Enhancing the Usability of XML Keyword Search

ZENG YONG

(B.Eng, South China University of Technology, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

ZENGYONG July 8, 2014

ZENG YONG

July 8, 2014

ACKNOWLEDGEMENT

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Ling Tok Wang, who has provided invaluable guidance in every stage of my research work. I am very grateful for the countless hours he has spent supervising me and discussing with me. It has been five years since I became a student of Prof. Ling. During the five years, I have learned a lot from Prof. Ling, from how to identify research problems to how to tackle a research problem. His rigorous attitude on research inspires me to think critically in my research. His technical advice is essential to the completion of this thesis, while his kindness and wisdom will keep inspiring me to move forward in the rest of my life.

Moreover, I also feel very grateful for the guidance given by my senior, Dr. Bao Zhifeng, who has collaborated with me for every piece of my research work. He has provided me with continues help through out my whole Ph.D study. His encouragement and calm manner had always helped me regain my confidence in my research.

Besides, I would also like to thank Prof. Stephane Bressan and Prof. Tan Kian-Lee for serving on my thesis committee and providing many useful comments on the thesis.

Last but not least, I wish to express my appreciation to my family, especially my wife DU YINGJUN, for their support to me, even at the most difficulty time in my Ph.D study.

CONTENTS

Acknowledgement **i**

Summary **viii**

1 Introduction **1**

1.1 Background 1

1.1.1 XML and Data Model 1

1.1.2 Querying XML 4

1.2 Research Problem: Enhancing the Usability of XML Keyword Search 6

1.3 Contributions of This Dissertation 9

1.3.1 MisMatch Problem in Keyword Search over XML without ID References 10

1.3.2 MisMatch Problem in Keyword Search over XML with ID References 11

1.3.3 Query Result Presentation 12

1.4 Thesis Outline 13

2 Related Work **14**

2.1 Labeling for XML 14

2.2 Structured Query on XML 17

2.3 Keyword Search on XML 18

2.3.1 Tree Model 18

2.3.2 Graph Model 24

2.4 Query Refinement 25

2.4.1 Query Cleaning 25

2.4.2 Query Relaxation 27

2.4.3 Query Substitution 27

2.4.4 MisMatch Problem in Structured and Unstructured Data . . 29

2.5 Query Results Visualization 31

3 MisMatch Problem in Keyword Search Over XML without ID References **35**

3.1 Introduction 35

3.2 Preliminaries 41

3.2.1 Semantics and Data Model 41

3.2.2 General Query Result Format 43

3.3 Detecting the Mismatch Problem 44

3.3.1 Detecting The MisMatch Problem based on Target Node Type 51

3.4 Finding Explanations and Suggested Queries 52

3.4.1 Distinguishability 53

3.4.2 Two-phase Solution 55

3.4.3 Ranking the Suggested Queries 62

3.4.4 Summary of Features of Our Approach 63

3.5 Efficient Approximate Results Detection 63

3.5.1 Node Labeling 64
3.5.2 Logical Operation 66
3.6 Algorithms 66
3.6.1 Data Processing and Index Construction 66
3.6.2 Solving the MisMatch problem 68
3.7 Experiments 71
3.7.1 Experimental Settings 72
3.7.2 Frequency of the MisMatch Problem 73
3.7.3 Sensitivity of the MisMatch Detector 73
3.7.4 Quality of the Suggested Queries 74
3.7.5 Comparison to XRank 78
3.7.6 Sample Query Processing Time 79
3.7.7 Scalability Test 81
3.8 XClear Demo System 84
3.9 Conclusion 85

4 MisMatch Problem in Keyword Search Over XML with ID References 87

4.1 Introduction 87
4.2 Preliminaries 90
4.2.1 Semantics and Data Model 90
4.2.2 Reference Types 91
4.3 Transforming Query Processing over XML IDREF Digraph to XML Tree 92
4.3.1 Naive Approach: Real Replication 92
4.3.2 Our Approach: Virtual Replication 94
4.3.3 Query Evaluation 98

4.4 Sequential References and Cyclic References 100

4.4.1 Sequential References 101

4.4.2 Cyclic References 101

4.4.3 Reachability Table Space Complexity 102

4.5 Further Extension and Optimization for Query Evaluation 103

4.5.1 Removing unnecessary checking of the reachability table . . 103

4.5.2 Adding Distance and Path to Reachability Table 104

4.6 Solving the MisMatch Problem in XML IDREF Digraph 105

4.6.1 Target Node Type for Detecting MisMatch Problem 107

4.6.2 Distinguishability for Measuring Keywords' Importance . . . 109

4.6.3 exLabel for Efficient Approximate Results Detection 112

4.7 Algorithms 114

4.8 Experiments 117

4.8.1 Keyword Search on XML IDREF Digraph 117

4.8.2 MisMatch Solution on XML IDREF Digraph 121

4.9 Conclusion 127

5 Query Result Presentation of XML Keyword Search 129

5.1 Introduction 129

5.2 Building XMAP 135

5.2.1 Generating Layers for XMAP 135

5.2.2 Index of XMAP 138

5.3 XMAP Working with a Search Engine 141

5.3.1 Static Approach: Highlight all Query Results in XMAP . . . 141

5.3.2 Dynamic Approach: Generate a New Display 143

5.4 Algorithms 146

5.4.1 Index Construction 146

5.4.2 Retrieving data from the index 148

5.5 Experiments 149

5.6 XMAP Demo System 151

5.7 Conclusion 153

6 Conclusion and Future Work 154

6.1 Conclusion 154

6.2 Future Work 160

Bibliography 165

Appendix A: XClear Demo System 177

Appendix B: XMAP Demo System 182

Appendix C: Integrating XClear and XMAP 187

SUMMARY

XML has become a de facto standard of information representation and exchange over the Internet. It has been used extensively in many applications. Such semi-structured data is normally queried by rigorous structured query languages, e.g., XPath, XQuery, etc. In recent years, keyword search on XML has become more and more popular due to its easy-to-use query interface. It provides an opportunity to explore the semi-structured data without knowing the data schema or learning the sophisticated structured query languages. It is becoming an equally important counterpart of structured query and an important way for novice to explore XML database.

