the return value of getNext*(q_i) (i.e., n_i) is different from q_i, this means we have found that n_i will need a re-test in the next step. Therefore, q_i will also need a re-test in the next step because q_i is an ancestor of n_i, and q_i.retest is set to true at line 7. If a solution extension is not found for the current element of q, n_min.retest will be set to true (line 17. Note: The return of n_min will result in early return of getNext* over n_min’s ancestors.) Otherwise, q.retest is set to false (line 14). Note that the value of q.retest may change at line 17 in the call of getNext* over q’s parent.

getNext* has the following properties:

1. Given a query rooted at Q, getNext* is only called over the nodes whose value of retest is true, including the nodes that have not been tested by getNext* before and the nodes have been tested by getNext* but need to be tested again.

2. Suppose getNext* has been called over each node at least once. If getNext*(Q) returns a node n in a step, getNext* will only be called over the nodes on the path from Q to n in the next step. The number of times getNext* is called will be bounded by the maximal depth of the query tree in the following steps.

With the properties above, the number of times getNext* is called can be significantly reduced, particularly when the query tree has many branches. For example, in Table 3.2 the unnecessary calls of getNext in the steps 2-4, 5-7, 9 and 12 can all be avoided with getNext*. Note that TJEssential cannot avoid the redundant computation in the steps 4, 9 and 12.

3.4.2.2 Forward-to-end

When the pointer PT_n of the stream T_n reaches the end, the rest of the elements in the streams of n’s ancestors and descendants may become useless because they will not
3.4 Reducing Redundant Computation in Twig Pattern Matching

contribute to any final solution. The symptom of late end is wasting time on processing these useless elements. Therefore, we need to find a solution to skip these useless elements. The key for solving this problem is to choose an appropriate time point to skip them because they do not immediately become useless and some of them may still contribute to final solutions. In our approach forward-to-end, there are two time points that are suitable for skipping. Consider a query tree rooted at $Q$. Suppose $\text{getNext}(Q)$ returns node $n$ in a step, and $PT_n$ reaches the end after calling $\text{Advance}(T_n)$. The two time points for skipping the rest of the elements in the streams of $n$’s ancestors and descendants are as follows:

**Time point 1** We immediately skip the rest of the elements in the streams of $n$’s ancestors after calling $\text{Advance}(T_n)$ because we cannot find any solution extension for them in the following steps.

**Time point 2** We cannot immediately skip the rest of the elements in the streams of $n$’s descendants after calling $\text{Advance}(T_n)$ because they are still potential elements that may contribute to final solutions. We have to wait until there are no elements in the stack $S_n$ for the two-phase algorithms that use stacks for storing intermediate results and all the end positions in the list $L_n$ have been set for the one-phase algorithms that use lists for storing final results.

The pseudocode of skipping the rest of the elements in the streams of the node $n$ and $n$’s ancestors and descendants is shown in Algorithm 5. Note that the procedure $\text{ForwardToEnd}(T_n)$ at line 3 and 6 directly forwards the pointer $PT_n$ of $T_n$ to the end in constant time.

**Influence to the correctness of re-test checking** Suppose $\text{getNext}^*(Q)$ returns node $n$ in a step, where $Q$ is the root of the query tree. Besides the current element of $n$, the current elements of $n$’s ancestors and descendants may also change after applying the approach above to skip the useless elements. This may influence the correctness of re-test checking in Section 3.4.2.1 because it assumes only the current element of $n$ changes after $\text{getNext}^*(Q)$ returns. However, if we consider that all the rest of the elements in the streams of $n$’s ancestors and descendants are discarded, we will find that the correctness of re-test checking is still guaranteed. The reasons are as follows:

1. For time point 1, the current elements of $n$ and $n$’s ancestors change, but re-test checking ensures that $\text{getNext}^*$ is called over these nodes in the next step.
Algorithm 6 TwigFast*(Q)

1: initialize the list \(L_n\) as empty, and set \(n_i.tail = 0\), for all \(n_i \in \text{Nodes}(Q)\);
2: while \(\neg\text{end}(Q)\) do
3: \(n_{act} = \text{getNext}^*(\text{root}(Q))\)
4: \(v_{act} = \text{getElement}(n_{act})\)
5: if \(\neg\text{isRoot}(n_{act})\) then
6: \(\text{SetEndPointers}(\text{parent}(n_{act}), v_{act}.start)\)
7: if \(\text{isRoot}(n_{act}) \lor \text{parent}(n_{act}).tail \neq 0\) then
8: if \(\neg\text{isLeaf}(n_{act})\) then
9: \(\text{SetEndPointers}(n_{act}, v_{act}.start)\)
10: for \(n_k \in \text{children}(n_{act})\) do
11: \(v_{act}.start_{n_k} = \text{length}(L_{n_k}) + 1\)
12: \(v_{act}.cancestor = n_{act}.tail\)
13: \(n_{act}.tail = \text{length}(L_{n_{act}}) + 1\)
14: append \(v_{act}\) into list \(L_{n_{act}}\)
15: else if \(\text{isEnd}(T_{\text{parent}(n_{act})}) = \text{true}\) then
16: \(\text{ForwardDestoEnd}(\text{parent}(n_{act}))\)
17: if \(\text{isEnd}(T_{n_{act}}) = \text{true}\) then
18: \(\text{ForwardAnstoEnd}(n_{act})\)
19: \(\text{SetRestEndPointers}(Q, \infty)\)

Therefore, the correctness of re-test checking are guaranteed.

2. For time point 2, the current elements of \(n\) and \(n\)’s descendants change. Re-test checking does not call \(\text{getNext}^*\) over \(n\)’s descendants, but the correctness is still guaranteed. The current elements of \(n\) and \(n\)’s descendants are \(\infty(s)\) so the return node of just calling \(\text{getNext}^*(n)\) without the recursive calls over \(n\)’s descendants is the same with the one of calling \(\text{getNext}^*(n)\) together with the recursive calls over \(n\)’s descendants in the following steps, and the return node is \(n\) and its current element is \(\infty\). This means just calling \(\text{getNext}^*(n)\) without the recursive calls over \(n\)’s descendants does not influence correctness of results.

Therefore, the correctness of re-test checking is still guaranteed after applying forward-to-end.

3.4.3 TwigFast* and TwigStack*

In this section, we present the algorithm TwigFast* and TwigStack* which extend TwigFast and TwigStack respectively by applying our proposed approach in Sec-

---

1When the pointer \(PT\) of a stream reaches the end, we use \(\infty\) to denote the current element in this stream.
Algorithm 7 TwigStack\(^*(Q)\)

1: \textbf{while} \(\neg\text{end}(Q)\) \textbf{do}
2: \(n_{\text{act}} = \text{getNext}\(*\text{root}(Q)\))
3: \(v_{\text{act}} = \text{getElement}(n_{\text{act}})\)
4: \textbf{if} \(\neg\text{isRoot}(n_{\text{act}})\) \textbf{then}
5: \(\text{CleanStack}(\text{parent}(n_{\text{act}}), v_{\text{act}.\text{start}})\)
6: \textbf{if} \(\text{isRoot}(n_{\text{act}}) \vee \neg\text{empty}(S_{\text{parent}(n_{\text{act}})})\) \textbf{then}
7: \(\text{CleanStack}(n_{\text{act}}, v_{\text{act}.\text{start}})\)
8: \(\text{MoveStreamstoStack}(T_{\text{act}}, S_{\text{act}}, \text{pointer to top}(S_{\text{parent}(n_{\text{act}})}))\)
9: \textbf{if} \(\neg\text{isLeaf}(n_{\text{act}})\) \textbf{then}
10: \(\text{ShowSolutionswithBlocking}(S_{\text{act}}, 1)\)
11: \(\text{Pop}(S_{\text{act}})\)
12: \textbf{else if} \(\text{isEnd}(T_{\text{parent}(n_{\text{act}})}) = \text{true}\) \textbf{then}
13: \(\text{ForwardDestoEnd}(\text{parent}(n_{\text{act}}))\)
14: \(\text{Advance}(T_{\text{act}})\)
15: \textbf{if} \(\text{isEnd}(T_{\text{act}}) = \text{true}\) \textbf{then}
16: \(\text{ForwardAnstoEnd}(n_{\text{act}})\)
17: \(\text{mergeAllPathSolutions}()\)

TwigFast\(^*\) is shown in Algorithm 6. It uses \text{getNext}\(*\text{root}(Q)\) at line 3 to find a node which has the minimal descendant extension in the query tree \(Q\). After advancing \(PT_{n_{\text{act}}}\) of the stream \(T_{n_{\text{act}}}\), if \(PT_{n_{\text{act}}}\) reaches the end, line 19 immediately skip all the rest elements in the streams of \(n_{\text{act}}\)’s ancestors. If \(PT_{\text{parent}(n_{\text{act}})}\) reaches the end and all the end positions of intervals in the list \(L_{\text{parent}(n_{\text{act}})}\) have been set (i.e., \(\text{parent}(n_{\text{act}}).\text{tail}\) is equal to 0), line 16 skips all the rest of the elements in the streams of \(\text{parent}(n_{\text{act}})\)’s descendants.

TwigStack\(^*\) is shown in Algorithm 7. It can be seen that the way of extension is very similar to TwigFast\(^*\) except for checking whether parent stack is empty instead of judging whether the tail pointer is equal to 0.

Unlike TJEssential, from TwigFast\(^*\) and TwigStack\(^*\), we can see that our approach for avoiding redundant computation can be easily applied to both existing one-phase and two-phase algorithms that are based on \text{getNext} in TwigStack and imposes minor overheads.

3.4.4 Summary

We presented re-test checking and forward-to-end, which can be easily applied to both holistic one-phase and two-phase twig pattern matching algorithms that are based on
3. XML TWIG PATTERN MATCHING

TwigStack, to resolve the redundant computation in getNext. We presented two algorithms TwigFast* and TwigStack* which extend TwigFast and TwigStack respectively by applying our proposed approaches. The better performance of our algorithms has been substantiated in our experiments.

3.5 Experiments

In this section, we first present the experiment results on the performance of TwigMix and TwigFast against TwigList [44] and HolisticTwigStack [45], with both real-world and synthetic data sets. TwigList is the most up-to-date one-phase twig pattern matching algorithm that applies the bottom-up approach. It is claimed to significantly outperform Twig2Stack [13] which, in turn, is claimed to be faster than TwigStack. HolisticTwigStack is also a one-phase holistic twig pattern matching algorithm, but the data structure used is complicated and expensive to maintain. The algorithms are evaluated with the following metrics: (1) number of elements pushed into the S stack and result lists, (2) processing time. Then, we present the experiment results on the performance of TwigFast* against our one-phase algorithm TwigFast, and TwigStack* against TwigStack [8] and TJEssential [36], with both real-world and synthetic data sets. Because TwigFast does not come with other optimization techniques, we can clearly see how much processing time can be reduced by applying our approach. We compare TwigStack* with TwigStack for the same reason. We compare TwigStack* with TJEssential because TJEssential resolves the redundant computation using another approach. Note that TJStrictPre [46] is the latest one-phase algorithm that is based on TwigStack. Since TJStrictPre is an extension of TwigFast, our approach applies to TJStrictPre in the similar way as it applies to TwigFast, and it will improve the performance of TJStrictPre to the similar extent as it improves TwigFast. Therefore, we do not provide an extended version of TJStrictPre and compare its performance with TJStrictPre. The algorithms are evaluated with the metrics of processing time. We selected the queries with different characteristics for more accurate evaluation.
3.5 Experiments

3.5.1 Experiment Set-up

The XML document parser we used is Libxml2 [90]. We implemented a generator in C to generate element encodings \((\text{start}, \text{end}, \text{level})\) for each element in an XML document and store them with B+ tree in Berkeley DB [91]. A simple XPath parser is also implemented, which generates the twig tree from an XPath expression.

We implemented all of the algorithms in C++. All the experiments were performed on 1.7GHz Intel Pentium M processor with 1G RAM. The operating system is Windows 7. We used the following three data sets for evaluation:

- **TreeBank**: We obtained TreeBank XML document from the University of Washington XML repository [92]. The data is deep and has many recursive elements with the same label. The maximal depth is 36 and there are more than 2.4 million elements.

- **DBLP**: DBLP XML document is also obtained from the University of Washington XML repository [92]. This data set is wide and shallow. There are more than 3.3 million elements.

- **XMark**: XMark is a synthetic data set, which is generated by the XML Benchmark Project [93]. We set the scaling factor as 1. The generated document is 111M with more than 1.6 million elements.

3.5.2 Experiment Results

3.5.2.1 TwigMix and TwigFast

We compared the algorithms TwigMix, TwigFast against TwigList and HolisticTwigStack with different twig pattern queries over the three data sets above. The queries are listed in Table 3.3.

**Number of elements moved to \(S\) and result lists** We compared the number of elements pushed into stack \(S\) and appended to the result lists during processing. TwigFast and HolisticTwigStack do not push an element into the stack \(S\), so we compared TwigMix with TwigList. The comparison results are presented in Table 3.4 and 3.5. Apart from the number of elements, we also calculated the reduction percentage made by TwigMix.
3. XML TWIG PATTERN MATCHING

<table>
<thead>
<tr>
<th>Data set</th>
<th>Query</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeBank</td>
<td>TQ1</td>
<td>//S[//MD]//ADJ</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ2</td>
<td>//S[//VP//IN]//NP</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ3</td>
<td>//S[//VP//PP//NP//VBN]//IN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ4</td>
<td>//S[//VP//PP//NN]//NP[//CD]//VBN]//IN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ5</td>
<td>//S[//VP//NP]//VP//PP//IN]//NP//VBN</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ1</td>
<td>//dblp/inproceedings[/title]//author</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ2</td>
<td>//dblp/article[/title]//year</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ3</td>
<td>//dblp/inproceedings[/cite]//title]//author</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ4</td>
<td>//dblp/article[/author]//url//ee</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ5</td>
<td>//article//volume[/cite]//journal</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ1</td>
<td>//item//location//description//keyword</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ2</td>
<td>//people//person//address//zipcode]//profile]//education</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ3</td>
<td>//item//location]//mailbox//mail//emph//description//keyword</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ4</td>
<td>//open auction]//parlist]//bidder</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ5</td>
<td>//people//person]//address//zipcode]//profile</td>
</tr>
</tbody>
</table>

Table 3.3: Queries over TreeBank, DBLP and XMark

As shown in the tables, TwigMix reduces a large percentage (up to 99.9%) of elements moved to stack $S$ and result lists. In some queries, the number of elements reduced is over 1 million. Even though one operation on stack or list is minor, such a large percentage of reduction is enough to significantly reduce the overall time. Additionally, the reduction is significant over all of the three data sets regardless of the structural characteristics of the data, which means the performance improvements brought by TwigMix are consistent.

The reduction of elements appended to result lists shows the advantage of TwigMix in memory consumption. Since the elements appended to result lists will not be released until the results enumeration finishes, they will waste memory space if they do not contribute to the final solutions. Therefore, the useless elements eliminated by TwigMix can significantly reduce the usage of memory.

**Processing time** The comparison of processing time is illustrated in Figure 3.4. As shown, both TwigMix and TwigFast significantly outperform TwigList and HolisticTwigStack. TwigFast shows better performance than TwigMix because it does not need to push elements into stack. This demonstrates that the overhead of maintaining the ancestor and tail pointers in TwigFast is well worthwhile. If we observe the figure together with Table 3.4 and Table 3.5, we can see that the processing time is closely related to the number of elements moved to $S$ and result lists. In other words, the reduction of ele-
3.5 Experiments

<table>
<thead>
<tr>
<th>Query</th>
<th>TwigMix Elements</th>
<th>TwigList Elements</th>
<th>Reduction percentage</th>
<th>Useful Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>TQ1</td>
<td>34</td>
<td>166,940</td>
<td>99.9%</td>
<td>34</td>
</tr>
<tr>
<td>TQ2</td>
<td>608,683</td>
<td>883,479</td>
<td>31.1%</td>
<td>608,683</td>
</tr>
<tr>
<td>TQ3</td>
<td>40,058</td>
<td>1,047,564</td>
<td>96.1%</td>
<td>40,058</td>
</tr>
<tr>
<td>TQ4</td>
<td>11,728</td>
<td>1,283,194</td>
<td>99.1%</td>
<td>11,728</td>
</tr>
<tr>
<td>TQ5</td>
<td>64,745</td>
<td>1,637,551</td>
<td>96.0%</td>
<td>64,745</td>
</tr>
<tr>
<td>DQ1</td>
<td>915,856</td>
<td>1,257,621</td>
<td>27.2%</td>
<td>915,856</td>
</tr>
<tr>
<td>DQ2</td>
<td>553,062</td>
<td>1,485,788</td>
<td>62.8%</td>
<td>553,062</td>
</tr>
<tr>
<td>DQ3</td>
<td>149,015</td>
<td>1,428,692</td>
<td>89.6%</td>
<td>149,015</td>
</tr>
<tr>
<td>DQ4</td>
<td>126,490</td>
<td>1,270,476</td>
<td>90.0%</td>
<td>126,490</td>
</tr>
<tr>
<td>DQ5</td>
<td>52,783</td>
<td>508,499</td>
<td>89.6%</td>
<td>52,783</td>
</tr>
<tr>
<td>XQ1</td>
<td>124,066</td>
<td>316,594</td>
<td>60.8%</td>
<td>124,066</td>
</tr>
<tr>
<td>XQ2</td>
<td>31,861</td>
<td>140,254</td>
<td>77.3%</td>
<td>31,861</td>
</tr>
<tr>
<td>XQ3</td>
<td>63,124</td>
<td>541,558</td>
<td>88.3%</td>
<td>63,124</td>
</tr>
<tr>
<td>XQ4</td>
<td>52,941</td>
<td>184,874</td>
<td>71.4%</td>
<td>52,941</td>
</tr>
<tr>
<td>XQ5</td>
<td>51,325</td>
<td>127,410</td>
<td>59.7%</td>
<td>51,325</td>
</tr>
</tbody>
</table>

Table 3.4: Number of elements pushed into S

![Figure 3.4: Processing Time(ms) of TwigMix and TwigFast](image)

ments for processing directly brings the improvement of performance. For example, for query TQ4, the percentage of reduction is up to 99.1% such that the gap of processing time is huge. For query DQ1, against TwigMix, TwigFast saves 915,856 elements from being pushed into the stack, so the processing time nearly decreases by 2 times.

3.5.2.2 TwigFast* and TwigStack*

The queries for evaluation are listed in Table 3.6 which contain both ‘//’ and ‘/’ edges.

Performance of answering the queries with different characteristics In order to make the experiments more objective, we selected the queries with different characteristics over the TreeBank dataset. For the queries TQ1-TQ5, there is at least one non-leaf node with low frequency in each query. On the contrary, the nodes with low frequencies appear on leaf nodes in the queries TQ6-TQ10. For the queries TQ11-
Table 3.5: Number of elements appended to result lists

<table>
<thead>
<tr>
<th>Query</th>
<th>TwigMix Elements</th>
<th>TwigList Elements</th>
<th>Reduction percentage</th>
<th>Useful Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>TQ1</td>
<td>34</td>
<td>13,686</td>
<td>99.8%</td>
<td>34</td>
</tr>
<tr>
<td>TQ2</td>
<td>608,683</td>
<td>770,052</td>
<td>21.0%</td>
<td>608,683</td>
</tr>
<tr>
<td>TQ3</td>
<td>40,058</td>
<td>207,930</td>
<td>80.7%</td>
<td>40,058</td>
</tr>
<tr>
<td>TQ4</td>
<td>11,728</td>
<td>414,380</td>
<td>97.2%</td>
<td>11,728</td>
</tr>
<tr>
<td>TQ5</td>
<td>64,745</td>
<td>797,917</td>
<td>91.9%</td>
<td>64,745</td>
</tr>
<tr>
<td>DQ1</td>
<td>915,856</td>
<td>1,257,384</td>
<td>27.2%</td>
<td>915,856</td>
</tr>
<tr>
<td>DQ2</td>
<td>553,062</td>
<td>1,484,711</td>
<td>62.7%</td>
<td>553,062</td>
</tr>
<tr>
<td>DQ3</td>
<td>149,015</td>
<td>1,222,789</td>
<td>87.8%</td>
<td>149,015</td>
</tr>
<tr>
<td>DQ4</td>
<td>126,490</td>
<td>1,183,417</td>
<td>89.3%</td>
<td>126,490</td>
</tr>
<tr>
<td>DQ5</td>
<td>52,783</td>
<td>398,708</td>
<td>86.8%</td>
<td>52,783</td>
</tr>
<tr>
<td>XQ1</td>
<td>124,066</td>
<td>255,278</td>
<td>51.4%</td>
<td>124,066</td>
</tr>
<tr>
<td>XQ2</td>
<td>31,861</td>
<td>82,829</td>
<td>61.5%</td>
<td>31,861</td>
</tr>
<tr>
<td>XQ3</td>
<td>63,124</td>
<td>410,540</td>
<td>84.6%</td>
<td>63,124</td>
</tr>
<tr>
<td>XQ4</td>
<td>52,941</td>
<td>167,433</td>
<td>68.4%</td>
<td>52,941</td>
</tr>
<tr>
<td>XQ5</td>
<td>51,325</td>
<td>89,244</td>
<td>42.5%</td>
<td>51,325</td>
</tr>
</tbody>
</table>

**Figure 3.5:** Processing time of queries with different characteristics

TQ15, all the nodes have high frequencies. We compare TwigFast* with TwigFast on these three types of queries. The results are shown in Figure 3.5. As shown in this figure, TwigFast* achieves better performance than TwigFast on all these three types of queries, and is more than 30% faster than TwigFast on most queries. Note that the deficiency of late end often happens on the queries with selective nodes, like the queries TQ1-TQ10. The better performance of TwigFast* on the queries TQ1-TQ10 suggests that the forward to end approach (see Section 3.4.2.2) can avoid the redundant computation brought by late end. Moreover, it should be noted that TwigFast* is many times faster than TwigFast on the query TQ3 and TQ10, and actually there are no matchings of these two queries in the TreeBank dataset. Based on our observation, TwigFast* runs very fast when there are no matchings in the dataset and some nodes are selective. This is mainly because the forward to end approach skips a large amount
3.5 Experiments

<table>
<thead>
<tr>
<th>Data set</th>
<th>Query</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeBank</td>
<td>TQ1</td>
<td>//V//S</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ2</td>
<td>//ADV/[S]//PP//NP</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ3</td>
<td>//A//S//VP</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ4</td>
<td>//ADJ//[NN]//DT</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ5</td>
<td>//VP//[ADV]//VP]//NP]//S</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ6</td>
<td>//S[NP]//CONJ</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ7</td>
<td>//NP]//NP]//PP]//NL</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ8</td>
<td>//S//VP//NP]//<em>HASH</em></td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ9</td>
<td>//S//ADV]//PP]//NP</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ10</td>
<td>//VP]//NP]//PP]//FILE</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ11</td>
<td>//VP]//NP]//NP]//S]//PP]//VP]//NN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ12</td>
<td>//NP//S]//VP]//NN]//PP]//NP]//VBN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ13</td>
<td>//S]//VP]//PP]//NP]//VBN]//IN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ14</td>
<td>//S]//VP]//PP]//NN]//NP]//CD]//VBN]//IN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ15</td>
<td>//S]//VP]//PP]//NP]//S]//PP]//IJ]//VBN]//PP]//NP]//NONE_</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ16</td>
<td>//S]//VP]//NP]//VP]//NP]//ADJ]//NP]//PP]//VBN]//DT]//NN</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ17</td>
<td>//S]//VP]//NP]//ADV]//VBN]//VP</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ18</td>
<td>//S]//VP]//NP]//IJ]//NP]//PP]//ADJ</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ19</td>
<td>//S]//VP]//NP]//JJ]//PP]//NN]//V</td>
</tr>
<tr>
<td>TreeBank</td>
<td>TQ20</td>
<td>//S]//VP]//NN]//PP]//A]//ADJ</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ1</td>
<td>//dblp/inproceedings]//title]//author</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ2</td>
<td>//dblp/article]//author]//title]//year</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ3</td>
<td>//dblp/inproceedings]//cite]//title]//author</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ4</td>
<td>//dblp/article]//author]//title]//[url]//[ee]//year</td>
</tr>
<tr>
<td>DBLP</td>
<td>DQ5</td>
<td>//article]//volume]//cite]//journal</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ1</td>
<td>//item]//location]//description]//keyword</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ2</td>
<td>//people]//person]//address]//zipcode]//profile]//education</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ3</td>
<td>//item]//location]//mailbox]//mail]//emphasis]//description]//keyword</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ4</td>
<td>//people]//person]//address]//zipcode]//id]//profile]//age]//education</td>
</tr>
<tr>
<td>XMark</td>
<td>XQ5</td>
<td>//open_auction]//annotation]//parlist]//bidder]//increase</td>
</tr>
</tbody>
</table>

Table 3.6: Queries over TreeBank, DBLP and XMark

of useless elements. On the other hand, TwigFast* achieves better performance than TwigFast on the queries TQ11-TQ15 mainly because re-test checking approach (see Section 3.4.2.1) avoids a large amount of unnecessary calls of getNext.