XML keyword search has been abundantly studied in the last ten years. The research efforts mainly focus on defining what should be returned as results (matching semantics) and designing efficient algorithms for a certain matching semantics.

However, in XML keyword search, how to reduce the gap between users' search intention and the query results remains a challenge. Even for the mature web search, users have to reformulate and resubmit their queries 40% to 52% of the time in order to get what they want [86]. Therefore, enhancing the usability by handling the mismatch between users' search intention and the query results is an important issue, no matter for web search, XML keyword search, or any other kind of search. In this dissertation, we will study how to enhance the usability of XML keyword search by addressing the following challenges.

First, we study the mismatch results in XML keyword search without considering ID references. In this case, the XML data can be modeled as a tree. We develop a low-cost post-processing algorithm on the results of query evaluation to detect the mismatch and generate helpful suggestions to users. The solution is based on two novel concepts that we introduce: *Target Node Type* and *Distinguishability*. *Target Node Type* represents the type of node a query result intends to match, and *distinguishability* is used to measure the importance of the query keywords in a query. Our solution can work with any LCA-based matching semantics and is orthogonal to the choice of result retrieval method adopted. We have also built an interactive XML keyword search engine, called XClear [104], with our mismatch solution incorporated. The demo system is available at [104]. The details of the demo system will be presented in Appendix A.

Second, we try to extend our mismatch solution to XML data with ID references considered. Then the XML data is usually modeled as a digraph, where keyword query results are usually computed by graph traversal. We call such a digraph as XML IDREF digraph in this dissertation. We observe that an XML IDREF digraph is mainly a tree structure with a portion of reference edges. It motivates us to propose a novel method to transform an XML IDREF digraph with ID references to a tree model, such that we can exploit abundant efficient XML tree search methods. Subsequently our mismatch solution designed for an XML tree can still apply.

Third, after the results are retrieved from the search engine, they need to be

presented to users. To further bridge the mismatch gap between users' search intention and the query results, we improve the result presentation method for XML keyword search, which plays an important role in users' digesting and exploring of the query results. The traditional way of returning a list of subtrees as query results is insufficient to meet the information needs of users. We find that such a presentation is imprecise and could be misleading. Users could misunderstand the query results. Therefore we propose an interactive and novel result presentation model, call XMAP, to visualize and work as a complementary component of the XML keyword search engine, in order to enhance the usability of XML keyword search. It allows users to view the inter-relationship among the query results and also further explore the query results according to their information needs. A demo system of XMAP has also been built [101], whose details will be presented in Appendix B.

Besides, we also discussed about how to integrate the two demo systems mentioned above, XClear and XMAP, in Appendix C.

LIST OF FIGURES

1.1 An Example XML Document about Store Inventory (inventory.xml) 2
1.2 XML Tree for inventory.xml in Figure 1.1 3
1.3 XML IDREF Digraph for inventory.xml in Figure 1.1 4

2.1 A sample XML Tree With Dewey Label (bookstore.xml) 16
2.2 Relationship among Main Keyword Search Techniques 22
2.3 Timeline for Main Keyword Search Techniques 24
2.4 Comparison of Query Refinement Approaches 28

3.1 Sample XML Document about an Online Shopping Mall 37
3.2 An XML Tree with Nodes Labeled by exLabels 64
3.3 Schema Tree Flattening and Virtual Bitmap Construction 64
3.4 Schema Graph of IMDB Dataset 72
3.5 Average Quality Measure of Suggested Queries 76
3.6 Precision for Top-5 results of XClear vs. XRANK 78
3.7 Processing Time for some Sample Queries 80
3.8 Impact of Data Size. 81

3.9 Impact of Distinguishability Threshold τ 82
3.10 Scalability Test of Random Queries 83
3.11 Suggested Queries & Sample Query Result 84

4.1 An Example XML Document (with Dewey Labels) 89
4.2 Naive Method: Real Replication 94
4.3 Advanced Method: Virtual Replication (Two Parts) 95
4.4 Constructing Reachability Table for Sequential References 100
4.5 Constructing Reachability Table for Cyclic References 102
4.6 Sample XML Document with ID References 105
4.7 Schema Graph of Figure 4.6 111
4.8 Query Execution Time (45MB data Size) 119
4.9 Query Execution Time (200MB Data Size) 119
4.10 Schema Graph of ACMDL Dataset (some parts are omitted because full schema graph is too big to display) 123
4.11 Average Quality Measure of Suggested Queries 123
4.12 Processing Time for some Sample Queries 125
4.13 Impact of Data Size. 126
4.14 Scalability Test of Random Queries 127

5.1 Sample XML Document about the Chain-stores in a Company . . . 130
5.2 Working of A Typical Digital Map System 133
5.3 Generating layer$_2$ and layer$_3$ for Figure 5.1 134
5.4 Index of the data shown in Figure 5.1 137
5.5 Query results highlighted of the query “Allen female” at layer$_3$ . . . 142
5.6 Context Display for the Query Results of Query “pencil black” . . . 145
5.7 Average Retrieval Time for Each Layer 150

5.8 Screenshot of XMAP for the query in Example 5.1 151
5.9 Screenshot of XMAP for the query in Example 5.1 (zoomed in) . . 152

1 Architecture of XClear System 178
2 Suggested Queries & Sample Query Result 179
3 Reasoning of "why" 180
4 Architecture of XMAP 183
5 Screenshot of XMAP for a query "pencil black" addressing Motivation 1 184
6 Screenshot of XMAP for a query "pencil black" (zoomed in) 184
7 Screenshot of XMAP for a query "Allen female" addressing Motivation 2 185
8 Architecture of XML ClearMap 188
9 XML ClearMap for Query without MisMatch Problem 189
10 Result Exploration Display of XML ClearMap 191
11 XML ClearMap for Query with MisMatch Problem 192

CHAPTER 1

INTRODUCTION

1.1 Background

1.1.1 XML and Data Model

XML (eXtensible Markup Language) has become a de facto standard of information representation and ex-change over the Internet. As compared to HTML which focuses on displaying and formatting data, XML does not have predefined elements and attributes. It provides a flexible way for users to define their own elements and attributes and define the structure of the data. With its powerful expressiveness and the recommendation of the World Wide Web Consortium (W3C), XML has been extensively used by many applications over the internet. Actually XML is a simplified subset of Standard Generalized Markup Language (SGML), whose specification is considered too complex to use and implement. XML's specification keeps the essence of SGML's power and extensibility with a much simpler specification.