**Performance of answering the queries over different datasets** We first compare TwigFast* with TwigFast over the datasets TreeBank, DBLP and XMark. The queries TQ16-TQ20 over TreeBank dataset mix different characteristics we mentioned above. The results are shown in Figure 3.6. As shown in this figure, TwigFast* has better efficiency than TwigFast on all of the queries over the three datasets. In addition, the advantages of TwigFast* are clearer on the queries over the TreeBank and XMark.
3. XML TWIG PATTERN MATCHING

![Figure 3.6: TwigFast vs TwigFast*](image1)

(a) TreeBank  
(b) DBLP  
(c) XMark

Figure 3.6: TwigFast vs TwigFast*

![Figure 3.7: TwigStack, TJEssential, TwigFast* vs TwigStack*](image2)

(a) TreeBank  
(b) DBLP  
(c) XMark

Figure 3.7: TwigStack, TJEssential, TwigFast* vs TwigStack*

dataset. This is mainly because there exist selective nodes in the queries and the queries are relatively complicated. Therefore, the approaches re-test checking and forward-to-end are working together to make the advantages clearer. On the contrary, for the queries over the DBLP dataset (i.e., DQ1-DQ5), all the nodes in the queries have high frequencies. This means the better performance of TwigFast* is mainly brought by avoiding redundant calls of getNext using re-test checking.

Then we compare TwigStack* with TwigStack, TJEssential and TwigFast* over the datasets TreeBank, DBLP and XMark. The results are shown in Figure 3.7. From the results, we can see that both TJEssential and TwigStack* achieves better performance than TwigStack by resolving the redundant computation even though they use different approaches. Additionally, TwigStack* is a little faster than TJEssential because TwigStack* can avoid some redundant computation that TJEssential cannot avoid and TwigStack* imposes less overheads. In addition, TwigFast* achieves better performance than TwigStack*. This is mainly because TwigFast* avoids the merging phase in TwigStack* by storing the whole final solutions in lists.
4

XML Keyword Query Processing

4.1 Introduction

Previously proposed approaches (e.g., XRank [58], MLCA [60], SLCA [59], GDMCT [64], XSeek [65], etc) try to improve the effectiveness of XML keyword search, but they still mainly suffer from the following problems:

1. The results contain too little useful information or too much irrelevant information.

2. The results may not be desired by the user when the keywords are ambiguous.

In this chapter, we try to resolve the problems mentioned above. We first propose XML keyword search system XKmis in Section 4.2, which improves the effectiveness and efficiency of keyword search by dividing an XML document into minimal information segments. In order to improve the precision of inferring the search-for node type, we propose a dynamic approach and the corresponding algorithm DynamicInfer in Section 4.3. Then, in Section 4.4, we present an XML keyword search system XInfer, which exploits users’ typing habit in constructing keyword queries and data statistics in XML documents. Finally, we present an interactive keyword search system XQSuggest in Section 4.5.
4. XML KEYWORD QUERY PROCESSING

4.2 XKMis: Effective and Efficient Keyword Search in XML Databases

4.2.1 Problems of LCA-based Approach

The LCA-based approaches have a common inherent problem, which is that they may link irrelevant XML nodes together and return large amounts of useless information to the user. This problem is called the false positive problem in [61]. Consider the data tree shown in Figure 4.1. Suppose the user wants to find papers in volume 12 which contain the word “Query”, and he submits the query \{volume, 12, Query\}. The early LCA-based approaches (e.g., XRank, SLCA) will return the subtree rooted at SigmodRecord (0, 102, 0), i.e., the entire document to the user, because the node (0, 102, 0) is the lowest common ancestor of nodes (53, 55, 2), (54, 54, 3) and (33, 33, 5) which contain these keywords. In order to solve the false positive problem, Li et al [61] proposed the concept of valuable LCA (VLCA). Suppose there are two XML nodes \( u \) and \( v \) that match the keywords, and their LCA is \( w \). If there exist nodes which have the same label on the paths \( w \rightarrow u \) and \( w \rightarrow v \), \( u \) and \( v \) are considered to be Heterogenous and will not be linked together. Consider the query \{volume, 12, Query\} again. On the paths SigmodRecord (0, 102, 0) → SQL Query (33, 33, 5) and SigmodRecord (0, 102, 0) → 12 (54, 54, 3), nodes issue (1, 51, 1) and issue (52, 101, 1) have the same label. Therefore, nodes SQL Query (33, 33, 5), volume (53, 55, 2) and 12 (54, 54, 3) are regarded as Heterogenous and not connected together. This approach solves the problem in some cases, but there are still many cases for which it will not help. Consider the query...
4.2 XKMIs: Effective and Efficient Keyword Search in XML Databases

![DBLP data tree](image)

**Figure 4.2:** DBLP data tree and partial search results

{Kurt, Software} issued on the a *dblp* data tree in Figure 4.2 (a). Nodes *Kurt* and *Software* are linked together through the LCA *dblp* because there are no nodes with the same label on the paths *dblp* → *Kurt* and *dblp* → *Software*. Hristidis et al. in [64] introduced the concept of *minimum connecting trees* (MCTs) to exclude the subtrees (rooted at LCAs) that do not contain keywords. This approach makes the results more compact, but it cannot prevent unrelated nodes from being linked together. Recently, Liu et al. [66] investigated the axiomatic approach to improve the relevancy of match nodes. They proposed two properties (i.e., *monotonicity* and *consistency*) that an XML keyword search engine should satisfy as well as an algorithm called *MaxMatch* with these properties. *MaxMatch* improves the quality of results, but it cannot solve the false positive problem.

Another problem of existing LCA-based approaches is that the returned answers may contain too little information so they are not informative enough to users. For example, if a query {John} is issued over the data tree in Figure 4.1 the node *John* (24, 24, 6) will be returned. For the query {John, Alex}, the subtree rooted at node *authors* (22, 29, 4) will be returned because it is the LCA of the matched nodes. Obviously, such results do not provide users with much useful information. To make the answers more meaningful, XSeek [65] tried to recognize the possible entities and attributes in the data tree, distinguish between search predicates and return specifications in the keywords, and return nodes based on the analysis of both XML data structures and keyword match patterns. Xu et al. [72] proposed to use *minimal information unit* (MIU) in place of nodes and return the lowest common ancestor MIUs. An XML document is partitioned
into a series of MIUs and they are treated as the minimal unit instead of XML nodes during processing. However, these proposals are LCA-based, so they suffer from the false positive problem mentioned above. For example, for the query \{volume, 12, Query\} discussed above, MaxMatch will return the tree shown in Figure 4.3 (a). The system described in [72] will return the subtree rooted at Sigmodrecord (0, 102, 0).

Our system is not LCA-based. In our system, we partition an XML document into a series of meaningful and self-containing segments, called minimal information segments (MISs). These MISs are similar to the MIUs in [72], but unlike [72], we do not require the existence of a schema file, and we do not return the lowest common ancestor MISs, instead, we return MIS subtrees which consist of MISs logically connected by the keywords (see Section 4.2.2 for the definition of MIS subtree). For example, for the query \{volume, 12, Query\} against the datatree in Figure 4.1, our system will return two partial match results shown in Figure 4.3 (b) and (c). For the query \{John, Alex\}, our system will return the result shown in Figure 4.3 (d) rather than the subtree rooted at authors (22,29,4), and for the query \{Kurt, Software\} against the data tree in Figure 4.2, our system will return two partial match results as shown in Figure 4.2 (b).
and (c). Overall, our system will significantly reduce the false positives and at the same time, make the returned result more meaningful. Furthermore, since we do not need to compute the LCA of nodes, we can use the region code $R$ (rather than the Dewey code) of data trees in our search algorithm. This enables us to significantly reduce the number of stack operations required, making our search more efficient than the LCA-based approaches, especially for large XML documents.

4.2.2 Minimal Information Segment and Answers to Keyword Query

In this section, we first discuss the intuition. Then, we formally define the concept of minimal information segment (MIS). After that, we define MIS subtrees and answers to a keyword query.

4.2.2.1 The Intuition

A keyword search system should return results that are both informative and compact. In other words, each result should not contain too little or too much information. To achieve informativeness, we divide the data tree into minimal information segments (MISs) which represent data about real-world objects. If a keyword occurs in a MIS, then the whole MIS, rather than the node containing the keyword, will be returned as part of the answer. This ensures that every node in the result is in a meaningful context. For example, the MISs in the data tree shown in Figure 4.1 are encircled by dotted lines. They represent the objects sigmodrecord, issue, article and so on. If the keyword John is submitted, then the entire MIS in which John appears, i.e., the MIS rooted at article (12, 30, 3), rather than the node John (24, 24, 6) alone, will be returned to the user.\footnote{In practice, sometimes the MIS may be very large and contain unwanted information. To deal with such cases, we may select only the most important information contained in the MIS. This is left as part of our future work.} To achieve compactness, we do not link two keyword-containing MISs via their lowest common ancestor as in previous work. Instead, we only link them together if they have an ancestor-descendant relationship, or if they are both linked to a third keyword-containing MIS via ancestor-descendant relationships. This will make sure that only closely related MISs are linked together, hence each result contains information about closely related objects. Intuitively, if the XML document is well-designed, then MISs that have ancestor-descendant relationships are directly related, while those that
do not have ancestor-descendant relationship are only loosely related. For example, in the SigmodRecord data tree, article objects that are descendants of an issue object represent articles published in that issue, while articles that are not descendants of an issue are not published in that issue. Therefore, we should only link articles nodes to an issue node which is the ancestor of the articles.

4.2.2.2 Formal Definitions

In this work, we use minimal information segments (MISs) to represent the objects in an XML document. A formal definition of MIS is given below.

**Definition 1** Let \( t \) be a tree and \( u \) be a node in \( t \). The full subtree of \( t \) rooted at \( u \), denoted \( t_u \), is the tree consisting of \( u \) and all descendants of \( u \). A subtree of \( t \) rooted at \( u \) is a tree obtained from \( t_u \) by removing zero or more full subtrees rooted at some descendants of \( u \). A lower subtree of \( t \) is a subtree of \( t \) rooted at a descendant of root(\( t \)).

Note that the above definition of subtree catches any tree consisting of part or all of the nodes in \( t \). But we deliberately stress that a subtree can be obtained by removing full subtrees from other full subtrees.

**Definition 2** Let \( t \) be an XML tree. A node \( u \) in \( t \) is said to be a simple node if it is a leaf node, or has a single child which is a leaf node. A minimal information segment (MIS) in \( t \) is a subtree \( S \) of \( t \) with the following properties: (1) root(\( S \)) is either root(\( t \)), or has siblings with the same label as itself, and (2) root(\( S \)) is not a simple node, (3) no lower subtree of \( S \) is a MIS.

The MISs in the data tree shown in Figure 4.1 are encircled by dotted lines.

With Definition 2 some recognized MISs may still not be informative enough. For example, if the author element in Figure 4.1 contains first name and last name sub-elements, it will also be considered as a MIS. However, this MIS does not contain much information. In practice, we can set a threshold (e.g., the minimum number of elements in a MIS) to guarantee a MIS is informative enough.

Note that any node in \( t \) belongs to one and only one MIS, that is, no two MISs overlap in their nodes. Furthermore, for any two MISs \( S_1 \) and \( S_2 \), either there are no edges from nodes in \( S_1 \) to nodes in \( S_2 \) (when root(\( S_1 \)) is not an ancestor of root(\( S_2 \))), or when there exists MIS \( S_3 \) such that root(\( S_1 \)) \( \prec \) root(\( S_3 \)) \( \prec \) root(\( S_2 \)), or there is a
4.2 XKMIS: Effective and Efficient Keyword Search in XML Databases

Figure 4.4: Search results over the SigmodRecord data tree, each node in a result represents a MIS

single edge from $S_1$ to $S_2$ (when root$(S_1)$ is an ancestor of root$(S_2)$ and there is no MIS $S_3$ such that root$(S_1) \prec$ root$(S_3) \prec$ root$(S_2)$), or there is a single edge from a node in $S_2$ to a node $S_1$. Therefore, if we treat each MIS as a node, and treat an edge from a node in $S_1$ to a node in $S_2$ as an edge from $S_1$ to $S_2$, then all such nodes and the edges between them form a tree, which we call the MIS tree of $t$, denoted MIS$(t)$.

Definition 3 An answer (or result) of a keyword query $K$ over a data tree $t$ is a subtree $S$ of MIS$(t)$ with the following properties: (1) every node in $S$ contains at least one keyword, (2) no lower subtree of $S$ contains all the keywords in $K$, (3) every descendant $m$ of root$(S)$ in MIS$(t)$ that contains strictly part of the keywords is in $S$, provided $m$ is not in another answer $S'$ such that root$(S) \prec$ root$(S')$ and $S'$ contains all of the keywords in $K$.

An answer $S$ is said to be optimal if root$(S)$ contains all of the keywords in $K$. It is said to be sub-optimal if the nodes in $S$ collectively contain all of the keywords but root$(S)$ alone doesn’t. An answer is said to be a partial match if it is neither optimal nor sub-optimal.

Example Consider the query \{Alex, SQL\} over the data tree in Figure 4.1. An optimal result, a sub-optimal result, and a partial match are shown in Figure 4.4 (a), (b) and (c) respectively. Note that the nodes in these answers are MIS nodes, each of which represents a collection of nodes in the original data tree.

In the above example, it is most likely that the user wants information about articles written by Alex which contain “SQL” in the title. The optimal answer in Figure 4.4 (a) has only one MIS which contains all the keywords, which is indeed an article object the user probably expects. A relatively lower possibility is that the user wants information about articles whose title contains “SQL” and which is published in an issue edited by Alex. The sub-optimal answer in Figure 4.4 (b) reflects this requirement of the user.
4. XML KEYWORD QUERY PROCESSING

The partial match in Figure 4.4 (c) reflects the (less likely) possibility that the user is interested in an issue edited by Alex.

Discussion:

- Optimal results have the greatest chance to meet users’ expectations. This is because, if an individual MIS contains all the keywords, then the nodes containing those keywords have the closest relationships.

- Sub-optimal results are a secondary choice compared with optimal results. Sometimes, two or more directly related MISs are needed to find all required data. This is similar to selecting data from multiple tables through joins in relational databases. In the query \{volume, 11, SQL\}, the issue entities of volume 11 and the article entities of SQL should be joined together and the join condition is the ancestor-descendant relationship between them. (Note: The join condition of ancestor-descendant relationship is important because it can prevent the articles of SQL from being linked with the issues which are not volume 11.)

- Partial match results are the last choice because each of them contains strictly part of the keywords. However, a partial match result also has its value especially when there are no or few optimal and sub-optimal results. It can provide users with useful clues and help users to refine their queries. Popular web search engines also return related partial match results but with lower ranks.

4.2.2.3 Partitioning a Data Tree into MISs

A straightforward method to partition the data tree $t$ into MISs is as follows. First, we do a width-first traversal of the data tree to identify all those nodes in $t$ which have the same label as some of its siblings and which are not simple nodes. The full subtrees rooted at these nodes are potential MISs. Then, in each of these full subtrees, we cut off those subtrees that are MISs themselves. Note that we do not need a schema file (such as DTD). If the schema file exists, the MISs can be more accurately identified and the partitioning process can be simplified, e.g., using the method in [72].

Region code for MIS tree and the modified coding scheme of data tree To facilitate the computation of MISs containing a given keyword, we modify the region code of the data tree so that all nodes within a MIS share the same region code. We can
do this by simply replacing the region code of \( v \), for each node \( v \) in \( t \), with the region code of the MIS in which \( v \) lies. For example, all nodes in the MIS rooted at issue (1, 51, 1) in Figure 4.1 will share the region code (1, 51, 1). We call the new region code of \( v \) the \textit{modified region code}, and the original code of \( v \) the \textit{normal region code}. The data tree in Figure 4.5 (a) uses the modified region code.

\subsection*{4.2.2.4 Further Ranking of Answers}

As discussed above, generally an optimal answer is preferable to a sub-optimal answer, and a sub-optimal answer is preferable to a partial match. However, not all answers within the same class (optimal, sub-optimal, or partial match) are equally interesting to the user. Therefore, the answers to a keyword query need to be further ranked according to their degree of relevance. To this end, we classify all MISs in data tree \( t \) into different MIS-\textit{types} according to their root label. Let \( S_1 \) and \( S_2 \) be two MISs in \( t \). If the roots of \( S_1 \) and \( S_2 \) have the same label, we say \( S_1 \) and \( S_2 \) are of the same MIS-type.

Classifying MISs into different MIS-types is very similar to organizing data into different tables in relational databases. A MIS of a MIS-type is like a record in a table. Consider the data tree in Figure 4.1: each MIS of the article-type is like an article record in the article table. Therefore, it is possible to apply the ranking scheme in relational databases. The only difference is that they rank a joining tree of tuples, but we rank a MIS subtree of MISs. In this work, we chose to use a ranking function derived from [94]. Suppose \( T \) is a MIS subtree that contains keywords. The score \( Score(T, K) \) of \( T \) is calculated using the formula below.

\[
Score(m_i, K) = \sum_{k \in K \cap m_i} \frac{1 + \ln(1 + \ln(tf))}{(1 - s) + s \frac{dl}{avdl}} \cdot \ln\left(\frac{N + 1}{df}\right) \tag{4.1}
\]

\[
Score(T, K) = \frac{\sum_{m_i \in T} Score(m_i, K)}{\text{size}(T)} \tag{4.2}
\]

where \( K \) is the keyword query; \( m_i \) is a MIS in \( T \); \( tf \) is the frequency of keyword \( k \) in MIS \( m_i \); \( N \) is the total number of MISs of \( m_i \)'s MIS-type; \( df \) denotes the total number of MISs that contain keyword \( k \); \( dl \) is the length of the text attribute of \( m_i \); \( avdl \) is the average length of the text attribute of \( m_i \)'s MIS-type; \( s \) is a constant value (usually 0.2); and \( \text{size}(T) \) is the size of a MIS subtree \( T \). Different from the size of a joining tree, the...
4. XML KEYWORD QUERY PROCESSING

Algorithm 8 \texttt{XKMis}(K)

1: let $M_i$ be the stream of sorted MISs which contain the keyword $k_i$, for all $1 \leq i \leq n$
2: let $M = \{M_1, \ldots, M_n\}$
3: \texttt{XKMis\_Construct}(K, M, n)
4: \texttt{XKMis\_Output}(OList, SList, PList)

![Diagram of MIS subtree with values 1, 4, 1 and level values 6, 6, 1]

**Figure 4.5:** Example to explain algorithms

size of a MIS subtree includes not only the number of MISs, but also the level of each MIS, as defined below:

$$\text{Size}(T) = \sum_{m_i \in T} 1 + (m_i.\text{level} - \text{root}(T).\text{level}) \quad (4.3)$$

For example, the size of the MIS subtree in Figure 4.4 (b) is $1 + (1 - 1) + 1 + (2 - 1) = 3$.