Figure 1.1 shows an XML document describing the inventory information of a store, including items, quantity, suppliers, etc. Generally, the XML document is organized in a hierarchical structure, where the data is bounded in a pair of starting tag and ending tag. For example, the tag "store_inventory" at line 1 is the root node of the whole XML document. It forms a pair with the tag at line 29. Line 2 to line 28 are the content within the root node. "stock" (line 2) and "supplier" (line 25) are two children of the root node "store_inventory".

```
01 <store_inventory>
02     <stock>
03         <category>
04             <name>stationery</name>
05             <item id="i001" supplier="sp21">
06                 <name>pencil</name>
07                 <color>black</color>
08                 <quantity>300</quantity>
09             </item>
10             <item id="i002"  supplier="sp21">
11                 <name>pencil</name>
12                 <color>yellow</color>
13                 <quantity>50</quantity>
14             </item>
15         </category>
16         <category>
17             <name>make-up</name>
18             <item id="i201"  supplier="sp21">
19                 <name>pencil</name>
20                 <color>black</color>
21                 <quantity>300</quantity>
22             </item>
23         </category>
24     </stock>
25     <supplier id="sp21">
26         <name>Alps</name>
27         <phone>380945</phone>
28     </supplier>
29 </store_inventory>
```

Figure 1.1: An Example XML Document about Store Inventory (inventory.xml)

Besides, each item or supplier has an ID attribute. And the relationship between the item and the supplier is expressed by the ID references among the data. For example, at line 5 of the document, the item has an ID as "i001". Its supplier is referencing to the supplier with ID being "sp21", which is at line 25.

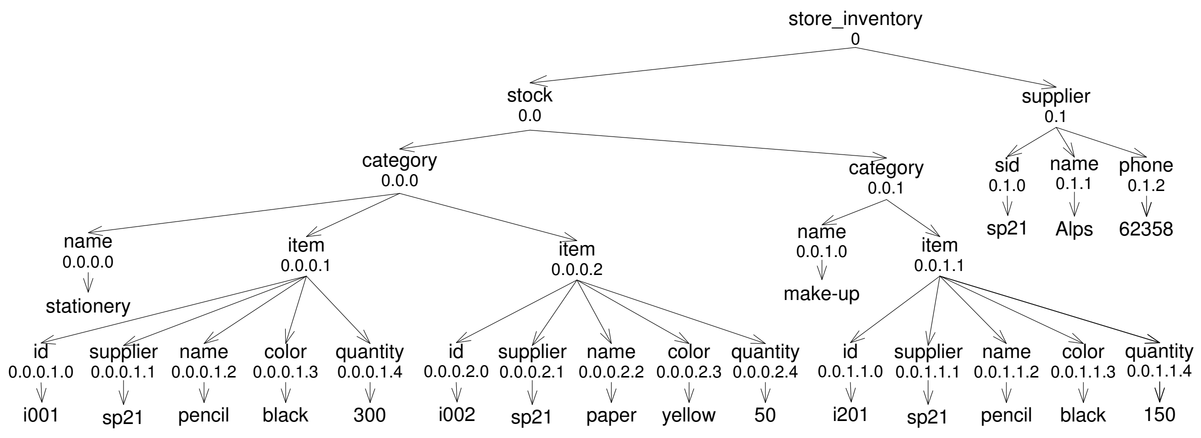


Figure 1.2: XML Tree for inventory.xml in Figure 1.1

If the ID reference relationship is not considered in the XML document, an XML document can be modeled as a tree. Each element or attribute in the XML data corresponds to one node in the tree; each element-subelement or element-attribute relationship in the XML document corresponds to an edge in the tree. For example, Figure 1.2 shows the tree model for the XML document in Figure 1.1. To uniquely identify each node in the tree, we assign each node a unique label, where we adopt dewey label [93]. The formal explanation of XML labeling scheme has to wait until the related work in Section 2.

As a comparison, if the ID reference relationship is considered, then an XML document is no longer a tree. Because for each reference node r in the XML document, the reference forms an edge from r to the element node which it references to. Therefore, an XML document considering ID references is usually modeled as a digraph, which we called as XML IDREF digraph in this dissertation. For example, Figure 1.3 shows the XML IDREF digraph for the XML document in Figure 1.1.

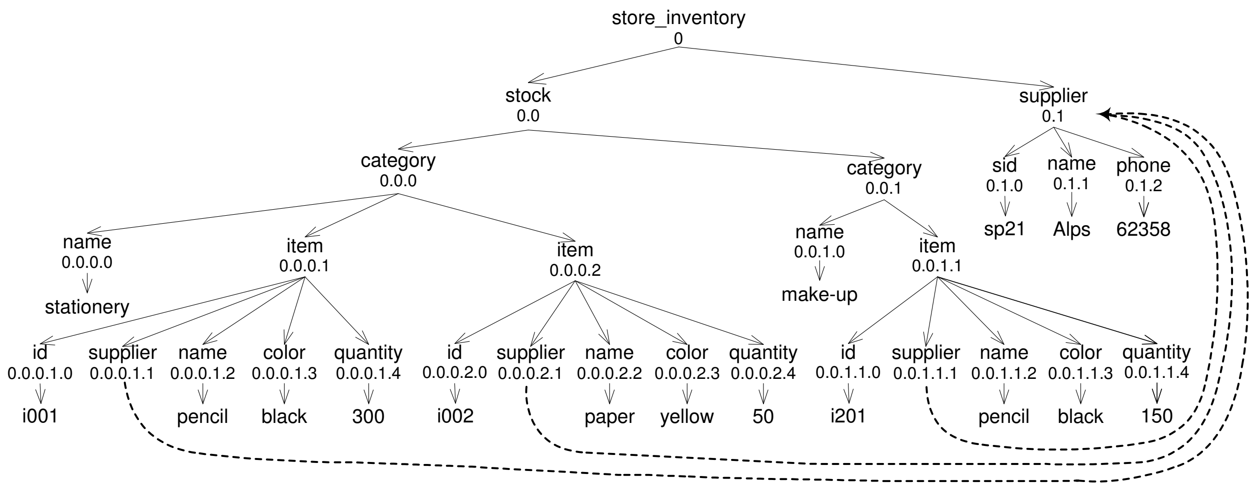


Figure 1.3: XML IDREF Digraph for inventory.xml in Figure 1.1

Comparing Figure 1.3 to Figure 1.2, we can see that the only difference is: the value under each reference node becomes an edge starting from the reference node to the corresponding element node.

1.1.2 Querying XML

There are mainly two categories of queries on XML data, i.e., structured queries and keyword queries. For structured queries, it is similar to SQL queries in relational database. Before a user can retrieve information from the XML data, the user is required to learn the complex query language and to be familiar with the schema of the XML data. XPath [11] and XQuery [13] are two structured query languages designed for XML data. The core pattern of XPath and XQuery queries is the called *twig pattern*.

Example 1.1. *For the XML data tree in Figure 1.2, if we want to find the phone number of supplier* Alps*, we can issue the following XQuery query:*

FOR $p IN

document("inventory.xml")//supplier[name="Alps"]/phone

RETURN $p

The core part of the query is to specify a supplier node in the XML document which has a descendent with name *being "Alps".*

As we can see from the Example 1.1, issuing a correct query according to the rigorous structured query language may not be an easy task for novice. In contrast, keyword search, which is the major form of retrieval method in information retrieval systems (like Google, Bing, etc.), can free users from learning complex query language and data schema before they issue a query. Therefore, XML keyword search is becoming more and more popular in recent years [85, 31, 62, 99, 36, 88, 64]. With XML keyword search, users can easily issue a keyword query in the same way they use any web search engine.

Example 1.2. *If we want to search for the phone number of supplier "Alps" in the XML data tree in Figure 1.2, we can simply issue a keyword query "Alps phone". According to the existing XML keyword search methods, like LCA [85], SLCA [99] or ELCA [31], the result being returned will be the subtree rooted at node supplier:0.1, which contains the information of the required supplier, like phone number, supplier id, etc.*

Comparing structured queries and keyword queries on XML data, we can see that, keyword queries is much easier to use and more user-friendly. However, XML keyword search still faces some challenges on how to enhance the usability for keyword search users.

1.2 Research Problem: Enhancing the Usability of XML Keyword Search

Inspired by the great success of keyword search on web, keyword search on XML data has emerged and is becoming more and more popular. XML keyword search has attracted a lot of research effort and been abundantly studied in the last ten year. Existing research works mainly focus on two topics: defining what should be returned as results (matching semantics) and designing efficient algorithms for a certain matching semantics. Unlike web search, where the data is a set of documents, XML keyword search mainly focuses on how to extract the desired information from one single XML document which is organized in a hierarchical structure. Therefore, the first job of XML keyword search is to define the matching semantics, i.e., what should be returned as results for a keyword query. All existing matching semantics so far, such as SLCA [99, 36], ELCA [31], entity-based SLCA [64] are all based on the concept of lowest common ancestor (LCA). The basic idea of LCA is to find the smallest subtree which contains all the keywords in users' query. Both SLCA and ELCA try to define a subset of LCA which is regarded as meaningful. Besides, another part of research effort focuses on the proposals of efficient result retrieval methods based on a certain matching semantics. For example, [62, 99, 88, 64] improve the result retrieval methods for computing SLCA nodes and [31, 110] for computing ELCA nodes.

However, in XML keyword search, how to reduce the gap between users' search intention and the query results remains a challenge. Even for the mature web search, users have to reformulate and resubmit their queries 40% to 52% of the time in order to get what they want [86]. Therefore, enhancing the usability of keyword search by handling the mismatch between users' search intention and the query results is an important issue, no matter for web search, XML keyword search, or any other kind of keyword search. If we do not detect the mismatch between users' search intention and the query results, users will be confused by the mismatch results returned by the search engine. For example, in XML keyword search, if what users search for is unavailable in the XML data, existing keyword search methods will still return a list of mismatch results, which will confuse the users. This is because existing keyword search methods simply return the smallest subtrees in the XML data which contain all the query keywords. But they do not consider users' search intention and detect the mismatch between users' search intention and the query results.

Example 1.3. *For the XML data in Figure 1.2, suppose a user wants to search for a yellow pencil in the inventory data, she may issue a query $Q = \{$'pencil','yellow'$\}$ to search for a pencil. Unfortunately, no pencil can meet all her requirements. The only available color for pencil is black. However, existing keyword search methods, such as LCA [85], SLCA [99], ELCA [31] or even the most recent variant [51] of LCA, still can find some subtrees containing all the query keywords as results. One query result is the subtree rooted at category:0.0.0, where keyword 'pencil' matches one item while the keyword 'yellow' match another item. Obviously, the subtree rooted at* category *is not expected by the user. It contains too much irrelevant information, i.e. all items under a category. Therefore, simply returning the smallest subtree containing all the query keywords without inferring users' search intention could lead to mismatch results, which will confuse users.*

As we can see, without considering users' search intention during XML keyword search could lead to some mismatch results. It is confusing and time-consuming for users to read and understand such mismatch results. So a solution to detect the mismatch results and provide some informative suggestion to users is in demand.

Besides, after the results are retrieved from the search engine, it needs to be presented to the user. To further bridge the gap between users' search intention and the query results, we find that how to present the results in a proper way is also an important issue. It plays an important role in users' digesting and exploring of the query results. The traditional way of XML keyword search is to return and show a list of independent subtrees as query results. However, it is insufficient to meet the information needs of users because it does not consider the fact that all the results are actually interconnected within a single XML tree. Showing the results as some independent subtrees is imprecise and could be misleading. Users may understand the results wrongly and have difficulty picking up the most suitable results from the result list.