4.2.3 Algorithm

In this section, we present our algorithm for finding all answers to a keyword query $K$ over data tree $t$. The main algorithm is shown in Algorithm 8. There are two steps involved. In the first step, it calls the procedure \texttt{XKMis\_Construct} to construct the result lists $OList$, $SList$ and $PList$, which contain the optimal, suboptimal, and partial match results respectively. In the second step, it calls \texttt{XKMis\_Output} to output these results. Before explaining the algorithm in detail, we need to define some notions.

4.2.3.1 Notation

Given a keyword query $K = \{k_1, \ldots, k_n\}$, for each keyword $k_i$, there is a stream, $M_i$, consisting of all the MISs which contain $K_i$. The MISs in each stream $M_i$ are arranged in ascending order of their start values. In our implementation, we have chosen to
allow multiple occurrence of the same MIS in a stream: if keyword $k_i$ occurs $N$ times in MIS $mis$, then $mis$ will appear $N$ times in $Mi$ (an alternative is to have an attribute in each MIS to record the number of times each keyword occurs in it). In addition, for each stream $Mi$, there exists a pointer $P_i$ pointing to the current MIS in $Mi$. The function $Advance(Mi)$ moves the pointer $P_i$ to the next MIS in $Mi$. The function $isEnd(Mi)$ judges whether $P_i$ points to the position after the last MIS in $Mi$. The function $getMis(Mi)$ retrieves the current MIS of $Mi$.

To compute streams $M_1, \ldots, M_n$, we build a $B^+$-tree index on all of the words in $t$, each word $k_i$ in the leaf of the $B^+$-tree points to the list of modified region codes of nodes containing $k_i$. Using this index and the modified region codes we can find the streams efficiently.

As mentioned above, the lists $OList$, $SList$ and $PList$ are used to store optimal results, sub-optimal results and partial match results respectively. Each item in these lists is a pointer which points to a list of MISs (Figure 4.5 (b)), which represents a MIS-subtree to be returned to the user. During processing, for each MIS $m$, in addition to the region code ($start$, $end$, $level$), we use four additional attributes $flag$, $optimal$, $score$, and $num$. The $flag$ attribute is a $n$-bit binary number, which indicates which keywords are contained in $m$. The function $SetFlag(flag,i)$ sets the $i^{th}$ bit of the flag to 1, which means the keyword $k_i$ is contained in $m$. The function $countOnes(flag)$ returns the number of 1’s in the flag. The $optimal$ attribute is used to indicate whether a MIS subtree is an optimal result (contains all keywords). The $num$ attribute records the total number of keywords contained in a MIS subtree (duplicate keywords are counted multiple times). The $score$ attribute records the ranking score of a MIS subtree.

### 4.2.3.2 XKMis_Construct

Our algorithm for building the result lists, $XKMis\_Construct$, is shown in Algorithm 9. After a MIS $m$ is initialized, the procedure $getMinMatch$ is repeatedly called (line 3, 4) to get the MIS, $m_{min}$, which has the smallest $start$ value among all the MISs in $M$ that have not been processed. For each MIS $m_{min}$ retrieved, the indexes of the keywords contained in $m_{min}$ are recorded in the $flag$ attribute, and the number of keywords contained in $m_{min}$ are recorded in the $num$ attribute (line 22, 23). The loop will stop when all the MISs in $M$ have been processed and there are no MISs left in the stack $S$ (line 3). From line 5 to 7, we check whether the current MIS $m$ is a descendant of
4. XML KEYWORD QUERY PROCESSING

Algorithm 9 XKMis_Construct \((K, M)\)

Input: keyword query \(K = \{k_1, \ldots, k_n\}\), a set \(M\) of streams

Output: result lists \(OList, SList\) and \(PList\)

1: Initialize a MIS \(m\), set \(m.start = -1, m.end = \infty\)
2: Create an empty stack \(S\)
3: while \(S \neq \emptyset\) OR \(\neg \text{isEnd}(M_1) \land \ldots \land \neg \text{isEnd}(M_n)\) do
4: \(m_{\text{min}} = \text{getMinMatch}(M, n)\)
5: while \(S \neq \emptyset\) AND \(\text{top}(S).end < m_{\text{start}}\) do
6: \(\text{mis}_{\text{top}} = \text{pop}(S)\)
7: \(\text{MovetoResultList}(\text{mis}_{\text{top}}, n)\)
8: if \(m_{\text{min}.start} > m.end\) then
9: \(\text{MovetoResultList}(m, n)\)
10: else
11: if \(m.start \neq -1\) then
12: \(\text{push}(S, m)\)
13: \(m = m_{\text{min}}\)

14: procedure \(\text{getMinMatch}(M, n)\)
15: Initialize a MIS \(\text{mis}\) set \(\text{mis}.start = \text{mis}.end = \infty\)
16: for \(i = 1\) to \(n\) do
17: if \(\text{getMis}(M_i).start < \text{mis}.start\) then
18: \(\text{mis} = \text{getMis}(M_i)\)
19: Set \(\text{mis}.flag = 0, \text{mis.optimal} = true, \text{mis.num} = 0\)
20: for \(i = 1\) to \(n\) do
21: while \(\text{getMis}(M_i).start = \text{mis}.start\) do
22: \(\text{SetFlag}(\text{mis}.flag, i)\)
23: \(\text{mis.num} + +\)
24: \(\text{Advance}(M_i)\)
25: return \(\text{mis}\)

26: procedure \(\text{MovetoResultList}(\text{mis}, n)\)
27: if \(\text{countOnes}(\text{mis}.flag) = n\) then
28: Calculate the ranking score \(\text{mis.score}\)
29: if \(\text{mis.optimal} = true\) then
30: \(\text{Append mis to OList}\)
31: else
32: \(\text{Append mis to SList}\)
33: else
34: if \(S \neq \emptyset\) then
35: \(\text{mis}_{\text{top}} = \text{top}(S)\)
36: Link \(\text{mis}\) to \(\text{mis}_{\text{top}}\)
37: \(\text{CopyFlags}(\text{mis}_{\text{top}}, \text{mis}, n)\)
38: else
39: Calculate the ranking score \(\text{mis.score}\)
40: \(\text{Append mis to PList}\)

41: procedure \(\text{CopyFlags}(\text{parent}, \text{child}, n)\)
42: if \(\text{countOnes}(\text{parent}.flag) < n\) then
43: \(\text{parent.optimal} = false\)
44: \(\text{parent.num} = \text{parent.num} + \text{child.num}\)
4.2 XMIs: Effective and Efficient Keyword Search in XML Databases

top(S), if not, we pop up top(S), and pass it to procedure MovetoResultList to see whether it should be appended to result lists and which result list it should be appended to. Procedure MovetoResultList (line 25-39) does not directly append a MIS mis to a result list. It first checks whether mis contains all the keywords (line 26), if yes, the MIS will be appended to OList or SList depending on the value of mis.optimal. If mis does not contain all the keywords, and the stack S is empty, then mis will be appended to PList. If mis does not contain all the keywords, and the stack S is not empty, then mis will be linked with the MIS at the top of S, mis_top, and mis.flag will be copied over to mis_top (line 37). It should be noted that while copying the flag, the set of keywords contained in mis_top is checked to determine whether its optimal attribute should be set to false (line 41, 42). When the current MIS m does not have descendants that have not been checked (line 8), m is directly passed to procedure MovetoResultList (line 9). If m has descendants that have not been checked (line 10), it is pushed into the stack S. Note that at any moment of time, the MISs in the stack are arranged in descendant-to-ancestor order (from top-down).

Example Consider the keyword query \{XML, John\} and the data tree in Figure 4.5 (a). The first call of procedure getMinMatch returns the MIS (0, 9). The flag attribute indicates it contains both “XML” and “John”, and in total contains 2 keywords (the value of num attribute is 2). Because the initial start value of the current MIS m is -1, m is not pushed into the stack S. At the end of the first loop, m is assigned with m_min (i.e. MIS (0, 9)). In the second loop MIS (1, 4) is returned by getMinMatch. The flag attribute indicates it only contains “XML” and in total contains 1 keyword. Because this MIS is a descendant of the current MIS m = (0, 9), m is pushed into S and the current MIS m becomes (1, 4). Similarly, the next call of getMinMatch returns MIS (2, 2), which causes MIS (1, 4) to be pushed into S, and m becomes (2, 2). The next call of MIS getMinMatch returns (3, 3), which is not a descendant of the current MIS m = (2, 2), so m is directly passed to procedure MovetoResultList. MIS (2, 2) does not contain all the keywords, so it is linked with the MIS at the top of stack S (i.e. MIS (1, 4)). Similarly, MIS (5, 8) also forces MIS (3, 3) to be linked with MIS (1, 4). In the next loop, MIS (6, 6) is returned by getMinMatch. The current MIS m = (5, 8) first makes top(S) (i.e. MIS (1, 4)) to be popped up. Because MIS (1, 4) contains all the keywords and the optimal attribute is false, it is appended to SList. Next, since
4. XML KEYWORD QUERY PROCESSING

MIS\((6, 6)\) is a descendant of \(m = (5, 8)\). There are no MIS left in \(M\), but the stack \(S\) is not empty. A MIS \((\infty, \infty)\) is returned by getMinMatch. This infinite MIS makes the current MIS \(m = (6, 6)\) appended to OList because it contains all the keywords and the attribute optimal is true. Now \(m\) becomes \((\infty, \infty)\), which first causes MIS \((5, 8)\) to be popped up and linked with top\((S)\) (i.e. MIS \((0, 9)\)) because MIS \((5, 8)\) does not contain all the keywords. Then, MIS \((0, 9)\) is also popped up and appended to OList because it contains all the keywords and the optimal attribute is true. The final result lists are shown in Figure 4.5 (b). ⊥

**Time complexity:** XKMis\(_{\text{Construct}}\) scans each \(M_i\) only once to generate all the MIS-subtrees. In the worst case, each MIS retrieved from \(M_i\) needs to be pushed into and popped-up from stack \(S\), which can be finished in constant time. Each MIS popped up from \(S\) is checked to see whether it contains all the keywords. The MIS that does not contain all the keywords is linked with the MIS on the top of the stack \(S\). This can also be done in constant time. Therefore, the worst-case time complexity is \(O(|M|)\) (\(|M|\) is the total number of MISs containing keywords), which is linear in \(|M|\).

4.2.3.3 XKMis\(_{\text{Output}}\)

**Algorithm 10 XKMis\(_{\text{Output}}\) (OList, SList, PList)**

```plaintext
1: for r in OList do
2: MS = RetrieveNodes(r)
3: OutputResult(MS)
4: for r in SList do
5: MS = RetrieveNodes(r)
6: OutputResult(MS)
7: for r in PList do
8: MS = RetrieveNodes(r)
9: OutputResult(MS)
10: procedure OutputResult(MS)
11: while ¬isEnd(MS\(_1\)) \&\& ... \&\& ¬isEnd(MS\(_n\)) do
12: \(n_i = \text{getMinNode}(MS)\)
13: Display(\(n_i\).name, \(n_i\).level)
14: Advance(MS\(_i\))
```

After the result lists are constructed with XKMis\(_{\text{Construct}}\), the final results need to be shown to users. Algorithm [10] shows how to build the final results and display them to the user. In our algorithm, optimal results are displayed first, followed by
sub-optimal results, and finally partial match results. Within each class of results, the individual results are ordered by their ranking score.

As mentioned in the previous section, each result in a result list is a pointer that points to a list $L$ of MISs (i.e. MIS subtree). To display that result, we need to recover the original nodes in these MISs and organize these nodes in the same way as they appear in the original data tree. To achieve this, we build a region index on the start attributes of the MISs. Each start value $val$ in the leaf nodes points to the set of nodes in the MIS whose start value is $val$, and for each node, there is the label or value as well as the normal region code in the original data tree. This index records the nodes and the organization of these nodes within each MIS. Using this index, the procedure $RetrieveNodes$ retrieves all the nodes that are in the list $L$ of MISs. The retrieved XML nodes of each MIS are stored in the stream $MS_i$ ($0 < i < N$, $N$ is the number of MISs in $L$) and $MS = \{MS_1, \ldots, MS_N\}$. The procedure $getMinNode(MS)$ returns the XML node which has the smallest start value among all the streams $MS_i$ in $MS$. Then this node can be displayed with an indent according to its level value (line 13). An example of the final output is shown in Figure 4.5 (c).

4.2.4 Summary

We presented our XML keyword search system $XKMis$. Unlike previous work, our method is not based on the lowest common ancestor (LCA) or its variant. Instead, we divide the XML nodes into minimal information segments (MISs) and return MIS-subtrees which consist of MISs that are more logically connected by the keywords. We will show later that we have conducted extensive experiments to compare our approach with $XRank$ and $SLCA$. The better overall performance, scalability and search quality have been verified by our experiments.

4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

4.3.1 Problem of $XReal$

In this section, we resolve the ambiguity problem of keywords and focus on precisely inferring the search-for node type (SNT) of a query. Figure 4.6 shows that a keyword
4. XML KEYWORD QUERY PROCESSING

Figure 4.6: An XML data tree of SigmodRecord

may have multiple meanings in an XML data tree. For example, 11 appears as a text value of volume and initPage node, and volume exists as an XML tag name and a text value of title node, but which meaning is desired by the user is hard to determine just from the query {volume 11}.

In order to resolve the ambiguities of keywords, Bao et al [1] introduced the statistics of XML data into answering keyword queries, which provides an objective way of identifying users’ major search intention. In their search engine XReal, they first use a formula, which is based on three guidelines (see Section 4.3.2 for details), to infer the search-for node type. Then they use an improved XML TF*IDF ranking strategy to rank individual matches of the identified SNT. For example, for the query {article Karen} over the data tree in Figure 4.6, XReal may first infer article as the SNT. Then it ranks all of the subtrees rooted at article nodes that contain keywords “article” and “Karen”. Unlike XQsuggest, XReal does not need the user’s feedback.

In XReal’s two-step approach, the accuracy of inferring SNT plays a crucial role in returning relevant final results. If the system selects an incorrect SNT, all of the final results would be irrelevant to the user. However, based on our experiments, their way of identifying the SNT suffers from inconsistency and abnormality problems even though each keyword in the query has only one meaning in an XML document. First, for the same query, XReal may return inconsistent SNTs when the data size changes (We will give examples to describe the details of these two problems in Section 4.3.3). In fact, XReal may infer different SNTs even though we simply replicate an XML document two times. This is an unreasonable behavior for an XML keyword search engine. Second,
4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

XReal may infer different SNTs when the keyword queries are similar queries. This is another type of inconsistency problem. For example, given the queries \{article data\} and \{article SQL\} over the data tree in Figure 4.6, both keywords “data” and “SQL” appear as a text value of title node. Intuitively, these two queries should yield the same SNT, but XReal may infer different SNTs for them. Third, XReal may suggest unreasonable SNT when the frequency of keywords is low. However, users often submit keywords which appear as text values and have low frequencies, so this problem is serious for a search engine.

The two problems above show that the formula used by XReal cannot effectively identify the SNT in some cases. In order to resolve these two problems, we propose a dynamic reduction factor scheme. Reduction factor is a constant value in the formula used by XReal to infer the SNT (see Section 4.3.2). In our solution, this factor is dynamic and changes on the fly. Its value is determined by a devised formula. We provide algorithm DynamicInfer which incorporates the dynamic reduction factor scheme to infer the SNT of a query. Extensive experiments verified the effectiveness of our approach to resolve the identified problems.

4.3.2 Background

4.3.2.1 Notations

**Definition 4** Let \( v \) be a node in data tree \( t \). The node type of \( v \) is the sequence of node labels on the path from the root to \( v \) if \( v \) is an internal node, and is denoted \( l_1l_2\cdots l_n \), where \( l_i \) \((1 \leq i \leq n)\) is the label of the \( i^{th} \) node on the path. If \( v \) is a leaf node, its node type is the node type of its parent. The length of a node type is the number of nodes on the path.

The type of a node actually represents the meaning of this node. In Figure 4.6, the node type of author (0.0.2.0.3.0) is the path (SigmodRecord.issue.articles.article.authors.author), and the node type of title (0.0.2.0.0) is the path (SigmodRecord.issue.articles.article.title). The nodes title (0.0.2.0.0) and title (0.0.2.1.0) own the same node type because they share the same path. Node volume (0.0.0) and node title (0.0.2.0.0) have different node types.

Note: For simplicity, we will use the tag name instead of the path of a node to denote the node type throughout this paper if there is no confusion.
4. XML KEYWORD QUERY PROCESSING

4.3.2.2 Overview of XReal

To make the paper self-contained, we review the XML keyword search engine XReal in this section. We will use a running example to briefly introduce its basic ideas.

Based on the fact that each query usually has only one desired node type to search for, keyword query processing in XReal is divided into two steps. First, three guidelines as well as a corresponding formula are used to identify the SNT. Second, an XML TF*IDF ranking mechanism is used to rank the matches of the identified SNT. We mainly discuss the first step because it is closely related to our work.

The three guidelines which are used to guide the identification of SNT are listed below. Given a keyword query \( q \), XReal determines whether a node type \( T \) is the desired node type to search for based on the following three guidelines:

**Guideline 1:** \( T \) is intuitively related to every query keyword in \( q \), i.e. for each keyword \( k \), there should be some (if not many) \( T \)-typed nodes containing \( k \) in their subtrees.

**Guideline 2:** XML nodes of type \( T \) should be informative enough to contain enough relevant information.

**Guideline 3:** XML nodes of type \( T \) should not be overwhelming to contain too much irrelevant information.

To apply these guidelines, XReal uses the following formula to calculate the confidence score \( C_{for}(T, q) \) of a node type \( T \):

\[
C_{for}(T, q) = \log_e(1 + \prod_{k \in q} f_k^T) \ast r^{\text{depth}(T)}
\]  

(4.4)

where \( k \) represents a keyword in query \( q \); \( f_k^T \) is the number of \( T \)-typed nodes that contain \( k \) as either values or tag names in their subtrees; \( r \) is some reduction factor with range \((0, 1]\) and normally chosen to be 0.8, and \( \text{depth}(T) \) represents the depth of \( T \)-typed nodes in document.

In Formula (4.4), the first multiplier (i.e., \( \log_e(1 + \prod_{k \in q} f_k^T) \)) enforces the first and third guidelines. The product of \( f_k^T \) ensures that the selected node type must be related to every keyword in the query, otherwise the score will be 0. For example, with the data in Figure 4.6, the value of \( \prod_{k \in \{\text{volume, Karen}\}} f_k^{\text{author}} \) is 0 because there is no subtree
4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

rooted at the node type *author* that contains the keyword “volume”. In addition, given a keyword $k$, the characteristics of tree structure determines that the node type $T$ at lower levels has a greater chance to have larger values of $f_k^T$. For example, $f_{\text{issue}}$ is smaller than $f_{\text{article}}$ in Figure 4.6. Therefore, the first multiplier usually keeps the level of SNT low enough to make the result small. The second multiplier $r^\text{depth}(T)$ enforces the second guideline by making the level of SNT high enough to contain more information.

4.3.3 Analysis of XReal’s Weaknesses

As we stated earlier, XReal uses Formula (4.4) to identify the search-for node type (SNT) of a query. However, there exist inconsistency and abnormality problems when this formula is applied. In this section, we will use examples to explain the details of these problems, and discuss why these problems are serious to an XML keyword search engine. To facilitate our discussion, we first give a data set (i.e., Data set 1) in Figure 4.7(a), which is identical to the SigmodRecord data set obtained from [92]. In the data set, we also provide the statistics of a certain meaning of the words that will be used in our examples. For example, the keyword “Karen” exists in two different articles as the text value of author node, and these two articles exist in two different issues. We replicated Data set 1 two times to get Data set 2 (Figure 4.7(b)). All the examples in this section will be based on these two data sets.

[Inconsistency problem 1] For the same query, Formula (4.4) may infer inconsistent SNTs when the size of a data set changes. To illustrate this problem, we simply replicate
4. XML KEYWORD QUERY PROCESSING

![XML data tree of customers](image)

**Figure 4.8:** An XML data tree of customers

**Table 4.1:** $C_{for}(T,q)$ for Inconsistency Problem (Query $q$: \{article Karen\})

<table>
<thead>
<tr>
<th>Data set</th>
<th>$T$</th>
<th>$\log_e(1 + \prod_{k \in {article,Karen}} f_k^T) \times 0.8^{depth(T)}$</th>
<th>$C_{for}(T,q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data set 1</td>
<td>issue</td>
<td>$\log_e(1 + 67 \times 2) \times 0.8$</td>
<td>3.9242</td>
</tr>
<tr>
<td></td>
<td>article</td>
<td>$\log_e(1 + 1504 \times 2) \times 0.8^3$</td>
<td>4.1008</td>
</tr>
<tr>
<td>Data set 2</td>
<td>issue</td>
<td>$\log_e(1 + 134 \times 4) \times 0.8$</td>
<td>5.0288</td>
</tr>
<tr>
<td></td>
<td>article</td>
<td>$\log_e(1 + 3008 \times 4) \times 0.8^3$</td>
<td>4.8104</td>
</tr>
</tbody>
</table>

a data set two times to simulate the change of data size.

**Example** Given the query \{article, Karen\} over Data set 1 and 2, we calculate and compare the confidence score $C_{for}(T,q)$ of the node types article and issue (Note: other node types, such as SigmodRecord and articles, are ignored here), and list the results in Table 1. Intuitively, article should be identified as the SNT no matter what the data size is, but the system infers inconsistent SNTs and selects an unreasonable node type issue when the data has a larger size. ⊥

Inconsistency problem is serious for an XML keyword search engine because the data in reality is not static. The data set may become larger or smaller when the data is inserted or deleted, but the precision of inferring SNT should not be affected by the scale of data.