Example 1.4. *For the XML data tree in Figure 1.2, a query "pencil black" will get the following results by LCA:*

1. *Subtree rooted at node item:*0.0.0.1*, which contains keywords "pencil" and "black".*

2. *subtree rooted at node item:*0.0.1.1*, which contains keywords "pencil" and "black".*

Traditional XML keyword search method will return and show the above two results as two independent subtrees. Without showing the relationships among these two results, it is hard to know that these two results are actually belonging to two different categories. One is a normal pencil belonging to stationery *category while the other is a make-up pencil belonging to* make-up *category. Therefore, the traditional way of showing query results as independent subtrees could be misleading and imprecise. A proper way for result presentation is in demand.*

From the example above, we can see that all the data in an XML tree is interconnected by the hierarchical structure. Therefore, each query result of XML keyword search is a part of the XML data tree rather than a piece of independent information. Among the query results (subtrees), they may have sibling or containment relationships. Without showing such relationships, the results could be misleading and imprecise. Users will misunderstand the results and it will hurt the usability of XML keyword search.

Therefore, we need a solution to detect the mismatch results in XML keyword search and give useful suggestion to users, as well as providing a proper and precise way to visualize the query results. It will help reduce the gap between users' search intention and the query results, which is crucial for improving the usability of XML keyword search.

The intuitive idea of our solution addressing such problems is (1) to infer users' search intentions and examine the actual query results for possible mismatch, then generate helpful suggestion based on the available data; (2) to provide users an interactive mechanism for browsing and exploring the query results in a context of the whole XML document.

1.3 Contributions of This Dissertation

In this dissertation, we focus on improving the usability of XML keyword search by reducing the gap between users' search intention and the query results. We tackle the problem in two aspects, namely mismatch caused by result retrieval and mismatch caused by result presentation. First, we will try to detect and solve the mismatch in the query results over the XML tree model. Then we will propose a novel approach to transform an XML IDREF digraph to an XML tree model, such that our solution on XML tree can be applied to the XML IDREF digraph as well. Second, for query result presentation, we propose a map-like model for presenting the query result in a proper way within the global context of the whole XML document and in an interactive way.

1.3.1 MisMatch Problem in Keyword Search over XML without ID References

If we do not consider the ID references in an XML document, then the XML document can be modeled as a tree. Most of the research efforts in XML keyword search are focusing on the XML tree model. As we have discussed in the previous section, existing keyword search methods [99, 36, 31, 64] are all based on the concept of lowest common ancestor (LCA). They will all try to return a set of subtrees containing all the query keywords as query results, regardless of users' search intention. Even what users search for is unavailable in the XML data, they are not able to be aware of such a fact and will still return a list of erroneous mismatch results to users. We call this MisMatch problem in XML keyword search. In this case, it poses three challenges for a search engine to help users: (1) how to design a detection method to distinguish queries with the MisMatch problem from those without; (2) how to explain why the query leads to mismatch results; (3) how to find good suggestions, and what should be a good way to present them to users.

Our solution to the MisMatch problem is based on two novel concepts that we introduce: 1) *Target Node Type*, which is used to infer users' search intention and detect the MisMatch problem; 2) *Distinguishability*, which is exploited to measure the importance of users' query keywords and help generate helpful suggestions to users. Our approach has three noteworthy features: (1) for queries with the MisMatch problem, it generates the explanation, suggested queries and their sample results as the output to users, helping users judge whether the MisMatch problem is solved without reading all query results; (2) it is portable as it can work with any LCA-based matching semantics and is orthogonal to the choice of result retrieval method adopted; (3) it is lightweight in the way that it occupies a very small proportion of the whole query evaluation time.

1.3.2 MisMatch Problem in Keyword Search over XML with ID References

XML documents usually contain some ID nodes and IDREF nodes to represent reference relationships among the data. If the ID references in an XML document are considered, an XML document is usually modeled as a digraph by existing works, where the keyword query results are computed by graph traversal [37, 26, 44, 35]. We call such a graph as XML IDREF digraph. Then the keyword search problem on an XML IDREF digraph is reduced to the problem of finding Minimal Steiner Tree (MST) or its variants in a digraph, where an MST is defined as a minimal subtree containing all query keywords in either its leaves or root. Since this problem is NP-complete [28], a lot of works are interested in finding the "best" answers of all possible MSTs, i.e. finding top-K results according to some criteria, like subtree size, diameter etc.

As compared to keyword search over XML tree model, keyword search over XML IDREF digraph poses new challenges. Since finding all MSTs in a graph is an NP-complete problem, efficiency is one of the notable issues. But more importantly, the matching semantics, i.e. MST, is also defined without considering users' search intention. Therefore, mismatch results are still possible to be returned by existing methods in keyword search over XML IDREF digraph.

To solve the MisMatch problem for keyword search over XML IDREF digraph, we propose a novel method to transform an XML IDREF digraph with ID/IDREF to a tree model, such that we can exploit the XML tree search methods to work on XML IDREF digraph, and subsequently our MisMatch solution designed for XML tree still applies to XML IDREF digraph. We transform an XML IDREF digraph to a tree model by virtually replicating the subtrees being referenced. Our tree model consists of two parts: an XML tree and a table (called reachability table), which is capable of handling different kinds of reference patterns in an XML IDREF digraph.

1.3.3 Query Result Presentation

To further reduce the gap between users' search intention and the query results, how to present the query results in a proper way also plays an important part. We find that, the traditional way of presenting the query results as a list of independent subtrees is imprecise and could be misleading. Actually each query result of XML keyword search is a part of the XML data tree rather than a piece of independent information. Among the query results (subtrees), they may have sibling or containment relationships. Without showing such relationships, users may misunderstand the query results and digest the information wrongly.

To improve the usability by addressing the above issues, we propose a map-like model for presenting the query results in the global context and in an interactive way. It can work as a complementary component of the XML keyword search engine. We present the query results in the context of the whole XML document such that users can clearly view the context and the relationship among the query results. Besides, an interactive mechanism is also provided for user to further explore the query results.

The works included in this thesis have resulted in a number of publications, more specially, [102] and [104], [103], [105] and [101].

1.4 Thesis Outline

This dissertation is organized follows.

- Chapter 2 presents the related work. The surveyed topics include XML query languages, XML labeling schemes, XML structured queries, XML keyword queries for both labeled tree and directed graph models, query refinement and query results presentation.
- Chapter 3 studies the mismatch results in XML keyword search without considering ID references.
- Chapter 4 talks about how to extend our mismatch solution to XML keyword search with ID references considered.
- Chapter 5 discusses our solution to present the XML keyword search results in a proper and interactive way, which allows users to manipulate and further explore the query results.
- Chapter 6 concludes the thesis with future work.

CHAPTER 2

RELATED WORK

XML keyword search has been studied for more than ten years. In this chapter, we are going to review the literature related to XML keyword search. As XML has become the standard of information representation and ex-change over the Internet, querying XML documents has attracted a lot of research efforts. There are mainly two kinds of queries on XML data, namely structured queries and keyword queries, both of which will require some labeling scheme to accelerate the query processing. Due to the intrinsic ambiguity of keyword search, query refinement and query result visualization are also important to improve the user experience. In the following sections we will review the related work on each of the above related topics.

2.1 Labeling for XML

During the processing of structured queries and keyword queries on XML data, it needs to uniquely identify each XML node as well as determining the structural relationship between any two nodes (e.g., Ancestor-Descendant (AD) relationship or Parent-Child (PC) relationship). To server such a purpose, many works focus on how to assign each node in an XML tree a special label, such that the structural relationship between two nodes can be easily inferred by just comparing the labels, meanwhile the label size should be kept as small as possible.

Basically there are three categories of labeling schemes, i.e. containment labeling scheme, Dewey labeling scheme and dynamic labeling scheme.

In containment labeling scheme [106], each node in the XML tree is assigned a label (*start*, *end*, *level*), where *start* and *end* denote a range that contains all its descendants' ranges and *level* denotes the level of a node in the XML tree. For example, if a node n is an ancestor of a node m, then the following property must holds: $start_n < start_m < end_m < end_n$. Therefore, the relationship between two nodes can be easily calculated:

- Ancestor-Descendant (AD) relationship. Node n is an ancestor of node m if and only if $start_n < start_m < end_m < end_n$.

- Parent-Child (PC) relationship. Node n is the parent of node m if and only if node n is an ancestor of m and $level_n = level_m - 1$.

Another labeling scheme widely adopted is Dewey labeling scheme [90]. The label for each node in the XML tree is formed by concatenating the label of its parent with its own local order. In other words, a Dewey label represent a unique path from the root node to that node. Take the XML tree in Figure 2.1 as an example, the Dewey label of the root node is 0; the first child of the root will be with Dewey label 0.0 and the second child will be with Dewey label 0.1. Given the Dewey label of any two nodes, i.e. node n with Dewey label $a_1.a_2...a_i$ and node m with Dewey label $b_1.b_2...b_j$, the relationship between these two nodes can also be calculated by comparing their Dewey labels:

- Ancestor-Descendant (AD) relationship. Node n is an ancestor of node m if and only if $i < j$ and $a_1 = b_1$, $a_2 = b_2$, ..., a_i=b_i.
- Parent-Child (PC) relationship. Node n is the parent of node m if and only if node n is an ancestor of m and $i = j - 1$.

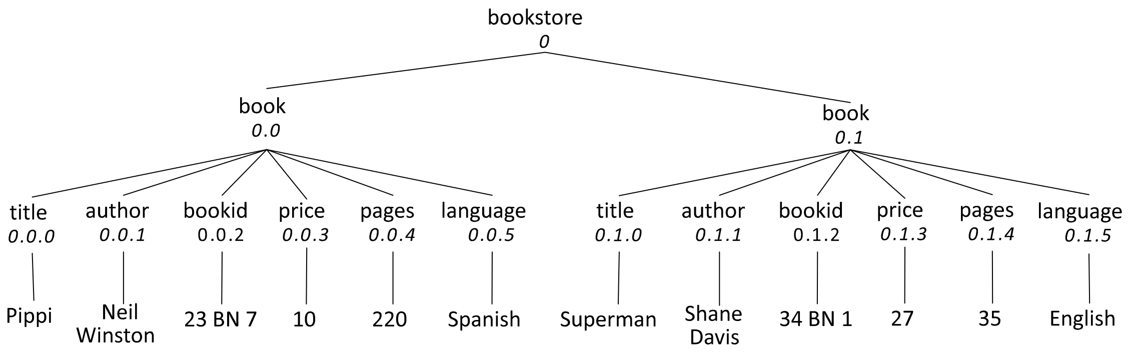


Figure 2.1: A sample XML Tree With Dewey Label (bookstore.xml)

However, containment labeling scheme and Dewey labeling scheme only consider the case of a static XML tree. If some updates are applied to the XML tree, like inserting a node or deleting a node, it will affect the existing labels and some of them will need to be changed accordingly. To cater for the need of labeling an XML tree which will be frequently updated, many dynamic labeling schemes have emerged.

One strategy to avoid relabeling is to reserve some labels for future usage. [60] tried to reserve some space between two adjacent labels. But it may need to relabel the whole XML tree when the reserved labels are used up later on. [78] proposed a hierarchical labeling scheme called ORDPATH, which is a variant of Dewey label. It reserves even and negative numbers for future node insertion. However, label size is not well controlled by such a method. Another strategy to avoid relabeling is to make use of some encoding scheme. Quaternary Encoding for Dynamic XML data

(QED) [55] is proposed to avoid relabeling. It guarantees that there always exist a QED label in between two adjacent QED labels. [97] proposed a vector based labeling scheme, which can also avoid relabeling but achieve better scalability for skewed node insertions. Later DDE (Dynamic DEwey) [98] is proposed with more compact label size and better query performance.

2.2 Structured Query on XML

XML queries can be classified into *structured queries* and *keyword queries*. As a counter part of XML keyword queries, structured queries in XML are similar to SQL queries in relational database. It requires users to have some pre-knowledge of the schema of the XML data before they issue a query. XPath [11] and XQuery [13] are two structured query languages of XML recommended by W3C (World Wide Web Consortium).

XPath [11] is a structured query language where users can specify a path structure as the constraints. Then it will return a node or a set of nodes which satisfy the structure constraints to the users. There are thirteen axes in the XPath specification. Seven of them are most commonly used: ancestor, descendant, parent, child, preceding, following, attribute. For example, "/" denotes parent-child relationship and "//" denotes ancestor-descendent relationship. An XPath expression consists of one or more segments. An expression A/B denotes to find all the nodes with name "B" which has a parent with name "A". For instance, a path expression "bookstore/book/title" issued on the XML tree in Figure 2.1 is to find the title of available books in the bookstore. Then the results being returned will be a set of nodes $\{< title > Pippi < /title >, < title > Superman < /title >\}$.