**[Inconsistency Problem 2]** For two similar queries, Formula (4.4) may infer inconsistent SNTs. Here by similar queries we mean their keywords have the same node types. Intuitively, similar queries are supposed to have the same SNT.

**Example** Consider the queries \{article John\} and \{article Karen\} over Data set 1. These two queries are similar and the SNTs should be the same. However, as shown in Table 2, XReal returns article for \{article Karen\}, and returns issue for \{article John\}. ⊥
### 4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

Table 4.2: $C_{for}(T, q)$ for Inconsistency Problem 2

<table>
<thead>
<tr>
<th>$q$</th>
<th>$T$</th>
<th>$log_e(1 + \prod_{k\in q} f_k^T) \times 0.8^{depth(T)}$</th>
<th>$C_{for}(T, q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{article Karen}</td>
<td>issue</td>
<td>$log_e(1 + 67 \times 2) \times 0.8$</td>
<td>3.9242</td>
</tr>
<tr>
<td></td>
<td>article</td>
<td>$log_e(1 + 1504 \times 2) \times 0.8^3$</td>
<td>4.1008</td>
</tr>
<tr>
<td>{article John}</td>
<td>issue</td>
<td>$log_e(1 + 67 \times 24) \times 0.8$</td>
<td>5.9067</td>
</tr>
<tr>
<td></td>
<td>article</td>
<td>$log_e(1 + 1504 \times 33) \times 0.8^3$</td>
<td>5.5360</td>
</tr>
</tbody>
</table>

Table 4.3: $C_{for}(T, q)$ for Abnormality Problem

<table>
<thead>
<tr>
<th>$q$</th>
<th>$T$</th>
<th>$log_e(1 + \prod_{k\in{Karen}} f_k^T) \times 0.8^{depth(T)}$</th>
<th>$C_{for}(T, q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Karen}</td>
<td>issue</td>
<td>$log_e(1 + 2) \times 0.8$</td>
<td>0.8789</td>
</tr>
<tr>
<td></td>
<td>article</td>
<td>$log_e(1 + 2) \times 0.8^3$</td>
<td>0.5625</td>
</tr>
</tbody>
</table>

[Abnormality Problem] Formula [4.4] may infer unreasonable SNTs when the keywords in a query have low frequencies. We use the keywords that occur in text values to illustrate this problem.

**Example** Consider the query {Karen} over the Data set 1. In this query, the keyword “Karen” has low frequency. We calculate $C_{for}(T, q)$ of the node type article and issue, and list the result in Table 3. From the result, it can be seen that the node type issue will be selected as the SNT, but article is intuitively a better choice in this case. More seriously, if the user submits a keyword which has extremely low frequency, XReal even returns the node type SigmodRecord as the SNT. ⊥ The Abnormality Problem is also a serious problem for an XML keyword search engine, because it is very common for the user to submit keywords which appear as text values in a data set, and normally this kind of keywords have low frequencies.

**Discussion** Inconsistency Problem 1 emerges because the logarithm function (i.e., $log_e(1 + \prod_{k\in q} f_k^T)$) in Formula [4.4] grows more and more slowly when its argument becomes larger due to the increased data size, but the exponential function (i.e., $r^{depth(T)}$) in Formula [4.4] decreases the value of the logarithm function more quickly when the depth increases. Therefore, the level of SNT tends to go up when the data size becomes larger. For similar queries, their argument values of the logarithm function may be very different but the reduction factor remains the same, so their SNTs are very likely to be different and lead to Inconsistency Problem 2. On the other hand, these two
inconsistency problems are more likely to occur on deep tree structures than shallow ones. For example, the data tree in Figure 4.8 is shallower than the SigmodRecord data tree in Figure 4.6. It is very hard for the node type customers to be selected as the SNT because there is only one customers node in the data tree which results in its very small confidence score compared with the confidence score of customer. Although more customer nodes may be inserted into the data set, the number of customers node does not change.

The Abnormality Problem arises because for the node types at different levels, their values of the first multiplier in Formula (4.4) are so close when the frequency of keywords is low that the second multiplier can easily make the node types at higher levels to be the SNT. In other words, the first multiplier will become negligible when the keywords have low frequencies. For example in Data set 1, the value of $\log_e f_{article,Karen}$ is the same with the value of $\log_e f_{issue,Karen}$, so their confidence scores are mainly determined by the second multiplier in the formula. In addition, this problem is likely to happen on both deep and shallow data trees. For the data tree in Figure 4.8 if the keyword has a very low frequency, the node type customers can also be inferred as the SNT, but this cannot be accepted.

One may wonder whether these problems can be resolved by manually adjusting the reduction factor for each data set by the database administrator. The problem is the reduction factor is not closely related to the scale of the data but the argument value of the logarithm function in Formula (4.4). In other words, the reduction factor is closely related to the number of occurrences of each node type containing the keywords in a query. Therefore, it is impossible to choose a value which can fit for every query. The only way is to use a dynamic reduction factor (i.e., $r$) in the formula and setting its value on the fly. In this paper, we will explore how to achieve this.

### 4.3.4 Inferring Search-for Node

In this section, we introduce the statistics of underlying data utilized in our solution, and illustrate how to employ dynamic reduction factor scheme in improving the precision of SNT identification. Before discussing our solution, some preliminaries are introduced.
4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

4.3.4.1 Preliminary definitions

**Definition 5** (Ancestor Node Type) Given a node type \( T \equiv l_0.l_1.\ldots.l_n \), we say the path \( l_0.l_1.\ldots.l_i \) is an ancestor node type of \( T \), for any \( i \in [0, n-1] \).

For example in Figure 4.6, the node type `Sigmodrecord.issue` is an ancestor node type of `SigmodRecord.issue.articles.article`.

**Definition 6** (The entity-type of a node type) If a node type \( T \) is not the node type of some entity node, its entity-type is its ancestor node type \( T' \) such that (1) \( T' \) is the node type of some entity node, and (2) \( T' \) is the longest among all ancestor node types of \( T \) satisfying condition (1). If \( T \) is the node type of some entity node, its entity-type is itself. Node types that have the same entity-type are called neighbor node types.

Note that every node type in the data tree owns one and only one entity-type.

**Example** In Figure 4.6, the entity-type of node type `initPage` is node type `article`. Node type `article`’s entity-type is itself (i.e., `article`). The node types `article`, `title`, `initPage`, `endPage`, `authors` and `author` are neighbor node types because they share the same entity-type `article`. \( \bot \)

4.3.4.2 Dynamic Reduction Factor

We use a dynamic reduction factor scheme to adjust the reduction factor (i.e., \( r \)) in Formula (4.4) on the fly in order to resolve the inconsistency and abnormality problems.

Intuitively, the inconsistency and abnormality problems shown in Section 4.3.3 are all caused by an inappropriate reduction factor value and the inability of that value to adapt to the user query. The score of a node type becomes lower than that of some ancestor node types, causing the ancestor node type to be chosen as the search-for node type. In order to solve this problem, we propose the following guideline as the basis of our approach.

**Guideline 4** Given a node type \( T \) which achieves the highest confidence score among its neighbor node types, the reduction factor should be such that it ensures that no ancestor node type \( T' \) of \( T \) achieves a higher confidence score than \( T \) if we ignore the

\(^1\)See Definition 13 in Section 4.5.2.1 for the details of entity node and entity type.
occurrence of keywords in other parts of the data tree than subtrees rooted at $T$-typed nodes.

It should be noted that among a set of neighbor node types, the node type of the entity nodes normally has the highest confidence score according to Formula (4.4). For example, consider the query \{SQL\} over the data tree in Figure 4.6. The confidence score of $article$, which is the node type of entity nodes, is larger than $title$ (Note: Here we do not consider other neighbor node types, such as initPage and endPage, because their confidence scores are 0) because $f_{article}^{SQL}$ is equal to $f_{title}^{SQL}$, and $article$-typed nodes are higher than $title$-typed nodes in the data tree.

**Example** The user submits query \{article Karen\} over the data tree in Figure 4.6. After calculation, the node type $article$ achieves the highest confidence score among all of the neighbor node types that have the entity-type $article$. According to Guideline 4 above, the new reduction factor should guarantee that the confidence score of $article$ is larger than the confidence score of $articles$, $issue$ and $SigmodRecord$ if we ignore the occurrence of the keywords “article” and “Karen” in other parts of the data tree than the subtrees rooted at $article$-typed nodes. ⊥

To formalize Guideline 4, we propose the statistics $f_{T'}^{T,T}$.

**Definition 7** $f_{T'}^{T,T}$ is the number of $T'$-typed nodes that contain keyword $k$ in the subtrees of their $T$-typed descendant nodes in the XML database.

**Example** Over the data tree in Figure 4.6, $f_{data}^{issue,article}$ is the number of $issue$-typed nodes that contain keyword “data” in the subtrees rooted at $article$-typed nodes. $f_{data}^{issue,article}$ ignores the occurrence of keywords “data” in other parts of the data tree than the subtrees rooted at $article$-typed nodes. ⊥

Guideline 4 can be formally defined with the following formula:

$$\frac{\log_e(1 + \prod_{k \in q} f_k^T) * r^{\text{depth}(T)}}{\log_e(1 + \prod_{k \in q} f_k^{T',T}) * r^{\text{depth}(T')}} > 1, (T' \in \text{Ancestors}(T))$$

(4.5)

so the reduction factor $r$ should satisfy the condition below.

$$\max\left\{ \frac{\text{depth}(T) - \text{depth}(T')}{\log_e(1 + \prod_{k \in q} f_k^{T',T})} \right\} = \frac{\log_e(1 + \prod_{k \in q} f_k^{T'})}{\log_e(1 + \prod_{k \in q} f_k^T)} < r \leq 1, (T' \in \text{Ancestors}(T))$$

(4.6)
4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

In the two formulas above, \( T \) is the node type which achieves the highest confidence score among all of its neighbor node types and is normally the node type of the entity nodes. \( T' \) is an ancestor node type of \( T \). \( f_k^T \) is the number of \( T \)-typed nodes that contain \( k \) in their subtrees. \( f_k^{T',T} \) is the number of \( T' \)-typed nodes that contain \( k \) in the subtrees of their \( T \)-typed descendant nodes.

**Reservation Space** The value of reduction factor \( r \) should satisfy the condition in Formula (4.6). In practice, we need to determine the exact value of \( r \), and we can add a small value to the max function. This small value is called reservation space (i.e., \( rs \)) in our approach, which cannot be too small or too large. If \( rs \) is too small, the confidence score of \( T' \) is very easy to exceed the confidence score of \( T \) when there exist more \( T' \)-typed nodes that contain the keywords in the parts of the data tree excluding the subtrees rooted at \( T \)-typed nodes. If \( rs \) is too large, it is very difficult for \( T' \) to be selected as the SNT even though much more \( T' \)-typed nodes contain the keywords in the parts of the data tree excluding the subtrees rooted at \( T \)-typed nodes.

Based on our experiments, 0.05 is an appropriate value for \( rs \). The formula for the reduction factor \( r \) is as follows, and the maximum value of \( r \) is 1.

\[
r = \min\{\max\{ \frac{\log_e(1 + \prod_{k \in q} f_k^{T',T})}{\log_e(1 + \prod_{k \in q} f_k^{T,T})} \} + 0.05, 1\}, (T' \in \text{Ancestors}(T))
\]

(4.7)

4.3.4.3 Algorithm

\texttt{XReal} computes the confidence scores of all node types and selects the node type with the highest confidence score as the SNT. For our approach, in order to enforce Formula (4.7), we cannot simply compute the confidence score of each node type, and need to design a new algorithm.

Our algorithm \texttt{DynamicInfer} for inferring the SNT is shown in Algorithm 11. Now we explain this algorithm. We first set the initial reduction factor as 0.8 (line 1). At line 2, we retrieve all of the leaf entity-types using procedure \texttt{GetLeafEntityTypes(NT)}. Leaf entity-types are the entity-types that do not have any descendant entity-types. For each leaf entity-type, we use a bottom-up strategy to infer the node type \( T_{current} \) which achieves the highest confidence score in current bottom-up process. In each bottom-up
Algorithm 11 DynamicInfer(NT, q)

Input: Node types $NT = \{nt_1, ..., nt_n\}$, a query $q$

Output: The search-for node type $T_{for}$

1: $r = 0.8$, $C_{for} = 0$, $T_{for} = \text{null}$  // $r$: reduction factor; $C_{for}$ is the confidence score of $T_{for}$
2: $le = \text{GetLeafEntityTypes}(NT)$  // $le$ is a list of leaf entity types
3: for $i = 1$ to $le.length()$ do
4:   $C_{path} = 0$, $T_{path} = \text{null}$
5:   $et = le[i]$
6:     while $et \neq \text{null}$ do
7:       $nn = \text{GetNeighborNodeTypes}(NT, et)$  // $nn$ is a set of neighbor node types which have entity type $et$
8:       $(T_{neighbor}, C_{neighbor}) = \text{GetNTWithHighestConfidenceScore}(nn)$  // $T_{neighbor}$ is the node type which has the highest confidence score in $nn$, and its confidence score is $C_{neighbor}$
9:       if $C_{neighbor} > C_{current}$ then  // $C_{current}$ is the confidence score of $T_{current}$ which has the highest confidence score in current bottom-up process
10:          $T_{current} = T_{neighbor}$
11:          $r = \text{AdjustReductionFactor}(T_{neighbor}, et)$
12:          $C_{neighbor} = \text{CalculateConfidenceScore}(T_{neighbor})$
13:          $C_{current} = C_{neighbor}$
14:          $et = \text{GetNextAncestorEntityType}(et)$  // Get the closest ancestor entity type of $et$
15:     end while
16:     $C_{current} = \text{CalculateConfidenceScore}(T_{current})$
17:     if $C_{for} < C_{current}$ then
18:        $T_{for} = T_{current}$
19:        $C_{for} = C_{current}$
20: end if
21: procedure $\text{AdjustReductionFactor}(T_{highest}, et)$
22:     $A = \text{GetAncestorNodeTypes}(et)$  // $A$ is a set of ancestor node types of $et$
23:     $r = \min\{\max\{\frac{\log_b(1+\prod_{k \notin q} \frac{f_k^{a,T_{highest}}}{f_k^{a,T_{highest}}})}{\log_b(1+\prod_{k \notin q} f_k^{a,T_{highest}})} + 0.05, 1\}, (a \in A)\}$
24: return $r$
4.3 Effectively Inferring the Search-for Node Type in XML Keyword Search

Process, we first use procedure $\text{GetNeighborNodeTypes}(NT, et)$ to collect all of the neighbor node types which have the entity-type $et$ (line 7). Then we use procedure $\text{GetNTWithHighestConfidenceScore}$ to get the node type $T_{neighbor}$ that achieves the highest confidence score $C_{neighbor}$ among the neighbor node types collected in the last step. If $C_{neighbor}$ is larger than $C_{current}$, we assign $T_{current}$ with $T_{neighbor}$ (line 10), and adjust the reduction factor with procedure $\text{AdjustReductionFactor}$ (line 11). Next we need to recalculate the confidence score $C_{neighbor}$ of $T_{neighbor}$ by calling procedure $\text{CalculateConfidenceScore}$ which uses Formula (4.4) (line 12), and set $C_{current}$ with $C_{neighbor}$ (line 13). We repeat the steps above until all of the ancestor entity-types of current leaf entity-type have been processed, and get the node type $T_{current}$ which has the highest confidence score in current bottom-up process. We reset $r$ to 0.8 (line 15), and recalculate the confidence score of $T_{current}$ (line 16) because we want to compare the node types that achieve the highest confidence score in different bottom-up process using the same reduction factor. We start another bottom-up process if there are other leaf entity-types unprocessed. The initial value of reduction factor $r$ in each bottom-up process is 0.8. Eventually, $T_{for}$ is the node type which achieves the highest confidence score among all bottom-up processes.

We use the query \{article John\} to illustrate $\text{DynamicInfer}$.

**Example** Consider the query \{article John\} over Data set 1. The algorithm first retrieves all of the leaf entity-types (line 2). In this data set, there is only one leaf entity-type article, so there is only one bottom-up process. The bottom-up process starts. We first collect all the neighbor node types that have the entity-type article (line 7) and apply Formula (4.4) to find the node type with the highest confidence score among these neighbor node types (line 8), which is article and its confidence score $C_{neighbor}$ is 5.5360 in this case. Because $C_{neighbor}$ is larger than $C_{current}$, we set $T_{current}$ with $T_{neighbor}$ (i.e., article) and adjust the reduction factor. After the adjustment (line 11), the new reduction factor is 0.8764. Then we recalculate the confidence score of article using Formula (4.4) (line 12). The new confidence score of article is 7.2783. Then we assign this value to $C_{current}$ (line 13). At line 14, we get the closest ancestor entity-type of article, which is issue. We also retrieve all of its neighbor node types and compute the confidence scores of these node types. We find that the node type issue has the highest confidence score 6.4708. Because 6.4708 is smaller than $C_{current}$
4. XML KEYWORD QUERY PROCESSING

Figure 4.9: A Sample XML data tree of SigmodRecord

(i.e., 7.2783), we do not change $T_{current}$ and the reduction factor. Because issue does not have ancestor entity-type, the bottom-up process ends. At line 18, $T_{for}$ is assigned with article. article is the only leaf entity-type in this case, so the algorithm ends. The node type article is identified as the SNT. ⊥

For the example above, XReal infers the node type issue as the SNT. For this case, article is a preferable SNT to issue.

4.3.5 Summary

In this section, we identified the inconsistency and abnormality problems in the approach of inferring the search-for node type used by XReal. To resolve these problems, we propose a dynamic reduction factor scheme as well as algorithm DynamicInfer to apply this scheme.

We have implemented the proposed approach and the extensive experiments showed that our approach can resolve inconsistency and abnormality problems.

4.4 Exploiting User’s Typing Habit and Data Statistics for Effective XML Keyword Search

4.4.1 Observation

In this section, we focus on resolving keyword ambiguities and inferring users’ search intention by exploiting their typing habit as well as the data statistics. We observe that
users seldom issue a query arbitrarily. Instead, most of the time they construct queries logically. They usually place closely related keywords at adjacent positions. For example, if the user intends to retrieve the articles about database from issue 16, he is more likely to submit the query \{issue 16 database\} than the query \{16 database issue\} because the keywords “16” and “issue” are closely related. This intuition motivates us to infer a keyword’s meaning by preferentially considering this keyword together with its adjacent keywords instead of equally considering this keyword with all of the keywords because the accuracy of inference may be influenced by those keywords that are loosely related to this keyword. In the example above, it is more likely to infer “issue” as the tag name issue (i.e., the desired meaning) by preferentially considering this keyword together with its adjacent keyword “16” than considering all three keywords together because introducing “database” may increase the chances to infer “issue” as another meaning.

In our work, we first design a formula to infer the desired meaning of a pair of adjacent keywords without considering other keywords in the query, which takes into account the statistics and structural properties of a keyword’s different meanings. Then we propose the Pair-wise Comparison Strategy (PCS), which utilizes the inference results of pairs of adjacent keywords, to further infer a set of likely search intentions and apply a proper ranking scheme. The result subtrees of the inferred search intentions are returned to users in clusters, which can save users’ browsing time. In addition, our system does not immediately calculate and show all result subtrees of a search intention after the user clicks the “Search” button. Instead, it first presents one result of each search intention to the user for verification. If this result matches the user’s search intention, the user can click a button to get all this kind of results.

For ease of understanding, we illustrate the result displayed by our system using the following two examples and the three screenshots in Figure 4.10. We also compare our results of the sample queries with the results of XReal and XBridge.

Example The user submits the query \{issue 16 database\} over SigmodRecord dataset to retrieve the articles about database from issue 16. As shown in Figure 4.10(a), our system returns result subtrees of one search intention that is the articles from issue 16 and whose titles contain “database”, and the results satisfy the user’s expectation.\footnote{The user can click “Show more results similar to the above” to get the rest of result subtrees.}
Figure 4.10: Three Screenshots of Our System
4.4 Exploiting User’s Typing Habit and Data Statistics for Effective XML Keyword Search

For **XReal** and **XBridge**, although they infer *issue* as the SWT, many irrelevant articles (e.g., the articles whose initPage is 16) are still returned to the user. Therefore, the user has to browse all of the result subtrees to locate the desired results. ⊥

**Example** In Figure 4.10(b), the user submits the query \{VLDB 2000 query\} over DBLP dataset. Our system returns result subtrees of two search intentions and ranks the VLDB inproceedings published in year 2000 and whose titles contain “query” first. Our system also returns the articles whose title contains “query” from the VLDB journal of year 2000. This is mainly because between inproceedings and articles, it is hard for a system to tell which one is really desired by the user. Therefore, it is safer to return all these two kinds of results to the user. On the contrary, **XReal** and **XBridge** just infer *inproceedings* as the SWT, so if the user wants the articles from VLDB journal, all of the desired results will be missing. Furthermore, if the user submits the query \{VLDB Journal 2000 database\} instead, our system only returns the articles from VLDB journals as shown in Figure 4.10(c) because the new keyword “Journal” makes the possibility of the articles coming from VLDB Journal larger. ⊥

In the examples above, our system returns better results than **XReal** and **XBridge**, and the new way of presenting results increases the *useability* of system.

### 4.4.2 Inferring the Meaning of Two Adjacent Keywords

In this section, we show how to infer the desired meaning of a pair of adjacent keywords, which is fundamental in resolving the ambiguities of keywords. Before discussing our approach, we introduce some definitions.

#### 4.4.2.1 Preliminary Definitions

For simplicity, we will use the tag name of a node to denote the node type\(^1\) when there is no confusion. For convenience, we use \( T_1 \prec T_2 \) to denote that \( T_1 \) is an ancestor node type\(^2\) of \( T_2 \).

**Definition 8** Let \( T_1, \ldots, T_n \) be node types. The longest common ancestor of \( T_1, \ldots, T_n \), denoted \( \text{NtLCA}(T_1, \ldots, T_n) \), is a node type \( V \) such that (1) \( V \prec T_i \) for all \( i \in [1, n] \); (2) there is no node type \( U \) such that \( V \prec U \prec T_i \) for all \( i \in [1, n] \).