XQuery [13] is built based on XPath by introducing FLWOR (For-Let-Where-

Order by-Return) constructs to offer more expressiveness. It can be viewed as a an extension of XPath, which allows users to define their own functions. It has been standardized as the major XML query language. For example, the following XQuery expression

FOR $b IN document("bookstore.xml")//book
LET $a := $b//author
WHERE contains ($a, "Winston")
RETURN $b

tries to find the books which is written by Winston.

The essential operation in structured queries processing is twig pattern matching. Twig pattern is a tree specifying the path structure pattern. Twig pattern matching is to find all the instances in an XML tree which satisfy the twig pattern constraint. How to reduce the processing time of twig pattern match has attracted a lot of research efforts [68, 21, 95, 15, 41, 42]. Among them, the holistic join [15] approach and its variants [42, 21, 68, 77, 41] have been proven to be able to avoid producing too many useless intermediate results.

2.3 Keyword Search on XML

In XML keyword search, extensive research efforts have been conducted to find the smallest sub-structures in the XML data that contains all query keywords, in either the tree data model or the directed graph (i.e. digraph) data model.

2.3.1 Tree Model

In tree data model, LCA (lowest common ancestor) semantics is first proposed and studied in [85, 31] to find the lowest XML nodes, each of which contains all query keywords within its subtree. Let $lca(m_1, ..., m_n)$ be the lowest common ancestor of nodes m_1,...,m_n. For a given query $Q = \{k_1$,...,$k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q) = \{v \mid v = lca(m_1, ..., m_n), m_i \in L_i(1 \leq i \leq n)\}$. Extended from Google's Pagerank algorithm for ranking, XRank [31] takes into account the proximity of the keywords and the references between attributes. Its aim is to find the top-k relevant answer. Ranking is one of the important job in this work. First it tries to define what should be returned as the query results. One important property defined in the work is: if a descendant of a answer node is also another answer node, then they cannot share a keyword node (which directly contain the keyword) in their answers. After that a PageRank-similar approach is used to compute the weight of each nodes in the XML document. With the weight, it computes the relevance between a node and a keyword. Then the relevance between a node and a query is measured by the sum of relevance to each keyword in the query. A stack-based algorithm is proposed to compute all the answer nodes in O(n) complexity. But in case of huge documents, inverted list for each keyword might be huge. Therefore, another algorithm, RDIL, targeted at top-k answer is proposed, which keeps finding the answers until no remaining nodes can form an answer with higher relevance than the so-far top-k results.

A variation of LCA is XSEarch [23], which proposed a concept called interconnection. Let n and n' be two nodes in an XML tree T, $T|_{n,n'}$ be the shortest path from n to n', then n and n' are interconnected if one of the following conditions holds:

1) $T|_{n,n'}$ does not contain two distinct nodes with the same label.

2) The only two distinct nodes in $T|_{n,n'}$ with the same label are n and n'.

The intuition of such a property is that it distincts the attributes which belong to different entities. XSEarch tries to find a set of answer nodes, where each answer node should contains all query keywords and every two keyword-matching nodes should be interconnected. However, the complexity for the approach calculating such results is NP-complete. So XSEarch only requires that each keyword-matching node should be interconnected with at least one other keyword-matching node. This looser condition is called star-interconnected and makes it possible to find all the results in polynomial time.

Subsequently, SLCA (smallest LCA [62, 99]) is proposed to find the smallest LCAs that do not contain other LCAs in their subtrees. In other words, SLCA is a node containing all the query keywords while none of its descendant also contains all the query keywords. It is claimed that SLCA is more suitable to be the answers for XML keyword search. To find all SCLAs, normally 2 tasks must be finished: finding all LCAs and remove all ancestor nodes among such LCAs being found. It is costly to find all the LCAs. When the number of keywords increases and the number of nodes containing each keyword increases, the number of combination will be huge. XKSearch [99] optimizes this as it directly finds out SLCAs in one step by following a particular order such that impossible search space is pruned. [99] proposed several algorithms to find the SLCAs efficiently. The first algorithm is called "Indexed Lookup Eager Algorithm". It transforms SLCA-finding problem on a sequence of keywords into a problem that repeatedly find SLCAs of two keywords. It is expressed by the following formula:

$slca(S_1, ..., S_k) = slca(slca(S_1, ..., S_{k-1}), S_k) = slca(slca(slca(S_1, ..., S_{k-2}), S_{k-1}), S_k)$ $= ...$, where S_i is a set of nodes that directly contain the i_{th} query keyword. To compute $slca(S_1, S_2)$, it first sorts S_1 in preorder. Then for each node v_i in S_1, it finds $slca(v_i, S_2)$. It will judge whether $slca(v_i, S_2)$ is in $slca(S_1, S_2)$ by comparing it to $slca(v_{i+1}, S_2)$. Another method proposed in this work is a stack-based algorithm, which is a modification of XRank [31]. It has an additional step to clear the flags in order to rule out of the LCAs which are not SLCAs.

Multiway-SLCA [88] generalized SLCA to support keyword search involving combinations of AND and OR boolean operators. For a query Q with any combination AND and OR operator, it rewrites the query Q in DNF (Disjunctive Normal Form). Then it evaluates the query in two stages: first, it evaluates each disjunction in Q using an existing AND-query evaluation algorithm; second, the results of the individual evaluations are combined by eliminating intermediate SLCAs that are ancestor nodes of some other intermediate SLCAs.

Besides of LCA and SLCA, another matching semantics, MCT (minimum connecting trees), is also proposed. It aims to find the minimum connecting trees by excluding sub-trees not covering any query keyword. Essentially, it checks all combinations of nodes from the inverted lists and computes an MCT (minimum connecting tree) for each combination. Then it merges the resulting MCT into the list of results, called Grouped Distance Minimum Connecting Trees (GDMCTs), whose size is controlled within the user-specified threshold. A stack-based algorithm is also proposed to maintain a minimum amount of information that allows the efficient and timely output of the GDMCTs.