\(^1\)See Definition 4 in Section 4.3.2.1 for the details of node type.
\(^2\)See Definition 5 in Section 4.3.2.1 for the details of ancestor node type.
4. XML KEYWORD QUERY PROCESSING

For example, the longest common ancestor of the node types
(SigmodRecord.issue.articles.article.title) and (SigmodRecord.issue.articles.article.authors.author)
is the node type (SigmodRecord.issue.articles.article).

In our system, each word in the data tree is associated with one or more node
types. We use the following definition to determine whether a node type is associated
with a word.

**Definition 9 (Word Type)** Let $t$ be the data tree and $w$ be a word that occurs in $t$.
A node type is associated with $w$ iff some nodes of this node type directly contain $w$.
This node type is called a word type of $w$.

In Table 1, we list the word types of several words for the data tree in Figure 4.9.

<table>
<thead>
<tr>
<th>Word No.</th>
<th>Word Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SigmodRecord.issue</td>
</tr>
<tr>
<td>2</td>
<td>SigmodRecord.issue.articles.article.title</td>
</tr>
<tr>
<td>3</td>
<td>SigmodRecord.issue.volume</td>
</tr>
<tr>
<td>4</td>
<td>SigmodRecord.issue.articles.article.initPage</td>
</tr>
<tr>
<td>5</td>
<td>SigmodRecord.issue.articles.article.title</td>
</tr>
<tr>
<td>6</td>
<td>SigmodRecord.issue.articles.article.authors.author</td>
</tr>
</tbody>
</table>

Next, we formally define the *search intention*:

**Definition 10 (Search Intention)** Given a keyword query $K = \{w_1, \cdots, w_n\}$ and
their corresponding word type sets $\{WT_1, \cdots, WT_n\}$, a search intention of this query is
a tuple of word types $(wt_1, \cdots, wt_n)$, where $wt_i \in WT_i$ ($1 \leq i \leq n$). All possible search
intentions of $K$ correspond to the set of tuples $WT_1 \times \cdots \times WT_n = \{(wt_1, \cdots, wt_n) : wt_i \in WT_i\}$.

**Discussion** We use a different way of representing a search intention from **XReal** and
**XBridge**. **XReal** and **XBridge** use Search-for Node Type (SNT) (or promising result
type) to represent a search intention. This representation has some disadvantages. For
example, if **XReal** and **XBridge** infer *issue* as the SNT for the query $\{\text{issue 16 database}\}$
over SigmodRecord dataset (see Example 4.4.1), all result subtrees that are rooted at
nodes *issue* and contain the keywords “issue”, “16” and “database” will be returned to
the user. This means **XReal** and **XBridge** do not separate the articles about database
from issue 16 from the articles about database whose initPage is 16 and some other articles. Therefore, the user has to spend much time on locating the desired result subtrees. On the contrary, in our system, each keyword has a specific meaning, thus articles about database from issues 16 and articles about database who initial page is 16 will belong to different search intentions, and they will be returned to the user in different clusters. The user can then choose his real search intention and retrieve all of the relevant search results.

4.4.2.2 Inferring the Word Types of Two Adjacent Keywords

A search intention is a tuple of word types. In our work, we first design a formula to infer the desired word types of a pair of adjacent keywords without considering other keywords, then we use the inference results to further infer the word type of each keyword.

If two adjacent keywords \( w_i \) and \( w_{i+1} \) are taken as a keyword query with two keywords, the number of possible search intentions is equal to \( |WT_i| \times |WT_{i+1}| \). We design a formula to compute the proximity score between \( wt_i \) and \( wt_{i+1} \) which is used to evaluate how closely \( wt_i \) and \( wt_{i+1} \) are related, where \((wt_i, wt_{i+1}) \in WT_i \times WT_{i+1}\). The tuple \((wt_s, wt_t)\) that achieves the largest proximity score is considered as the desired search intention of \( w_i \) and \( w_{i+1} \).

Basically, the proximity score between two word types is affected by the distance between them and the statistics of them.

**Distance between two word types** In our work, the distance between two word types \( wt_1 \) and \( wt_2 \) is defined as the total number of edges from \( \text{NtLCA}(wt_1, wt_2) \) to the ends of \( wt_1 \) and \( wt_2 \), which can be calculated using the following formula:

\[
\text{Dis}(wt_1, wt_2) = \text{len}(wt_1) + \text{len}(wt_2) - 2 \times \text{clen}(wt_1, wt_2)
\] (4.8)

In the formula above, \( \text{len} \) is the length of a word type. \( \text{clen}(wt_1, wt_2) \) is the length of the longest common ancestor (see Definition 8) of \( wt_1 \) and \( wt_2 \). Consider word types 2 and 6 in Table 1. According to Formula (4.8), the distance between them is \( 5 + 6 - 2 \times 4 = 3 \).

Intuitively, the shorter the distance between two word types, the more closely these two word types are related.
In our work, we use $p^{\text{Dis}(\text{wt}_1, \text{wt}_2)}$ to formulate the intuition above. $p$ is a tuning parameter which is used to determine how much penalty should be given to the distance between two word types.

**Statistics of word types** Another important factor that influences the proximity score is the statistics of word types.

Given any two word types of two adjacent keywords, we formulate the influencing factor of the statistics of these two word types as follows:

$$
\text{Sta}(\text{wt}_1, \text{wt}_2) = \log_e(f^\text{NtLCA}(\text{wt}_1, \text{wt}_2) + f^\text{NtLCA}(\text{wt}_1, \text{wt}_2)) \ast R
$$

where $R$ is a reduction factor which will be explained shortly. $f^T_{\text{wt}, \text{wt}}$ is the number of T-typed nodes that contain the keyword $w$ with the word type $\text{wt}$ in their subtrees. The intuition behind the statistics is as follows:

*The more XML nodes of node type $\text{NtLCA}(\text{wt}_1, \text{wt}_2)$ contain the keywords with word types $\text{wt}_1$ and $\text{wt}_2$, the more likely these two word types are related through the nodes of node type $\text{NtLCA}(\text{wt}_1, \text{wt}_2)$ and this kind of relationship between $\text{wt}_1$ and $\text{wt}_2$ is desired by the user.*

We now explain the reduction factor $R$. Due to the tree structure, the number of nodes at higher levels is usually significantly less than the number of nodes at lower levels. This may bring unfairness when collecting statistics. Fewer nodes at higher levels are usually caused by design, which may not reflect the real occurrences of data. For example, some types of nodes are moved to higher levels to avoid redundancies. Therefore, we put a reduction factor of depth to the formula to reduce the unfairness. A straightforward reduction factor of depth is $\frac{1}{\text{Dep}(\text{NtLCA}(\text{wt}_1, \text{wt}_2))}$, but it reduces too much and too quickly when the depth increases. Instead, we use the following formula as the reduction factor:

$$
R = \sqrt{\frac{1}{\text{Dep}(\text{NtLCA}(\text{wt}_1, \text{wt}_2))}}
$$

With the two influencing factors above, the proximity score between two word types is defined as follows:

$$
P(\text{wt}_1, \text{wt}_2) = p^{\text{Dis}(\text{wt}_1, \text{wt}_2)} \ast \text{Sta}(\text{wt}_1, \text{wt}_2)
$$

In the experiments, we found that $p = 0.87$ yielded good retrieval results for most of the queries.
Desired word types: Given two adjacent keywords \( w_i \) and \( w_{i+1} \), the word types that achieve the largest proximity score among all \( P(wt_i, wt_{i+1}) \), where \( wt_i \in WT_i \) and \( wt_{i+1} \in WT_{i+1} \), are the desired word types of \( w_i \) and \( w_{i+1} \). In the following, the largest proximity score between \( w_i \) and \( w_{i+1} \) will be denoted \( HP(w_i, w_{i+1}) \), i.e.,

\[
HP(w_i, w_{i+1}) = \max\{P(wt_i, wt_{i+1}) \mid wt_i \in WT_i, wt_{i+1} \in WT_{i+1}\}
\] (4.12)

4.4.3 Inferring Likely Search Intentions

In this section, we show how to infer a set of likely search intentions of a query. In Section 4.4.3.1, we show how to infer the search intentions from a query that contains only one keyword. In Section 4.4.3.2, we explain the way of inferring likely search intentions when a query contains two or more keywords.

4.4.3.1 One Keyword

If a query contains only one keyword, each word type of this keyword is considered as a search intention. We use the following formula to compute the confidence that the word type \( wt \) is desired by the user.

\[
C(wt) = \log_e(f^wt_{k,wt}) \cdot \sqrt{\frac{1}{\text{Dep}(wt)}}
\] (4.13)

In the formula above, \( f^wt_{k,wt} \) is the number of nodes that contain the keyword \( k \) with the word type \( wt \). The returned word types are sorted in descending order of the confidence score of each word type.

4.4.3.2 Two or More Keywords

When there are two or more keywords, we use a pair-wise comparison strategy (PCS) to infer a set of likely search intentions. We first explain the ideas in Sections 4.4.3.2 and 4.4.3.2. Then we present the detailed algorithms in Section 4.4.3.3.

Inferring one likely search intention

We first use an example to explain the basic idea.

Example: Consider the query \{issue 16 database\} over the data tree in Figure 4.9. PCS compares the largest proximity score \( p_1 \) between “issue” and “16” with the largest
4. XML KEYWORD QUERY PROCESSING

proximity score $p_2$ between “16” and “database” (Note: the proximity score between “issue” and “database” is not considered because these two keywords are not at adjacent positions in the query). If $p_1$ is larger than $p_2$, PCS first infers the word types of “issue” and “16” as $wt_{issue}$ and $wt_{16}$ where $P(wt_{issue}, wt_{16})$ (see Formula (4.11)) is equal to $HP(issue, 16)$ (see Formula (4.12)). Then PCS uses the inferred word types of “issue” and “11” to infer the word type of “database” which is most closely related to them.

From the example above, we can see that PCS compares the proximity scores of pairs of adjacent keywords to find a likely search intention. The details are described below.

Given the keywords $\{w_1, \cdots, w_n\}$ ($n > 1$) and their corresponding word type sets $\{WT_1, \cdots, WT_n\}$, we infer a likely search intention as follows.

In the case $n = 2$, i.e., there are only two keywords $w_1$ and $w_2$ in the query, we will compute the largest proximity score between $w_1$ and $w_2$ (i.e., $HP(w_1, w_2)$) using formula (4.12), and choose the word types of $w_1$ and $w_2$ that achieve $HP(w_1, w_2)$ as the word types of $w_1$ and $w_2$.

In the general case of $n > 2$, we need to go through several iterations. In the first iteration, we scan the keywords from left to right pair by pair and compute the proximity scores $HP(w_i, w_{i+1})$ ($1 \leq i < n$). If $HP(w_1, w_2) > HP(w_2, w_3)$, then $w_1$ and $w_2$ will be put into the same group; otherwise if $n = 3$, we will put $w_2$ and $w_3$ into a group; if $n > 3$ we will compare the next two pairs $(w_2, w_3)$ and $(w_3, w_4)$. This scan of pairs of keywords continues until either (A) we find $i$ such that $HP(w_{i-1}, w_i) \leq HP(w_i, w_{i+1})$ and $HP(w_i, w_{i+1}) > HP(w_i, w_{i+2})$, in which case we will put $w_i, w_{i+1}$ into a group; or $HP(w_{n-2}, w_{n-1}) \leq HP(w_{n-1}, w_n)$, in which case we will put $w_{n-1}$ and $w_n$ into a group.

Whenever a pair of adjacent keywords $w_i, w_{i+1}$ are put into a group, the word types of $w_i$ and $w_{i+1}$ that achieve $HP(w_i, w_{i+1})$ will be chosen as the word types of $w_i$ and $w_{i+1}$ respectively, and we will go to the next iteration. In each subsequent iteration, we scan the keywords or groups of keywords from left to right, grouping a pair of two keywords, or a keyword and a group, into the same group by using the largest proximity score in a way similar to the first iteration, except that for each group (that contains more than one keyword), a unique word type of each keyword in the group has been chosen as the word type for that keyword (and this unique word type will not be changed later), thus the computation of the largest proximity score between a keyword and a group, or two
groups will use the word types of each keyword in the group as shown in the following formulas.

For a keyword \( w_i \) and a group \( g = \{ w_k, \ldots, w_s \} \) of keywords, suppose the desired word types of \( w_k, \ldots, w_s \) are \( wt_k, \ldots, wt_s \) respectively, then

\[
\text{HP}(w_i, g) = \text{HP}(g, w_i) = \max\{P(wt_i, wt) \mid wt_i \in WT_i, wt \in \{wt_k, \ldots, wt_s\}\} \tag{4.14}
\]

For two groups of keywords \( g_i \) and \( g_j \) (each group contains more than one keyword)

\[
\text{HP}(g_i, g_j) = 0 \tag{4.15}
\]

The iteration ends when every keyword is grouped with some others, and while a keyword is grouped with other keywords or groups, its desired word type will be determined.

**Discussion** PCS preferentially considers the relationships between adjacent keywords, but it does not completely ignore the influence from the nonadjacent keywords. It considers the relationships between nonadjacent keywords when inferring the desired word types of a group and a keyword by computing the proximity score between them (see Formula (4.14)). Similar to query segmentation (see Chapter 2), our work also groups the keywords but the idea is different. Our approach is mainly based on user’s typing habit in constructing keyword queries and groups the keywords by evaluating the relationship between adjacent keywords. Besides the statistics of data, our approach also takes into account the structural information of data, but this is not considered by existing query segmentation approaches because they mainly handle plain documents.

**Inferring a set of likely search intentions** In practice, it is much safer to return several likely search intentions instead of one to reduce the possibility of missing the real search intention.

**Approach** Given the keywords \( \{ w_1, \ldots, w_n \} \) and their word type sets \( \{ WT_1, \ldots, WT_n \} \), finding likely search intentions is an iterative process. In each iteration, we treat a different word type in \( WT_i \) as the only word type in this set, and use PCS to infer a search intention which is considered as a likely search intention. This iterative process ends until all of the word types in \( WT_i \) have been processed.
4. XML KEYWORD QUERY PROCESSING

The thought behind the approach above is to use PCS to infer a likely search intention for each word type. Note that there may exist duplicates in the inferred likely search intentions. The total number of search intentions found by the approach above is theoretically \( \sum_{1 \leq i \leq n} |WT_i| \) if there are no duplicates, which is much smaller than \( \prod_{1 \leq i \leq n} |WT_i| \) (the number of all possible search intentions). This can keep the size of the set of likely search intentions small, so the user needs to spend little time on browsing the results. Applying PCS against each word type guarantees a good coverage of different word types in the query. In other words, a word type exists in at least one likely search intention. The user can always find the desired word type of a keyword from the inferred likely search intentions.

### Ranking likely search intentions

In order to rank the inferred likely search intentions, each likely search intention should have a ranking score. In our system, the ranking score of a search intention inferred by PCS is not a single number but a sequence of proximity scores, which is defined as follows.

**Definition 11 (Ranking Score)** Given a keyword query \( K = \{w_1, \cdots, w_n\} \) \((n \geq 2)\) and one search intention \( si = (wt_1, \cdots, wt_n) \) inferred by PCS, the ranking score of \( si \) is \( R = (r_1, \cdots, r_{n-1}) \), where \( r_i \geq r_{i+1} \) \((1 \leq i < n - 1)\), and \( r_i \) \((1 \leq i \leq n - 1)\) is a largest proximity score between two keywords or one keyword and one group that are grouped together in PCS (calculated by formula HP), or the proximity score between the inferred word types of two adjacent keywords that sit in different groups (calculated by formula P, see Formula \((4.11)\)). \( r_i \) is called a component of \( R \). Given two ranking scores \( A = (a_1, \cdots, a_n) \) and \( B = (b_1, \cdots, b_n) \), \( A \) is smaller than \( B \) iff the first \( a_i \) which is different from \( b_i \) is smaller than \( b_i \).

**Example** Suppose a keyword query \( \{a_1, a_2, a_3, a_4\} \) is processed by PCS. The keywords are grouped as \( \{(a_1, a_2), (a_3, a_4)\} \) and the inferred search intention is \( si = (wt_1, wt_2, wt_3, wt_4) \). The ranking score of \( si \) is a sequence of proximity scores \( HP(a_1, a_2), HP(a_3, a_4) \) and \( P(wt_2, wt_3) \), which will be sorted in descending order. The ranking score \((2.5, 2, 1.2)\) is larger than \((2.5, 1.8, 1)\).  

### Filtering unreasonable search intentions

Sometimes, unreasonable search intentions may still be found. There are mainly two types of unreasonable search intentions:

---

\(^1\)The word types of the keywords \( a_2 \) and \( a_3 \) have been determined during grouping, so the proximity score between \( a_2 \) and \( a_3 \) is the proximity score between the determined word types of \( a_2 \) and \( a_3 \), which is calculated by formula P.
First, if two identical word types in a search intention are separated by some other word types, this search intention will be filtered out. Consider the query \{issue 16 database\} over the data tree in Figure 4.9. If the system infers “issue” and “database” as the text values in the title of articles but infers “16” as a text value of initPage, this inference should be considered unreasonable because intuitively the user is less likely to separate two keywords of the same word type with another keyword of a different word type.

Second, the word types of two adjacent keywords in a search intention has a very low proximity score. This may happen when a search intention involves more than one entity type and these entity types do not have ancestor-descendant relationships. For example, a likely search intention of a query over DBLP dataset simultaneously involves the entity types inproceedings and article. In this situation, the result subtree is very large and rooted at the root of DBLP dataset (i.e., dblp). This kind of result is difficult to read to find useful information. We observe that under this circumstance the proximity score between the word types of the adjacent keywords that sit in different groups is usually very small. In order to solve this problem, we can simply set a threshold for the component in a ranking score. If a search intention’s ranking score contains a component which is smaller than this threshold, this search intention will be removed. On the other hand, this approach can effectively keep the result subtree compact because all keywords have relatively close relationships. In the experiments, we found that the threshold 1.15 can effectively filter out unreasonable search intentions for most of the queries.

4.4.3.3 Algorithms

We implemented PCS in Algorithm \[\text{12}\]. Before explaining this algorithm, we first present some notation and indexes used in the algorithm.

**Notation.** Given a keyword query \(K = \{w_1, \cdots, w_n\}\), for each keyword \(w_i\), there is a set \(WT_i\) which stores \(w_i\)'s word types to be processed by PCS. For convenience, each keyword has an attribute \(w_i.num\) to record the number of keywords in the group that \(w_i\) sits. If \(w_i\) is not grouped with others, the value of \(w_i.num\) is 1. In addition, the output of Algorithm \[\text{12}\] is list \(l\), which records the word type of each keyword, and

\[\text{For consistency, } WT_i \text{ denotes the set of word types of } w_i \text{ in this paper, but it is implemented with a list in the algorithm.}\]
4. XML KEYWORD QUERY PROCESSING

Algorithm 12 PCS (K,WT,n)

Input: Query K = \{w_1,\ldots, w_n\}, word type sets \{WT_1,\ldots, WT_n\}

Output: a search intention: list l and ranking score r

1: l.clean()
2: r.clean()
3: while ExistUnpairedKeyword(K, n)=true do
4:   PAIRING(0,0,1,0,-1,-1)
5: Add proximity scores between the word types of two adjacent keywords that sit in different groups into r.
6: r.sort(Desc)
7: procedure ExistUnpairedKeyword(K, n)
8:  i = 1
9:  while i ≤ n do
10:    if w_i.num = 1 then
11:      return true;
12:    i = i + w_i.num
13:  return false

14: procedure PAIRING(i, j, largestscore, s, t)
15:  k = j + w_j.num
16:  if k > n then
17:    score = 0
18:  else
19:    (score, x, y) = CalculateHPScore(w_j, w_k)
20:    if score ≥ largestscore then
21:      i = j
22:      largestscore = score
23:      PAIRING (i, k, largestscore, x, y)
24:  else
25:    r.insert(largestscore)
26:    w_i.num = w_i.num + w_j.num
27:    if s ≠ -1 then
28:      l_i = WT_i[s]
29:    if t ≠ -1 then
30:      l_j = WT_j[t]

l’s ranking score r (see Definition 11) which is also a list. l_i denotes the word type of keyword w_i in the inferred search intention.

Indexes We use three types of indexes in the algorithm. The first one is word type index, which is a B+ tree index and used for retrieving all the word types of a keyword. The search key of this index is the keyword. The second one is inverted list index, which is also a B+ tree index and used for retrieving all the DeweyIDs of the nodes that contain a keyword with a certain word type. The key of this index is a combination object (keyword, word type). The third one is data statistics index, which is a hash table and used for storing \(f^T_{k,wt}\) (see Section 4.4.2.2). The hash key is a combination of T, k
4.4 Exploiting User’s Typing Habit and Data Statistics for Effective XML Keyword Search

and $wt$. We report the size of these three indexes in Section 4.7.4.1.

Now we explain Algorithm 12. Line 1-2 empties list $l$ and $r$. Line 3-6 infers a likely search intention for the keyword query $K$. It first realizes PCS at line 3-4. Line 3 checks whether every keyword has been grouped with some others. Procedure PAIRING groups two keywords or one keyword and one group at line 4. Then the proximity scores between the word types of adjacent keywords that sit in different groups are added into list $r$ (line 5). Finally the proximity scores in list $r$ are sorted in descending order (line 6).

Procedure PAIRING (line 14-30) is recursive and the core of the algorithm. This procedure first calculates the value of $k$ (line 15). $w_k$ is a single keyword or the first keyword in its group, and $w_k$ or $w_k$’s group is next to $w_j$ or $w_j$’s group. To facilitate the following discussion, $w_i$, $w_j$ and $w_k$ hereafter denote single keywords or groups. If they denote groups, they are the first keywords in their groups. Then we check whether $k$ is a position out of the set (line 16). If it is true, the new largest proximity score is 0 (line 17). Otherwise, we use procedure CalculateHPScore to calculate the largest proximity score between $w_j$ and $w_k$ (line 19) (see PCS for the way of calculation). The return values of procedure CalculateHPScore is $score$, $x$ and $y$. $score$ is the newly calculated largest proximity score. For the returned values $x$ and $y$, if $w_j$ (or $w_k$) is a group, the value of $x$ (or $y$) will be -1 because the word types of the keywords in $w_j$ (or $w_k$) have been determined before. Otherwise, $WT_j[x]$ and $WT_k[y]$ are the word types of $w_j$ and $w_k$ that achieve $score$. If the newly calculated largest proximity score is larger than $largestscore$, $largestscore$ will be assigned with this new score (line 22), and procedure PAIRING will be recursively called (line 23). Otherwise, $w_i$ and $w_j$ are paired together. The largest proximity score between them is first inserted into list $r$ at line 25. Then the value of $w_i$.num is updated (line 26), where $w_i$ is the first keyword in the new group. Finally the desired word types of $w_i$ and $w_j$ are stored in $l_i$ and $l_j$ (line 27-30). We use the following example to illustrate the process of inferring a likely search intention.

**Example** Suppose the user intends to retrieve the articles written by Karen from volume 11, and he submits the query \{volume 11 article Karen\} over the real Sigmod-Record data set obtained from [92]. To help illustrate, we present the result of each call of procedure PAIRING in Figure 4.11 (including the recursive calls). We also list the word types of each keyword in that figure. Initially, as shown in Figure 4.11(a),
the values of $i$, $j$, and $k$ are 0. The initial value of $largestscore$ is 0. In the first call of $pairing$, the value of $j$ is 1, and the value of $k$ is calculated at line 15, which is 2. Then the procedure $CalculateHPScore$ calculates the largest proximity score between “volume” and “11” (line 19). As shown in Figure 4.11(b), the word types (1) (where they are linked with a line) of “volume” and “11” achieve the largest proximity score. Because this score is larger than $largestscore$, $largestscore$ is assigned with this score (line 22), and the procedure $pairing$ is called again (line 23). In this call, the proximity score between the keywords “11” and “article” is smaller than $largestscore$, so the word types of “volume” and “11” are determined (i.e., word types (1) of “volume” and “11”) (Figure 4.11(c)). The first call of procedure $pairing$ terminates.

Because there still exist unpaired keywords, we need to call $pairing$ again (line 4). The initial state before the second call of $pairing$ is shown in Figure 4.11(d). The $largestscore$ is set to 0. In the second call of $pairing$, the largest proximity score between the group (volume 11) and the keyword “article” is achieved by the word types (1) of the keywords “volume” and “article” (as shown in Figure 4.11(e)). Because this score is larger than $largestscore$, $largestscore$ is assigned with this score. Then procedure $pairing$ is called again. In this call, the largest proximity score between “article” and “Karen” is larger than $largestscore$ (as shown in Figure 4.11(f)), so $largestscore$ is assigned with this new score, and procedure $pairing$ is called again. In this call, the value of $k$ is larger than $n$, so the new largest proximity score is 0 and smaller than
4.4 Exploiting User’s Typing Habit and Data Statistics for Effective XML Keyword Search

Figure 4.12: Two extreme cases

largestscore. Therefore, the terminating condition of recursion is satisfied, the keywords “article” and “Karen” are paired, and their word types are 2 and 1 as shown in Figure 4.11(g). Now, all of the keywords have been paired, and their word types have been determined. ⊥

**Time complexity** Given a keyword query \( K = \{w_1, \ldots , w_n\} \), the worst case time complexity of Algorithm 12 is \( O(n^2|WT_1||WT_2|) \), where \( WT_1 \) and \( WT_2 \) are the sets of word types of the two keywords which have the most word types. The detailed analysis of time complexity is as follows: Computing the largest proximity score between keywords \( w_i \) and \( w_{i+1} \) needs \( O(|WT_i||WT_{i+1}|) \) time. Computing the largest proximity score between a keyword and a group \( k_i \) and \( g_j \) needs \( O(|WT_i||g_j|) \) time. We can use two extreme cases to get the upper bound of the worst case time complexity. The first extreme case is the inference that takes the maximum steps in computing the largest proximity scores of pairs of adjacent keywords. Figure 4.12(a) presents such an extreme case. In this case, the keywords are sorted in descending order of their number of word types, so the first keyword (i.e., \( k_1 \)) has the most word types. The pairing starts from the last two keywords and forms a group. Then in each following pairing operation, one more adjacent keyword is put into this group. Therefore, the steps of computing the largest proximity scores of pairs of adjacent keywords is \( O(n^2|WT_1||WT_2|) \). The second extreme case is the inference that takes the maximum steps of computing the largest proximity scores between keywords and groups. Figure 4.12(b) presents such an extreme case. The pairing starts from the second and third keywords and forms a group. Then in each following pairing operation, one more following adjacent keyword is put into this group. In addition, the first keyword has the most word types among all of the keywords. In this case, the steps of computing proximity scores between keywords and groups is \( O(n^2|WT_1|) \). Therefore, the worst time complexity of Algorithm 12 is \( O(n^2|WT_1||WT_2|) \).

**Algorithm for inferring likely search intentions** The algorithm for inferring the likely search intentions is shown in Algorithm 13. This algorithm returns list \( L \) which
Algorithm 13 XInfer(K,n)

Input: Query K = \{w_1, \cdots, w_n\}, number of keywords n
Output: A list L that contains a set of likely search intentions

1: Initialize L as empty
2: Retrieve the word types of \{w_1, \cdots, w_n\}, and store them in a set of arrays WT: \{WT_1, \cdots, WT_n\}
3: for 1 \leq i \leq n do
4: Block all word types in WT_i
5: for 1 \leq j \leq |WT_i| do
6: Unblock WT_i[j] //WT_i[j] is the j^{th} item in WT_i
7: \((l, r) = \text{PCS}(K, WT, n)\)
8: \(L.insert(l, r)\)
9: Block WT_i[j]
10: Unblock all word types in WT_i
11: Remove duplicates in L
12: Filter out the unreasonable search intentions from L
13: \(L.sort()\)

contains the inferred likely search intentions. Each item in L is comprised of a search intention l and l’s ranking score r. L is first initialized as empty at line 1. Then the algorithm retrieves the word types of each keyword w_i, and stores them in set WT_i at line 2. In order to treat word type WT_i[j] in WT_i as the only word type in this set, we block all word types in WT_i at line 4 and unblock WT_i[j] at line 6. The blocked word types will not be processed in PCS procedure (see Algorithm 12). Note that when calculating the largest proximity score using CalculateHPScore procedure in Algorithm [12], this procedure will check whether a word type is blocked. Procedure PCS returns a search intention l and its ranking score r at line 7. Line 8 inserts this search intention and its ranking score into L. Then WT_i[j] is unblocked at line 9. After every word type in WT_i is processed, all of the word types in WT_i are unblocked at line 10. Then the algorithm will process the word types in WT_{i+1} one by one. After the duplicates are removed from L (line 11), the unreasonable search intentions are filtered out (line 12). The remaining search intentions in L are sorted in descending order of their ranking scores (line 13).

4.4.4 Generating Results

In this section, we explain how to retrieve result subtrees for a set of likely search intentions.
4.4 Exploiting User’s Typing Habit and Data Statistics for Effective XML Keyword Search

Algorithm 14 GenerateResults($L$)

**Input:** A set of likely search intentions $L$

**Output:** result subtrees

1: for each $l$ in $L$ do
2:  $rt = $GetNodeTypeofRoot($l$) // $rt$: node type of the root
3:  $rl = $RetrieveInvertedList(root)
4:  for $1 \leq i \leq ||l||$ do
5:      $IL_i = $RetrieveInvertedList($l_i$)
6:  while $rl$.isEnd() = false do
7:      for $1 \leq i \leq ||l||$ do
8:          $IL_i$.MoveTo($rl$.getCurrent())
9:      if isAncestor($rl$.getCurrent(), $IL_i$.getCurrent()) == false then
10:         break
11:   if $i > ||l||$ then
12:      $resultList$.insert($rl$.getCurrent())
13:   $rl$.moveToNext()
14: Build result subtrees rooted at the nodes in $resultList$ and exclude irrelevant entities.
15: Return result subtrees to the user

We first define the concept of result subtrees. In the definition below, $\text{ENtLCA}(wt_1, \cdots, wt_2)$ denotes the entity type of $\text{NtLCA}(wt_1, \cdots, wt_2)$.

**Definition 12 (Result Subtree)** Given a keyword query $K = \{w_1, \cdots, w_n\}$ and an inferred likely search intention $\{wt_1, \cdots, wt_2\}$, a subtree of $t$ is a result subtree iff: (1) its root has the node type $\text{ENtLCA}(wt_1, \cdots, wt_2)$, (2) it contains all of the keywords in $K$, and the contained occurrences of keyword $w_i$ has the word type $wt_i$.

The purpose of using $\text{ENtLCA}$ instead of $\text{NtLCA}$ in the definition above is to make the returned result subtrees more informative and meaningful.

**Excluding irrelevant entities** Sometimes, a retrieved result subtree may contain lots of irrelevant information. Consider the query $\{\text{issue 16 database}\}$. Suppose our approach is to return the result subtrees rooted at $\text{issue}$ nodes. However, if we return the whole subtrees rooted at $\text{issue}$ nodes, a number of articles that are not related to database are also returned to the user. Therefore, we should exclude these irrelevant entities. In order to achieve this goal, we first find the entity-type of each keyword’s word type, then we know the keywords and their word types that an entity should contain. If an entity does not contain the keywords with the inferred word types, it will be excluded. In the example above, an article entity should contain the keyword “database” with the word type No.5 in Table 1. The entities that do not satisfy this condition will be excluded.
4. XML KEYWORD QUERY PROCESSING

**Algorithm** The algorithm for generating results is presented in Algorithm 14. In the algorithm, given an inverted list $IL_i$ of DeweyIDs, we use $IL_i.getCurrent()$ to return the current item in $IL_i$ for processing, which is pointed by a cursor. $IL_i.moveToNext()$ advances the cursor to the next item. Function isAncestor($a, b$) is used to check whether node $a$ is an ancestor of $b$.

For each likely search intention in $L$, this algorithm first computes the node type of the root at line 2, then it retrieves a list of nodes represented by DeweyIDs which contains the roots of the computed node type at line 3. Note that this list can be efficiently retrieved through the inverted list index. Then line 4-5 retrieves the inverted list for each keyword $w_i$ and its word type $l_i$, and stores the retrieved list in $IL_i$. Line 6-14 tests whether the subtrees rooted at the nodes in $rl$ contain all of the keywords with the inferred word types. For each root in $rl$, line 8 tries to move the cursor of $IL_i$ to the first node that is a descendant of $rl.getCurrent()$ or on the right of $rl.getCurrent()$ (if there is no descendant of $rl.getCurrent()$). If $i$ is larger than the size of $l$, the subtree rooted at $rl.getCurrent()$ contains all of the keywords of the inferred word types. Therefore, line 12 inserts $rl.getCurrent()$ into the resultList.

**Displaying result subtrees** Because all the result subtrees of a search intention have the same semantics, we cluster the result subtrees according to the search intentions, and first compute and show one result subtree for each search intention. The user can easily determine which search intentions are what he wants by viewing the initially displayed results, then he can click expansion links to get more results. Note that only when the expansion link is clicked, the remaining result subtrees of the desired search intention are computed. The benefits of displaying results in this way are (1) it takes very little time to return initial results to the user, (2) the system does not waste time on computing the result subtrees with undesired search intentions.

**Implementation:** Algorithm 14 generates all the result subtrees of a search intention at one time. In our system, we first display one result subtree of each search intention to the user for verification, so we need to make a few minor adjustments to the algorithm as follows: (1) The algorithm will terminate when one result subtree of each search intention is found. (2) The cursors of the inverted lists need to be saved for retrieving the remaining result subtrees in the future. We will use the saved cursors to find the remaining result subtrees when the user clicks the expansion link.
4.5 XQSuggest: an Interactive XML Keyword Search System

4.5.1 Introduction

Accurately acquiring the user’s real intention is a difficult task because the keywords are inherently ambiguous. We use the following example to illustrate this problem.

**Example** Suppose a user is interested in the population of all countries, and he submits the query \{country, population\} over the data tree in Figure 4.13 which comes from Mondial data set [92]. However, every province and city element also has country and population attributes. In other words, the keywords country and population appear in different types of nodes. The systems cannot know exactly which one is desired by the user just from the keywords. As a result, the irrelevant answers province (0.4) and city (0.4.4) will also be returned. ⊥
4. XML KEYWORD QUERY PROCESSING

The example above shows the weakness of existing XML keyword search systems on returning relevant results when the submitted keywords have multiple meanings. Our approach in Section 4.2 cannot solve this problem either because it aims to make the results informative and reduce irrelevant information in the results. In practice, most users of keyword search would roughly know the contents of the data and the meaning of each keyword they pose. They want to unambiguously express their needs but the keywords alone cannot help them to do so. In this section, we propose to solve the problem of ambiguity using query suggestion: When a keyword is typed in, the system can instantly suggest several understandable and distinct semantic strings. Then the user can select one to replace the ambiguous keyword. Figure 4.14 shows a screen shot of query suggestion in our interactive XML keyword search system XQSuggest.

4.5.2 Query Suggestion

4.5.2.1 Semantic String

The basic idea of query suggestion is to replace each keyword with a semantic string, which has a more specific meaning than the keyword. Here we first define the entity node. Then we precisely define the semantic strings associated with a keyword.

**Entity nodes** In reality, an XML document is usually a container of related entities. For instance, Figure 4.13 is a collection of country, province and city entities. We use an approach similar to that of [65] to identify entity nodes.

**Definition 13 (Entity Node)** Let \( t \) be a data tree. A node \( v \) in \( t \) is said to be a simple node if it is a leaf node, or has a single child which is a leaf node. A node \( v \) is said to be an entity node if: (1) it corresponds to a \( * \)-node in the DTD (if DTD exists), or has siblings with the same tag name as itself (if DTD does not exist), and (2) it is not a simple node. The entity type of an entity node \( e \) refers to the node type of \( e \). A node \( v \) in \( t \) is called a grouping node if it has only children of entity nodes.
Definition 14 (Semantic String) Let \( t \) be the data tree and \( k \) be a keyword that occurs in \( t \). A semantic string of \( k \) is a colon-separated sequence of labels \( l_1 : l_2 : \cdots : l_n \) such that there is a path \( u_1 . u_2 . \cdots . u_n \) in \( t \), where \( u_n \) is a node whose label or value contains \( k \), \( u_1 \) is the only entity node in the path (in other words, either \( u_n \) is an entity node and \( u_1 = u_n \), or \( u_1 \) is the closest entity node above \( u_n \)), and \( l_i \) is the label of \( u_i \) for \( i \in [1, n] \). The length of a semantic string is the number of labels in the path. The type of a semantic string is the same as the entity-type of its corresponding entity node.

In some cases, a keyword can exist in both the element name and the value. In order to differentiate these two cases, if the keyword appears in the value, it will be double quoted in the semantic string. We call semantic strings that have double quoted keywords predicates.

Example For the data tree in Figure 4.13, the semantic strings of the keyword “population” are \textit{country:population}, \textit{province:population} and \textit{city:population}.

Note: Each keyword occurring in the data tree has at least one semantic string. For the keyword in the label name of an entity node, its semantic string is the label name.

4.5.2.2 Boolean Operators

Sometimes, the user will specify more than one predicates within the same context. For the system, it is very difficult to judge the relationship between these predicates. Therefore, if the system allows the user to specify boolean operators (AND or OR) between semantic strings, the users will be able to express their queries more clearly. In our current implementation, the boolean operators can only be placed between the predicates of the same entity-type. If there are more than one predicates of the same entity-type in a query but no boolean operators are specified, the default boolean operator AND will apply.

4.5.2.3 Result of Query

When the ambiguous keywords are replaced with the semantic strings, each keyword has an exact meaning and is associated with a specific entity-type. We call the new query a \textit{semantic string query}. Each semantic string will appear within some entity nodes, and we can join such entity nodes together as the result of the query. Note: the “join” here means concatenating entity nodes where the semantic strings appear and
where an ancestor-descendant relationship exists. A more precise definition of a query result is given in Definition 15.

In the following, a subtree \( T \) of data tree \( t \) refers to a tree that can be obtained from \( t \) by erasing some entity nodes and all of their descendants. We say a semantic string \( s \) appears in an entity node \( v \in T \) (and \( v \) contains \( s \)) if there is a path from \( v \) which is isomorphic to \( s \). We say \( s \) appears in subtree \( T \) (and \( T \) contains \( s \)) if \( s \) appears in some entity node in \( T \).

**Definition 15** A result of a semantic string query \( K \) is a subtree \( T \) of \( t \) with the following properties: (1) the root of \( T \) is an entity node in \( t \), (2) every entity node in \( T \) contains at least one semantic string, (3) \( T \) contains all semantic strings in \( K \), but no lower subtree of \( T \) contains all semantic strings in \( K \), (4) if two predicates of the same entity-type have logic AND relationship, they must appear in the same entity node. We call such a result tree a joined entity-node tree (JET).

4.5.3 Algorithm

4.5.3.1 Notation

Given a semantic string query \( K = \{s_1, ..., s_n\} \), for each semantic string \( s_i \), there is a stream, \( S_i \), consisting of all the nodes which have the corresponding semantic string \( s_i \). The nodes (i.e., the Dewey codes) in each stream \( S_i \) are arranged in lexicographic order. The system needs to instantly suggest semantic strings after a keyword is typed in, so the performance of retrieving semantic strings of a keyword is very important. We build a \( B^+ \)-tree on all of the words in \( t \), each word \( k_i \) in the leaf of the \( B^+ \)-tree points to the list of the corresponding semantic strings. It should be noted that there do not exist duplicates in a list.

The results of the algorithm are stored in the list \( RL \). Each item in the list is a pointer which points to a list of entity nodes, which represents a JET to be returned to the user.

In order to facilitate keyword containment check and logical relationship check, we introduce four additional attributes \( flag, pattern, isEntity \) and \( logic \) for each semantic string during processing. The \( flag \) attribute is a \( n \)-bit binary number, which indicates which keywords are contained in the sub-tree rooted at a node. The \( pattern \) attribute is also a \( n \)-bit binary number, which presents which predicates of the same semantic
string type are involved into a boolean formula. If the \( i^{th} \) bit is set to 1, it means the semantic string \( s_i \) is involved into the boolean formula within the corresponding context.

We use the function \( \text{GetMinDewey}(S_i) \) to get the smallest Dewey code (in lexicographic order) among the streams of semantic strings. The function \( \text{GetClosestEntityCode} \) returns the Dewey code of the entity node associated with a semantic string, given the Dewey code and length of the semantic string.

### 4.5.3.2 The Stack-based Algorithm

Our stack-based algorithm is shown in Algorithm [15].

Now, we explain the algorithm for evaluating queries. As mentioned earlier, we represent results using the joined tree of entity nodes (JET) associated with each semantic string, so after the minimal Dewey code of a semantic string is retrieved (line 5), the Dewey code of its entity node should be computed immediately for further processing (line 6), and this can be easily achieved with the length of the semantic string. After a node is popped up from the stack \( DS \), if this node is an entity node, the algorithm first needs to perform a boolean logic check. If the desired boolean logic between the predicates of the same entity type is OR and any predicate is satisfied (line 12-13), the algorithm will modify the flag value to indicate all of the predicates are satisfied (line 14). After that, the algorithm will check whether it contains all the keywords (line 15). If a node is an entity node but does not contain all the keywords or it is not an entity node, the algorithm needs to copy the flag value to the top entry of the stack (line 20 and 22). Whether the unprocessed entity nodes exist is determined by the value of \( DS.\text{NumOfEntityNode} \), which is maintained by the algorithm to indicate how many entity nodes associated with the selected semantic strings exist in the stack \( DS \) (line 11 and 30).

### 4.5.3.3 Optimization Techniques

The performance of query evaluation can be seriously affected by the frequency of the keywords. If the keyword in the query is contained in the element names, there can be a large number of match nodes to be processed. First, given a query \( K = \{s_1, \ldots, s_n\} \), if \( s_m \) is a sub-string of \( s_n \), the \( s_m \) can be removed from \( K \). This is because \( s_n \) can guarantee the satisfaction of \( s_m \). Second, given a query \( K = \{s_1, \ldots, s_n\} \), for the
4. XML KEYWORD QUERY PROCESSING

Algorithm 15 EvaluateQuery(K, S)

Input: Query K = \{s_1, ..., s_n\}, a set S of streams
Output: Result List RL

1: Create an empty result list RL
2: Create an empty list ListItem
3: Create an empty stack DS
4: while DS ≠ ∅ OR ¬isEnd(S_1) ∧ ... ∧ ¬isEnd(S_n) do
5: \(s_{\text{min}} = \text{GetMinDewey}(S_i)\)
6: \(e_{\text{min}} = \text{GetCloestEntityCode}(s_{\text{min}}, S_{\text{min}}, \text{SemanticStringSize})\)
7: Get longest common prefix lcp such that DS[i].id = \(e_{\text{min}}[i]\) (1 ≤ i ≤ lcp)
8: while DS.size > lcp do
9: \(\text{stackentry} = \text{DS.pop}()\)
10: if stackentry.isEntityNode = true then
11: DS.NumofEntityNode −−
12: if stackentry.logic = OR then
13: if stackentry.flag & stackentry.pattern ≠ 0 then
14: stackentry.flag = stackentry.flag | stackentry.pattern
15: if ContainAllKws(stackentry) then
16: Append current Dewey code in stack DS into the list ListItem
17: RL.append(ListItem)
18: Empty the list ListItem
19: else
20: CopyFlags(DS, stackentry)
21: else
22: CopyFlags(DS, stackentry)
23: for i = lcp + 1 to \(e_{\text{min}}.\text{length}\) do
24: DS.push(\(e_{\text{min}}[i]\))
25: if DS is not empty then
26: \(\text{SetFlag}(DS[DS.size].\text{flag}, \text{min})\)
27: DS[DS.size].pattern = \(S_{\text{min}}.\text{pattern}\)
28: DS[DS.size].logic = \(S_{\text{min}}.\text{logic}\)
29: if lcp = \(e_{\text{min}}.\text{length}\) then
30: DS.NumofEntityNode ++
31: procedure CopyFlags(DS, child)
32: if DS.NumofEntityNode > 0 then
33: if child.flag & child.pattern = child.pattern then
34: DS[DS.size].flag = child.flag
35: if child.isEntityNode = true then
36: Append current Dewey code in stack DS into the list ListItem
4.6 Comparison of Approaches

semantic strings with the same type, the algorithm can remove the semantic strings of the element or attribute, which at least occur once as a child of another element. This can be achieved by exploring the DTD file. The two optimization techniques work well because the time spent on processing semantic strings before query evaluation is much less than processing the keyword match nodes with a high frequency.

4.5.4 Summary

In this section, we explored the application of query suggestion in XML keyword search. Our system XQSuggest suggests several semantic strings after each keyword is typed in, which significantly reduces keyword ambiguity and facilitates the use of Boolean operators. We proposed an algorithm to find the results of the transformed query. Two optimization techniques were used in our algorithm in order to improve performance. The experiments verified better effectiveness of our approach.

4.6 Comparison of Approaches

In this chapter, we proposed four approaches to improve the effectiveness of XML keyword search. XKMis in Section 4.2 has a different focus from the other three approaches. It tries to make the results more meaningful and informative, and reduce the irrelevant information in the result subtrees. In other words, it improves the quality of results. However, it does not analyze the user’s real search intention. The other three approaches try to resolve the ambiguity of keywords, but they have totally different directions. XQSuggest in Section 4.5 is a helper tool in practice. It let users clarify their search intentions themselves, so it needs the user to roughly know the dataset that he is querying and the meaning of the keywords he will submit. This is not a big problem in the real world because most serious users know what he is doing and what he wants. More importantly, with a little training the user will know how to use it. The approach in Section 4.3 is an improvement of XReal on inferring the search-for node type (SNT), but it can also act as a basis of other work. Precisely inferring the SNT can significantly narrows the search space so as to save the user’s time. In practice, the data set is always very large, so a query may result in the return of a large number of results. Therefore, effectively reducing the search space will decrease the processing time as well as the user’s browsing time. The third approach for resolving the ambiguity of keywords and
inferring the user’s real search intention is by exploiting the user’s typing habit (Section 4.4). Most previous approaches infer the user’s search intention just by analyzing the data (e.g., the statistics of data), but this is not enough. We also need to analyze the keyword query because different sequences of the keywords often have different intentions. Therefore, we formulate the user’s typing habit in XInfer. Compared with our work in Section 4.3, XInfer is a relatively complete XML keyword search engine that infers the user’s real search intention by considering both the keyword query and the data.

4.7 Experiments

In this section, we present the experiment results on evaluating our four approaches XKMis (Section 4.2), DynamicInfer (4.3), XInfer (Section 4.4) and XQSuggest (Section 4.5).

4.7.1 Experiment Set-up

The XML document parser we used is the XmlTextReader Interface of Libxml2 [90]. All the algorithms were implemented in C++. The executable file of MaxMatch is kindly provided by its authors. All the experiments were performed on an Intel Pentium-M 1.7G laptop with 1G RAM. We used the following four data sets that are obtained from [92] for evaluation:

DBLP: The structure of this data set is wide and shallow. It has many different types of entities. Many words in this data set have multiple word types.

SigmodRecord: This data set has a little more complicated structure than DBLP, but has less entity types. Figure 4.9 provides a similar sample.

WSU: Similar to DBLP, the structure of this data set is also wide and shallow. However, it has only one entity type and many words have only one word type.

Mondial: This data set has a complicated structure. A keyword may have multiple word types.

4.7.2 XKMis vs. XRank, SLCA and MaxMatch

In this section, we present the experiment results on the efficiency and effectiveness of our approach against XRank [58], SLCA [59] and MaxMatch [60].
4.7 Experiments

<table>
<thead>
<tr>
<th>Name</th>
<th>Dataset</th>
<th>Keyword Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>QW1</td>
<td>WSU</td>
<td>ACCTG</td>
</tr>
<tr>
<td>QW2</td>
<td>WSU</td>
<td>CAC, 101</td>
</tr>
<tr>
<td>QW3</td>
<td>WSU</td>
<td>instructor, MCELDOWNEY</td>
</tr>
<tr>
<td>QW4</td>
<td>WSU</td>
<td>bldg, TODD, course, ECON</td>
</tr>
<tr>
<td>QW5</td>
<td>WSU</td>
<td>course, ACCTG, times, place</td>
</tr>
<tr>
<td>QW6</td>
<td>WSU</td>
<td>course, ECON, crs, title, days</td>
</tr>
<tr>
<td>QW7</td>
<td>WSU</td>
<td>course, crs, sect, title, credit, days, times, place</td>
</tr>
<tr>
<td>QS1</td>
<td>SigmodRecord</td>
<td>Karen, Anthony</td>
</tr>
<tr>
<td>QS2</td>
<td>SigmodRecord</td>
<td>Anthony, Data, Design</td>
</tr>
<tr>
<td>QS3</td>
<td>SigmodRecord</td>
<td>Stephen, Database</td>
</tr>
<tr>
<td>QS4</td>
<td>SigmodRecord</td>
<td>volume, 11, article</td>
</tr>
<tr>
<td>QS5</td>
<td>SigmodRecord</td>
<td>article, Data, System</td>
</tr>
<tr>
<td>QS6</td>
<td>SigmodRecord</td>
<td>database, author, Tom</td>
</tr>
<tr>
<td>QS7</td>
<td>SigmodRecord</td>
<td>issue, volume, article, title, initpage, endpage, author</td>
</tr>
<tr>
<td>QM1</td>
<td>Mondial</td>
<td>Europe</td>
</tr>
<tr>
<td>QM2</td>
<td>Mondial</td>
<td>Continent</td>
</tr>
<tr>
<td>QM3</td>
<td>Mondial</td>
<td>Tirane, population</td>
</tr>
<tr>
<td>QM4</td>
<td>Mondial</td>
<td>organization, name, members</td>
</tr>
<tr>
<td>QM5</td>
<td>Mondial</td>
<td>city, name, longitude, latitude</td>
</tr>
<tr>
<td>QM6</td>
<td>Mondial</td>
<td>country, name, population</td>
</tr>
<tr>
<td>QM7</td>
<td>Mondial</td>
<td>country, name, capital, population, datacode, government</td>
</tr>
<tr>
<td>QD1</td>
<td>DBLP</td>
<td>Automated Software Engineering</td>
</tr>
<tr>
<td>QD2</td>
<td>DBLP</td>
<td>Han data mining</td>
</tr>
<tr>
<td>QD3</td>
<td>DBLP</td>
<td>WISE 2000</td>
</tr>
<tr>
<td>QD4</td>
<td>DBLP</td>
<td>Jeffrey XML</td>
</tr>
<tr>
<td>QD5</td>
<td>DBLP</td>
<td>author Jim Gray</td>
</tr>
<tr>
<td>QD6</td>
<td>DBLP</td>
<td>Relational Database Theory</td>
</tr>
<tr>
<td>QD7</td>
<td>DBLP</td>
<td>article spatial database</td>
</tr>
</tbody>
</table>

Table 4.5: Keyword queries

The approaches are evaluated with the following metrics: (1) processing time, (2) scalability on the document size, (3) effectiveness based on precision, recall and F-measure.

We used the data sets WSU, SigmodRecord, Mondial and DBLP for evaluation. In order to get a larger data size, we replicated the first three data sets 20 times, and the sizes of them are 31M, 9M and 34M respectively.

We selected seven keyword queries for each data set. The queries are listed in Table 4.5.

4.7.2.1 Processing time

We compared the processing time of $XKMis$ with $XRank$ and $SLCA$, which have better efficiency than $MaxMatch$ [66]. The comparison results are illustrated in Figure 4.15.
As shown, \texttt{XKMis} outperforms \texttt{XRank} in all queries even though it needs to construct MIS subtrees and takes partial match results into consideration. Both \texttt{XKMis} and \texttt{XRank} are stack-based algorithms. The improvements on performance mainly come from the following. First, \texttt{XKMis} is not a LCA-based approach, so much time is saved on the computation of LCAs. Second, \texttt{XKMis} uses region-based coding. Compared with the Dewey coding, it will be more efficient to determine the ancestor-descendant relationship. In addition, our approach has much less entries pushed/popped-up into/from the stack. Given a keyword query, there is a great chance that several keywords appear in the same MIS, and our approach guarantees these same MISs are pushed into the stack only once. In contrast, each keyword match node in \texttt{XRank} may push several entries into the stack. For example, suppose a keyword match node is coded with 0.1.1. Three entries (i.e. 0, 1, 1 with some other attributes) will be pushed into the stack. The more keywords appear in the same MIS, the more time will be saved on stack operations. For example, the query QW7 and QS7 have the most keywords among the queries and these keywords are very likely to appear in the same MIS. As shown in Figure 4.15, the improvement on performance of these two queries is greater than for other queries.

\texttt{XKMis} significantly outperforms \texttt{SLCA} in most queries even though some time is spent on constructing MIS subtrees. However, in some queries (e.g., QS3, QS6 and...
4.7 Experiments

![Graph (a) QW3](image)

![Graph (b) QW7](image)

**Figure 4.16:** Processing time with increasing document size

QS7), SLCA achieves better performance than both XKMis and XRank. This is mainly because of the **false negative** problem of SLCA. Some valid results are ignored by the algorithm.

### 4.7.2.2 Scalability

We compare the scalability of XKMis, XRank and SLCA using the queries over the data set WSU. In order to make the evaluation more accurate, we selected a short query QW3 which includes two keywords and a long query QW7 which has eight keywords. The parameter of scalability selected for evaluation is the document size. We replicated the WSU data set of size 1.6M between 1 and 8 times to get increasingly larger data sets. The processing time of query QW3 and QW7 over these data sets are shown in Figure 4.16. It can be seen that the processing time of these three approaches increases linearly when the document size increases. XKMis and XRank have the similar increasing speed. SLCA increases the fastest among the three approaches.

### 4.7.2.3 Effectiveness

We evaluate the effectiveness of XKMis, XRank, SLCA and MaxMatch based on precision, recall and F-measure. These measures are extensively used in IR research to evaluate the relevancy of the results. Precision measures the percentage of retrieved results desired by users. Recall is the probability that a relevant result is retrieved by the query. F-measure is the weighted harmonic mean of precision and recall. They are defined as follows:

\[
Precision = \frac{|Rel \cap Ret|}{|Ret|}
\]  

(4.16)
4. XML KEYWORD QUERY PROCESSING

\[ \text{Recall} = \frac{|\text{Rel} \cap \text{Ret}|}{|\text{Rel}|} \]  \hspace{1cm} (4.17)

\[ F = \frac{(1 + \alpha) \times \text{Precision} \times \text{Recall}}{\alpha \times \text{Precision} + \text{Recall}} \]  \hspace{1cm} (4.18)

Here, Rel is the set of relevant results that should be found (We transformed the keyword query to XQuery to get such results), Ret is the set of results actually retrieved using the keyword search system. For F-Measure, the value of \( \alpha \) is normally set to 2, which means precision and recall are evenly weighted.

As shown in Figure 4.17, XKMis generally achieves higher precision than XRank, SLCA and MaxMatch. However, the precisions of some queries are not high (e.g., QW5, QW6, QW7, QS4 and QM3). This is mainly because the user is not interested in all the information contained in a MIS. For example, for the query QM3, the user is only interested in the population of the city Tirane, but our approach also returns some other information of Tirane. Actually, this kind of results is acceptable. First, the size of a MIS is not big, so it will not bring difficulties on finding the desired information with the help of highlight. Second, the extra information is still about the same MIS, not others, so it is understandable and will not bring confusion to users. The precision
of the query QM6 is not high either. It is because the keywords “country”, “name” and “population” widely exist in the MISs of city as well as the MISs of country. However, these MISs of city are not desirable. Actually, this kind of problem can be properly solved in our approach. We can further classify the results according to their MIS-types. The users should know which MIS-type they really want. In this case, the user wants the MISs of country, so he can choose the MISs of country to be shown by the system. The MISs of city are effectively filtered out. We leave this as part of our future work. The precisions of many results returned by XRank are very low (e.g., QW2, QW4, QS1, QS2 and QS5). This is mainly because large trees may be returned due to the false positive problem. For example, for the query QS1, two articles in different issues written by Karen and Anthony respectively are connected via the LCA SigmodRecord. This causes a very large tree rooted at SigmodRecord to be returned to users. This is unacceptable even though the keywords are highlighted because it is very difficult for users to find their desired results. The precisions of some results returned by SLCA and MaxMatch are also low because of the false positive problem. But they are better than XRank, because the semantics of SLCA and MaxMatch prevent some false positive problems. However, for the query QS2, QS3 and QS5, the false positive problems cannot be avoided.

As shown in Figure 4.18, XKMis also achieves higher recall than XRank, SLCA and
4. XML KEYWORD QUERY PROCESSING

<table>
<thead>
<tr>
<th></th>
<th>F-Measure</th>
<th>XKMis</th>
<th>XRank</th>
<th>SLCA</th>
<th>MaxMatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSU</td>
<td>0.80</td>
<td>0.43</td>
<td>0.67</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>SigmodRecord</td>
<td>0.87</td>
<td>0.44</td>
<td>0.66</td>
<td>0.68</td>
<td></td>
</tr>
<tr>
<td>Mondial</td>
<td>0.79</td>
<td>0.51</td>
<td>0.48</td>
<td>0.48</td>
<td></td>
</tr>
<tr>
<td>DBLP</td>
<td>0.89</td>
<td>0.77</td>
<td>0.76</td>
<td>0.79</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Comparison on F-Measure: XKMis vs. XRank, SLCA and MaxMatch

MaxMatch especially when a query has only one keyword. For XRank, SLCA and MaxMatch, if a query includes only one keyword, they just return the nodes which contain that keyword. This kind of results is not informative and not desirable. The recall of SLCA and MaxMatch is worse than XRank because some valid results are ignored due to the false negative problem. Actually, in some queries, it is trivial for XRank and SLCA to achieve high recall because they return the entire XML data tree to users.

We calculated the average F-measure of the queries over each data set and they are listed in Table 4.6. It can be seen that XKMis achieves higher F-measure than both XRank, SLCA and MaxMatch.

4.7.3 DynamicInfer vs. XReal

In this section, we present the experimental results on the accuracy of inferring the search-for node type (SNT) of our approach DynamicInfer against XReal [1]. We selected several queries which have specific search intentions, but XReal produces inconsistency and abnormality problems. We present the SNTs after applying our dynamic reduction factor scheme and the new reduction factor in the results.

The keyword inverted list and statistics information are implemented in C++ and stored with Berkeley DB [91]. We used the data sets SigmodRecord, WSU, DBLP and Mondial for evaluation.

4.7.3.1 Results of Inferring Search-for Node

Inconsistency Problem 1 We used SigmodRecord data set for the experiments on preventing Inconsistency Problem 1 because this data set has a deep structure and lots of queries have this problem. As we stated earlier, inconsistency problems are most likely to happen on deep data trees. We replicated the data set two times to simulate the data size changes. The selected queries (Q1-Q15), the SNTs inferred by XReal, the
4.7 Experiments

<table>
<thead>
<tr>
<th>Query</th>
<th>SNT of XReal</th>
<th>SNT of Our approach</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>QI1 {article Karen}</td>
<td>article</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QI2 {article title SQL}</td>
<td>article</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QI3 {article database}</td>
<td>article</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QI4 {title data author}</td>
<td>article</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QI5 {title web initPage}</td>
<td>article</td>
<td>issue</td>
<td>article</td>
</tr>
</tbody>
</table>

Table 4.7: Results on Resolving Inconsistency Problem 1

SNTs inferred by DynamicInfer and the new reduction factor are listed in Table 4.7. For these five queries, XReal infers unpreferable SNTs when the data set is double sized, so we only listed the new reduction factor for the queries over the double-sized data set in the table. It can be seen that our approach can resolve Inconsistency Problem 1 by applying the dynamic reduction factor scheme.

Inconsistency Problem 2 We also used SigmodRecord data set to do the experiments on preventing Inconsistency Problem 2. We selected three pairs of similar queries and listed the new reduction factor for the second query in each pair in Table 4.8. The results show that the dynamic reduction factor scheme can successfully solve Inconsistency Problem 2.

Abnormality Problem Abnormality Problem is likely to happen on both deep and shallow tree structures, so we use SigmodRecord, WSU, DBLP and Mondial data set to do the experiments on preventing Abnormality Problem. We selected three queries for each data set. From the results shown in Table 4.9, our approach can also resolve abnormality problems. It should be noted that the keywords in query QA3 have relatively high frequencies compared with other queries, but XReal also infers an unpreferable SNT.

4.7.4 XInfer vs. XReal, XBridge, MaxMatch, DynamicInfer

In this section, we present the experimental results on the effectiveness of our approach against XReal [1], XBridge [67], MaxMatch [66] and DynamicInfer (see Section 4.3). We also present the experimental results on efficiency. The following metrics are used for evaluation:
4. XML KEYWORD QUERY PROCESSING

<table>
<thead>
<tr>
<th>Query</th>
<th>SNT of XReal</th>
<th>SNT of Our approach</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>QI6</td>
<td>{author Karen}</td>
<td>article</td>
<td>article</td>
</tr>
<tr>
<td>QI7</td>
<td>{author John}</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QI8</td>
<td>{title database}</td>
<td>article</td>
<td>article</td>
</tr>
<tr>
<td>QI9</td>
<td>{title query}</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QI10</td>
<td>{article title database}</td>
<td>article</td>
<td>article</td>
</tr>
<tr>
<td>QI11</td>
<td>{article title relational database}</td>
<td>article</td>
<td>article</td>
</tr>
</tbody>
</table>

Table 4.8: Results on Resolving Inconsistency Problem 2

<table>
<thead>
<tr>
<th>Query</th>
<th>SNT of XReal</th>
<th>SNT of Our approach</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA1</td>
<td>{Karen}</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QA2</td>
<td>{XML}</td>
<td>SigmodRecord</td>
<td>article</td>
</tr>
<tr>
<td>QA3</td>
<td>{database system}</td>
<td>issue</td>
<td>article</td>
</tr>
<tr>
<td>QA4</td>
<td>{crowe}</td>
<td>root</td>
<td>course</td>
</tr>
<tr>
<td>QA5</td>
<td>{models}</td>
<td>root</td>
<td>course</td>
</tr>
<tr>
<td>QA6</td>
<td>{labor}</td>
<td>root</td>
<td>course</td>
</tr>
<tr>
<td>QA7</td>
<td>{Han}</td>
<td>root</td>
<td>inproceedings</td>
</tr>
<tr>
<td>QA8</td>
<td>{XML}</td>
<td>root</td>
<td>inproceedings</td>
</tr>
<tr>
<td>QA9</td>
<td>{Software}</td>
<td>root</td>
<td>inproceedings</td>
</tr>
<tr>
<td>QA10</td>
<td>{Tirane}</td>
<td>root</td>
<td>city</td>
</tr>
<tr>
<td>QA11</td>
<td>{muslim}</td>
<td>root</td>
<td>country</td>
</tr>
<tr>
<td>QA12</td>
<td>{Turin}</td>
<td>root</td>
<td>city</td>
</tr>
</tbody>
</table>

Table 4.9: Results on Resolving Abnormality Problem

- index size
- search quality
  1. number of returned search intentions
  2. Precision, Recall and F-measure
  3. ranking effectiveness
  4. effect of individual factors on ranking effectiveness
- efficiency
  1. processing time
4.7 Experiments

2. scalability of efficiency on the number of word types

The queries we use for evaluation are listed in Table 4.10. The word type index, inverted list index and data statistics index are implemented in C++ and stored with Berkeley DB [91]. The data sets used for evaluation is DBLP, SigmodRecord, WSU and Mondial. Note that XBridge and DynamicInfer only provides information on how to suggest the promising result types, so we extend XBridge and DynamicInfer with the part of generating result subtrees. Similar to XReal, the subtrees that are rooted at the nodes of the suggested result types and contain all of the keywords are considered as the result subtrees.

4.7.4.1 Index Size

The sizes of three indexes (i.e., word type index, inverted list index and data statistics index) for each data set are shown in Table 4.11. We do scalability test and present the index size for DBLP fragments of different size in Figure 4.19. It can be seen that the size of inverted list index increases most quickly.

4.7.4.2 Search Quality

Number of returned search intentions

The number of returned search intentions directly affects the useability of the system. The user has to spend much time on locating his real search intentions if too many search intentions are returned. We list the minimum, maximum and average number of search intentions returned by our system for each data set in Table 4.12. It can be seen that the number of returned likely search intentions is quite small for each query.
## XML Keyword Query Processing

<table>
<thead>
<tr>
<th>Query</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>QD1</td>
<td>{Automated Software Engineering}</td>
</tr>
<tr>
<td>QD2</td>
<td>{Han data mining}</td>
</tr>
<tr>
<td>QD3</td>
<td>{WISE 2000}</td>
</tr>
<tr>
<td>QD4</td>
<td>{Jeffrey XML}</td>
</tr>
<tr>
<td>QD5</td>
<td>{author Jim Gray}</td>
</tr>
<tr>
<td>QD6</td>
<td>{Relational Database Theory}</td>
</tr>
<tr>
<td>QD7</td>
<td>{article spatial database}</td>
</tr>
<tr>
<td>QD8</td>
<td>{Wise database}</td>
</tr>
<tr>
<td>QD9</td>
<td>{Han VLDB 2000}</td>
</tr>
<tr>
<td>QD10</td>
<td>{twig pattern matching}</td>
</tr>
<tr>
<td>QW1</td>
<td>{CAC 101}</td>
</tr>
<tr>
<td>QW2</td>
<td>{title ECON}</td>
</tr>
<tr>
<td>QW3</td>
<td>{instructor MCELDOWNEY}</td>
</tr>
<tr>
<td>QW4</td>
<td>{FINITE MATH}</td>
</tr>
<tr>
<td>QW5</td>
<td>{prefix MATH}</td>
</tr>
<tr>
<td>QW6</td>
<td>{place TODD}</td>
</tr>
<tr>
<td>QW7</td>
<td>{COST ACCT enrolled}</td>
</tr>
<tr>
<td>QW8</td>
<td>{CELL BIOLOGY times}</td>
</tr>
<tr>
<td>QW9</td>
<td>{ECON days times place}</td>
</tr>
<tr>
<td>QW10</td>
<td>{prefix ACCTG instructor credit}</td>
</tr>
<tr>
<td>QM1</td>
<td>{Europe}</td>
</tr>
<tr>
<td>QM2</td>
<td>{Continent}</td>
</tr>
<tr>
<td>QM3</td>
<td>{Tirane, population}</td>
</tr>
<tr>
<td>QM4</td>
<td>{organization, name, members}</td>
</tr>
<tr>
<td>QM5</td>
<td>{city, name, longitude, latitude}</td>
</tr>
<tr>
<td>QM6</td>
<td>{country, name, population}</td>
</tr>
<tr>
<td>QM7</td>
<td>{muslim,country}</td>
</tr>
<tr>
<td>QM8</td>
<td>{Belarus, population}</td>
</tr>
<tr>
<td>QM9</td>
<td>{Turin, longitude, latitude}</td>
</tr>
<tr>
<td>QM10</td>
<td>{country, name, capital, population, datacode, government}</td>
</tr>
</tbody>
</table>

**Table 4.10: Queries**
4.7 Experiments

Figure 4.20: Precision: XInfer vs. XReal, XBridge, MaxMatch and DynamicInfer

Figure 4.21: Recall: XInfer vs. XReal, XBridge, MaxMatch and DynamicInfer
4. XML KEYWORD QUERY PROCESSING

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data size</th>
<th>Word Type</th>
<th>Dewey</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>SigmodRecord</td>
<td>0.5M</td>
<td>0.6M</td>
<td>2.6M</td>
<td>5.1M</td>
</tr>
<tr>
<td>DBLP</td>
<td>743M</td>
<td>540M</td>
<td>2897M</td>
<td>1201M</td>
</tr>
<tr>
<td>WSU</td>
<td>1.6M</td>
<td>0.4M</td>
<td>4.5M</td>
<td>1.4M</td>
</tr>
<tr>
<td>Mondial</td>
<td>4.6M</td>
<td>2.4M</td>
<td>12.3M</td>
<td>4.1M</td>
</tr>
</tbody>
</table>

Table 4.11: Index size

<table>
<thead>
<tr>
<th>Data set</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>1</td>
<td>8</td>
<td>4.2</td>
</tr>
<tr>
<td>SigmodRecord</td>
<td>1</td>
<td>4</td>
<td>1.8</td>
</tr>
<tr>
<td>WSU</td>
<td>1</td>
<td>3</td>
<td>1.7</td>
</tr>
<tr>
<td>Mondial</td>
<td>1</td>
<td>7</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Table 4.12: Number of returned search intentions by XInfer

**Precision, Recall and F Measure**

We conducted a user survey on the search intentions of the queries in Table 4.10. Nineteen people participated in the survey. We used the search intentions selected by at least ten out of nineteen people to determine the relevant matches. We evaluate the effectiveness of XReal, XBridge, MaxMatch, DynamicInfer and XInfer based on precision, recall and F-measure (See Formula 4.16, 4.17 and 4.18).

From Figure 4.20(a), it can be seen that XReal and XBridge have a very low precision on the queries QD1, QD6 and QD8. This is mainly because XReal and XBridge infer undesired Search-for Node Types (SNTs). For these queries, XReal and XBridge infer the same SNTs, even though they use different strategies. Suppose the user submits QD1 to retrieve the articles from the journal of Automated Software Engineering. XReal and XBridge just return the inproceedings that are related to the automated software engineering, but both MaxMatch and XInfer return the articles from the journal of Automated Software Engineering besides the inproceedings about automated software engineering. Suppose the user submits the query QD8 to retrieve the publications written by Wise. XReal and XBridge return the inproceedings of WISE conference. MaxMatch and XInfer return the inproceedings of WISE conference as well as Wise’s publications. Suppose the user wants to search the book called Relational Database Theory and submits the query QD6. XReal and XBridge does not return this book. MaxMatch and XInfer return this book but have low precisions because they return much irrelevant information at the same time (e.g., the inproceedings about
4.7 Experiments

relational database theory, etc). On Query QD2-QD4, XReal and XBridge achieve a little higher precisions than MaxMatch and XInfer because XReal and XBridge just infer one search-for node type which reduces the irrelevant information in the results. Actually this is not a serious problem for XInfer because XInfer rank the desired search intention as the top-1 search intention, so it is very easy for the user to find their desired results. The user has to spend much time on browsing desired results. Suppose the user wants to retrieve the publications written by Jim Gray and submits query QD5. XReal, XBridge and XInfer return the desired results, but MaxMatch just returns the author nodes, which means the returned information is too limited. Therefore, MaxMatch has a very low precision on this query. Figure 4.21(a) presents the recalls of XReal, XBridge, MaxMatch and XInfer on the query QD1-QD10. XReal and XBridge have very low recalls on the query QD1, QD6 and QD8 because it does not return the relevant results as we explained above. MaxMatch has a very low recall on the query QD5 because it just returns the subtrees rooted at author nodes.

As shown in Figure 4.20(b), XInfer achieves higher precision than XReal, XBridge and MaxMatch for the SigmodRecord dataset. For the queries QS1 and QS4, XReal shows low precision mainly because it infers issue as the search-for node type for these two queries, which results in many irrelevant articles being returned to the user. For example, according to the survey, the user intends to retrieve articles about database design with the query QS1. However, lots of articles that are not related to database design are also returned to the user in XReal. XBridge and XInfer returns the articles about database design for QS1 and the articles written by Karen Ward for QS4, which are desired by the user. XInfer, XReal and MaxMatch achieve good precisions on the queries QS2 and QS3. MaxMatch gets very low precision on the queries QS4 and QS5. Most participants think the user wants the articles written by Karen Ward with the queries QS4 or QS5, but MaxMatch only returns author nodes, which do not provide much information desired by the user. In order to retrieve the articles about database from the issues of volume 11, the user submits the query QS6 or QS7. XReal and XBridge return lots of irrelevant articles (including the articles that are not about database, the database articles whose initPage is 11, etc) to the user. XInfer infers five search intentions on QS6 and one search intention on QS7. Compared with QS6, QS7 adds a new keyword “issue” which is used to specify the meaning of “11”. XInfer
4. XML KEYWORD QUERY PROCESSING

Table 4.13: Comparison on F-Measure: XInfer vs. XReal, XBridge, MaxMatch and DynamicInfer

<table>
<thead>
<tr>
<th></th>
<th>XReal</th>
<th>XBridge</th>
<th>MaxMatch</th>
<th>DynamicInfer</th>
<th>XInfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>0.69</td>
<td>0.69</td>
<td>0.79</td>
<td>0.69</td>
<td>0.90</td>
</tr>
<tr>
<td>SigmodRecord</td>
<td>0.51</td>
<td>0.65</td>
<td>0.48</td>
<td>0.51</td>
<td>0.88</td>
</tr>
<tr>
<td>WSU</td>
<td>0.96</td>
<td>0.96</td>
<td>0.91</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td>Mondial</td>
<td>0.68</td>
<td>0.71</td>
<td>0.78</td>
<td>0.68</td>
<td>0.89</td>
</tr>
</tbody>
</table>

notices this difference, and correctly infers the user’s search intention. For the recall, as shown in Figure 4.21(b), all of these three approaches present good recalls.

For data set WSU, Figure 4.20(c) shows that all of these approaches generally achieve good precision. This is mainly because WSU has a simple and shallow structure compared with the data set SigmodRecord. For the query QW9, they present a relatively low precision. The user intends to retrieve the days, times and place of the courses whose titles contain “ECON”, but the systems return the courses whose prefix contain “ECON” as well and give them the highest ranks because most words “ECON” appear in the prefix nodes. MaxMatch has a low precision on query QW2 because it also returns the courses whose prefix contain “ECON” even though the user adds a describing word “title”. As shown in Figure 4.21(c), all of these three approaches present good recalls.

We calculated the average F-measure of the queries over each data set and they are listed in Table 4.13. It can be seen that XInfer achieves higher F-measure than XReal, XBridge, MaxMatch and DynamicInfer over all three datasets.

Discussion XInfer exploits user’s typing habit and stresses the importance of the relationship between adjacent keywords. When the keywords in a query have many word types and the data tree is deep, the advantages of XInfer will be much clearer (e.g., QS6, QS7, QS8). In addition, if the number of occurrences of the desired word types is not overwhelming in the dataset, XInfer will get better results. In this situation, it is difficult to infer the desired search intention very accurately, so it is much safer to return a list of likely search intentions with a proper ranking. XInfer chooses this way and tries to exclude the unreasonable search intentions. For other systems, the desired results may be missing (e.g., QD1, QD7, QD8).

Ranking effectiveness XInfer ranks the search intentions before they are returned
4.7 Experiments

Table 4.14: Ranking effectiveness of XInfer

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Top-1 number/Total Number</th>
<th>R-Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>25/30</td>
<td>0.92</td>
</tr>
<tr>
<td>SigmodRecord</td>
<td>27/30</td>
<td>0.87</td>
</tr>
<tr>
<td>WSU</td>
<td>29/30</td>
<td>0.97</td>
</tr>
<tr>
<td>Mondial</td>
<td>22/30</td>
<td>0.84</td>
</tr>
</tbody>
</table>

Table 4.15: Influencing factors

<table>
<thead>
<tr>
<th>Factor</th>
<th>Dataset</th>
<th>Top-1 num/Total num</th>
<th>R-Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dis</td>
<td>DBLP</td>
<td>20/30</td>
<td>0.79</td>
</tr>
<tr>
<td>Dis</td>
<td>SigmodRecord</td>
<td>19/30</td>
<td>0.72</td>
</tr>
<tr>
<td>Dis</td>
<td>WSU</td>
<td>26/30</td>
<td>0.91</td>
</tr>
<tr>
<td>Dis</td>
<td>Mondial</td>
<td>15/30</td>
<td>0.67</td>
</tr>
<tr>
<td>Sta</td>
<td>DBLP</td>
<td>16/30</td>
<td>0.78</td>
</tr>
<tr>
<td>Sta</td>
<td>SigmodRecord</td>
<td>14/30</td>
<td>0.65</td>
</tr>
<tr>
<td>Sta</td>
<td>WSU</td>
<td>25/30</td>
<td>0.91</td>
</tr>
<tr>
<td>Sta</td>
<td>Mondial</td>
<td>19/30</td>
<td>0.68</td>
</tr>
</tbody>
</table>

To the user. In order to evaluate the effectiveness of its ranking strategy, we use the following two measures which are widely applied in IR field: (a) Number of top-1 answers that are relevant; (b) Reciprocal rank (R-Rank). For a given query, the reciprocal rank is 1 divided by the rank at which the first correct answer is returned, or 0 if no correct answer is returned.

Besides the queries in Table 4.10, we produced twenty more queries for each dataset. The relevance judgments are done by nineteen people. A search intention is considered relevant if at least ten out of nineteen people agree. The evaluation results can be found in Table 4.14. As shown in this table, XInfer achieves good ranking effectiveness.

Effect of individual factors on ranking effectiveness The ranking of likely search intentions is mainly influenced by the factors of distance and statistics (See Section 4.4.2.2). In order to see the individual effects of each factor, we report the ranking effectiveness by ignoring one of them in Table 4.15. It can be seen that the system has worse ranking effectiveness when utilizing individual factor. The ranking effectiveness while using the factor of distance is better than that while using the factor of statistics. Therefore, the intuition on distance takes the major job in intention ranking.
4. XML KEYWORD QUERY PROCESSING

4.7.4.3 Efficiency

Processing time

The processing time of a query can be categorized into two parts: inferring likely search intentions and generating result subtrees. In Figure 4.22, we present the categorized processing time of the queries DQ1-DQ10 over the DBLP data set, and list the total number of word types processed for each query above each bar. Each query is run ten times after memory warm-up. From Figure 4.22, we can see that the time spent on inferring likely search intentions is very little and is correlated with the total number of word types processed.

Scalability of efficiency on the total number of word types
As we discussed in Section 4.4.4, XInfer does not immediately return all result subtrees after the user clicks the search button. It first returns one result subtree of each search intention to the user for verification, which is different from existing systems. Therefore, it is meaningless to compare the performance of XInfer with XReal and MaxMatch. Instead, we studied the performance of inferring search intentions by varying the total number of word types that are processed by the system. We vary the number of word types from 10 to 60 with a difference of 10. For each step, we manually selected 10 queries, which have the corresponding number of word types, over DBLP dataset. Each query is run five times, and the average processing time is calculated. Then we calculated the average processing time of 10 queries for each step. The results are shown in Figure 4.23. As shown in this figure, the processing time spent on inferring search intentions increases moderately when the number of word types increases.

---

1 We choose the DBLP dataset because this dataset has much larger size than the other two datasets, and most words own many word types.
4.7 Experiments

Figure 4.23: Scalability of efficiency of XInfer

![Figure 4.23: Scalability of efficiency of XInfer](image)

(a) Precision  (b) Recall  (c) F-Measure

Figure 4.24: Precision, Recall and F-Measure on Mondial Data Set

4.7.5 XQSuggest vs. MaxMatch, XInfer

In this section, we evaluate the search quality of XQSuggest. We used the data sets WSU, SigmodRecord, Mondial and DBLP. The queries are listed in Table 4.10.

4.7.5.1 Search Quality

We compared the search quality of XQSuggest with MaxMatch [66] and XInfer (see Section 4.4). We presented in Section 4.7.4 that XInfer has better effectiveness than previous systems. We evaluate the effectiveness of XQSuggest, MaxMatch and XInfer based on precision, recall and F-Measure (See Formula 4.16, 4.17 and 4.18).

The comparisons of precision, recall and F-Measure over Mondial are illustrated in Figure 4.24. As shown in the figure, XQSuggest achieves higher precision, recall and F-Measure than MaxMatch and XInfer. This is mainly because the submitted keywords exist in different types of nodes and the user can eliminate the irrelevant meanings with our system. Figure 4.24 (c) presents the F-Measure of the queries with $\alpha = 0.5, 1$ and 2. It is shown that XQSuggest outperforms MaxMatch and XInfer. The comparisons of Precision, Recall and F-Measure over SigmodRecord, WSU and DBLP are illustrated in Figure 4.25, 4.26 and 4.27. On the SigmodRecord, WSU and DBLP data set, the advantages of XQSuggest is not that obvious because most keywords in the data set have unique meanings.
4. XML KEYWORD QUERY PROCESSING

![Precision](image1)
(a) Precision

![Recall](image2)
(b) Recall

![F-Measure](image3)
(c) F-Measure

Figure 4.25: Precision, Recall and F-Measure on SigmodRecord Data Set

![Precision](image4)
(a) Precision

![Recall](image5)
(b) Recall

![F-Measure](image6)
(c) F-Measure

Figure 4.26: Precision, Recall and F-Measure on WSU Data Set

4.8 Summary

In this section, we proposed four approaches to improve the effectiveness of XML keyword search. These approaches achieve the goal in two directions:

1. Improving the meaningfulness of results. Meaningfulness means a result should contain enough useful information but should not be too overwhelming. On the other hand, irrelevant results should not be linked together. To achieve this, we proposed XKMis. In XKMis, the XML data is organized, stored and indexed in Minimum Information Segments (MISs). The experiments showed that XKMis outperforms XRank [58], SLCA [59] and MaxMatch [66] on search quality. It also achieves better performance than XRank and SLCA because it uses region-based coding scheme instead of DeweyIDs.

2. Inferring the user’s search intention. A keyword may have multiple meanings in the XML data, so it is difficult to know the user’s search intention. The Search-for Node Type (SNT) is an important part of the user’s search intention. We proposed DynamicInfer to improve the accuracy of inferring SNT by resolving the problems (i.e., the inconsistency and abnormality problem) of XReal [1]. The experimental results verified this. Different from DynamicInfer, XInfer tries to infer the meaning of each keyword by analyzing the user’s typing habit and data...
statistics. It was shown that XInfer achieves better search quality than XReal, XBridge [67], MaxMatch and DynamicInfer. Although DynamicInfer and XInfer improves the precision of inferring the user’s search intention, it is impossible for a search engine to correctly infer the user’s search intention all the time. Therefore, we can solve the problem from another angle: let the user specify the meaning of each keyword. XQSuggest provides such a utility. The system can instantly suggest a list of semantic strings for the user to select when the user is typing each keyword. This will significantly improve the effectiveness if the user knows what he wants. The experiments showed that XQSuggest achieves better effectiveness than XInfer and MaxMatch.
4. XML KEYWORD QUERY PROCESSING
5

Conclusion and Future Work

XML has become a popular standard for storing and exchanging semi-structured data. This thesis addresses the problems of XML twig pattern matching and XML keyword search. XML twig pattern matching is a complex selection on the structure of an XML document and lies in the centre of most XML query languages. XML keyword search is a user friendly tool for unprofessional users to retrieve information from XML documents.

5.1 Thesis Contributions

In this thesis, we first proposed several algorithms for improving the efficiency of XML twig pattern matching. Then we proposed several approaches for improving the effectiveness of XML keyword search. For both XML twig pattern matching and XML keyword search, we first built positional information for each node in an XML data tree, then we find and output final results to the user.

The contributions of this thesis can be summarised as follows:

1. We proposed two one-phase holistic twig pattern matching algorithms TwigMix and TwigFast based on region-based labelling schemes. TwigMix introduces efficient filtering of useless elements into TwigList. TwigFast improves the efficiency further by avoiding using an intermediate stack. Our extensive experiments showed that both TwigMix and TwigFast outperform TwigList and HolisticTwigStack up to several times faster.

2. Existing twig pattern matching algorithms suffer from redundant computation. We proposed two approaches, namely re-test checking and forward-to-end, to
resolve the redundant computation. These two approaches introduce minor overheads and can be easily applied to existing twig pattern matching algorithms. According to our experiments, these two approaches can significantly accelerate existing algorithms.

3. To improve the efficiency and effectiveness of XML keyword search, we proposed an XML keyword search system XKMis, which is not LCA-based and divides an XML document into meaningful information segments (MIS). This approach ensures that the result is informative but not that overwhelming. We conducted extensive experiments which verified better effectiveness and efficiency of our approach.

4. For XML keyword search, the precision of inferring the type of return nodes is important. We proposed a dynamic reduction factor scheme and a corresponding algorithm DynamicInfer to infer the search-for node type, which resolves the two problems in the inference approach proposed by XReal. Our experiments showed that the precision was significantly improved after applying our inference method.

5. We proposed an XML keyword search system XInfer which exploits users’ typing habit while constructing keyword queries and data statistics in an XML document. XInfer uses Pair-wise Comparison Strategy (PCS) to infer a set of likely search intentions of a keyword query. The user can easily identify the result subtrees of his real search intention. We conducted extensive experiments which verified better effectiveness of our approach.

6. We introduced query suggestion to XML keyword search and proposed an interactive XML keyword search system XQSuggest. Unprofessional users can help the system resolve the ambiguity of keywords by selecting the semantic string for each keyword. Our experiments and survey demonstrate the usability of our system and better effectiveness.

Some of the material in this thesis appear in our papers [16], [17], [18], [19], [20], [21].
5.2 Future Work

For XML twig pattern matching and XML keyword search, we list several issues that need to be further investigated.

1. **XML twig pattern with /-edges** Existing XML twig pattern matching algorithms may have much redundant computation when /-edges appear in a query tree. Although the most recent algorithms \texttt{TJStrictPre} and \texttt{TJStrictPost} \cite{16} are proved to be optimal when /-edges exist, they still have much redundant computation and overheads in practice.

2. **Exploring ID-References for XML keyword search** An XML document may introduce ID-Reference to reduce redundancy. This kind of documents can be modelled as digraphs. The key concept in digraph model is called reduced subtrees, which searches for minimal connected subtrees in graphs. However, the problem of finding all reduced subtrees and enumerating results by increasing sizes of reduced subtrees is NP-hard. Thus, most previous algorithms on XML digraph model are intrinsically expensive and heuristics-based. Therefore, it is valuable to find better ways to efficiently and effectively processing XML keyword search in graphs.

3. **Query expansion for XML keyword search** Query expansion is a useful tool for helping users refine their queries. To get better results, the initial query can be refined using synonyms and related words, but how to choose appropriate synonyms and related words is not easy. This technique has been studied and applied in web search but has not been explored much for XML keyword search. For XML documents, the structural relationships between different words should be considered as an important factor.

4. **Relevance feedback in XML keyword search** Relevance feedback is a query refinement technique in web search. It is also worth exploration in XML keyword search. The basic idea is system can reformulate the query (e.g., re-weight the query terms) according to the user’s feedback. This technique is actually an application of machine learning. The user’s feedback can be regarded as the training data. Another good application of relevance feedback is when the user
5. CONCLUSION AND FUTURE WORK

When a user clicks a result that he is interested in, the system can instantly provide a list of similar and relevant results. These results are more likely desired by the user.
References


REFERENCES


References for Effective Keyword Search in XML


REFERENCES

[90] http://xmlsoft.org/ 45 104