ELCA [31], which is also a widely adopted subset of LCA, is defined as: a node v is an ELCA node of a query Q if the subtree T_v rooted at v contains at least one occurrence of all query keywords, after excluding the occurrences of keywords in each subtree $T_{v'}$ rooted at v's descendant node v' and already contains all query keywords. [56] proposed Valuable LCA (VLCA) by eliminating redundant LCAs that should not contribute to the answer, but also retrieves the false negatives filtered out wrongly by SLCA.

XSeek [64] identifies the return nodes by inferring the pattern of the search keywords. The idea behind is simple but useful. Firstly, it finds out all the matching nodes for each query keyword. Then the keyword-matching nodes are classified into two categories: return nodes and search predicates. For a non-leaf node v matching a query keyword, if none of its descendants is both a value node and keyword-matching node, then v is called a return node. Otherwise it's called predicates. For a query, if return node exists, then the return node and its descendants will be returned as the result. Otherwise it will return the first entity node along the path from the SLCA node to root.

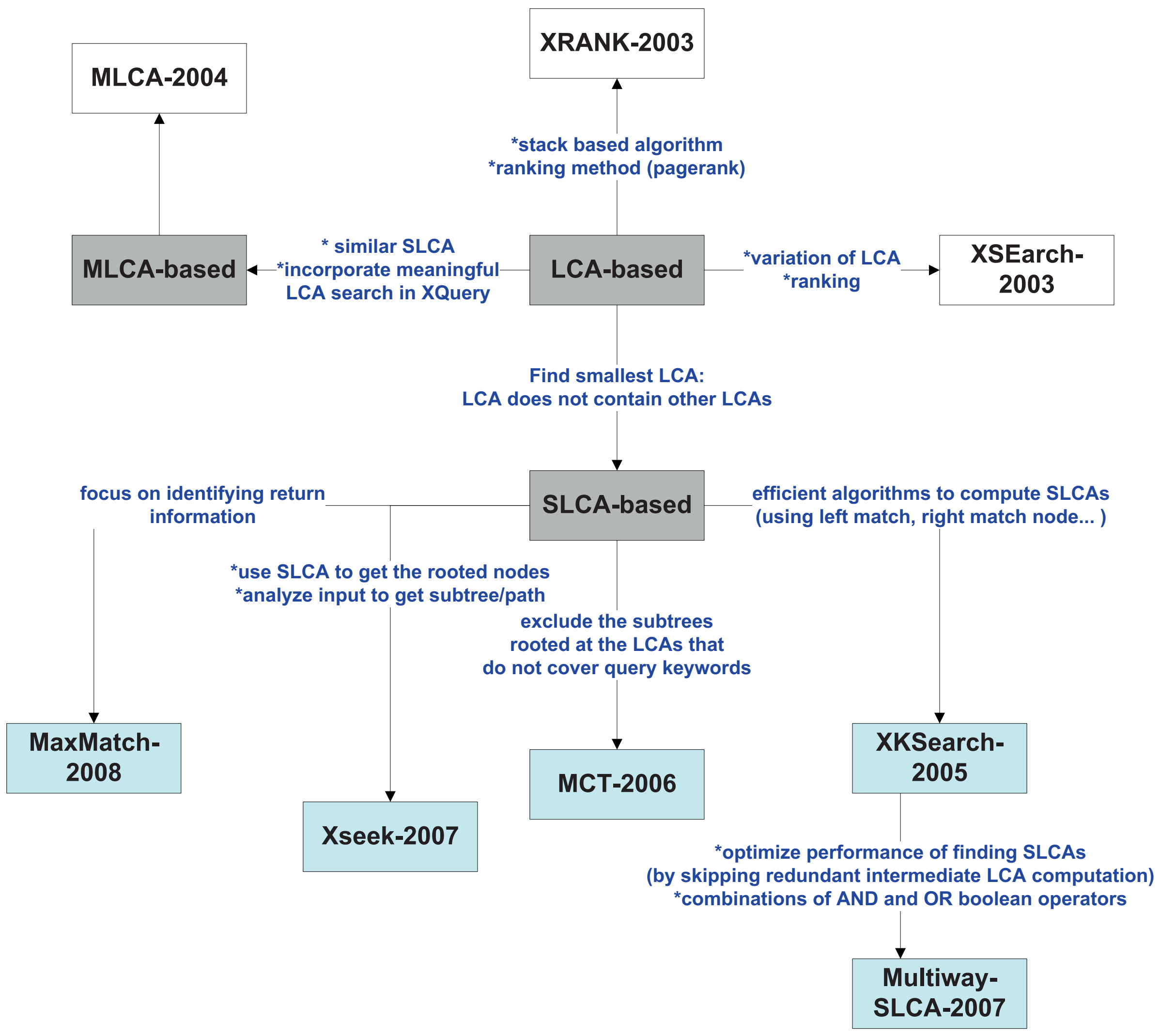

MLCA-2004
XRANK-2003
*stack based algorithm
*ranking method (pagerank)
MLCA-based
* similar SLCA
*incorporate meaningful
LCA search in XQuery
LCA-based
*variation of LCA
*ranking
XSEarch-2003
Find smallest LCA:
LCA does not contain other LCAs
focus on identifying return information
SLCA-based
efficient algorithms to compute SLCAs
(using left match, right match node...)
*use SLCA to get the rooted nodes
*analyze input to get subtree/path
exclude the subtrees
rooted at the LCAs that
do not cover query keywords
MaxMatch-2008
Xseek-2007
MCT-2006
XKSearch-2005
*optimize performance of finding SLCAs
(by skipping redundant intermediate LCA computation)
*combinations of AND and OR boolean operators
Multiway-SLCA-2007

Figure 2.2: Relationship among Main Keyword Search Techniques

Based on SLCA, [65] further proposed an axiomatic way to decide whether a result is relevant to a keyword query in term of the monotonicity and consistency properties w.r.t the XML data and query. This is the first novel algorithm that satisfies both the properties of monotonicity and consistency. [66] studied how to differentiate the search results of an XML keyword query, aiming to save user efforts in investigating and comparing potentially large results.

XReal [8] proposed a statistical way to identify the search target candidates. It proposes an IR-style method to handle the keyword search problem, which is the first one to exploit the statistics of underlying XML database to address search intention identification, keyword ambiguity and relevance oriented ranking as a single problem. Given a query of several keywords, firstly, it tries to find which type of node is most likely the type user is searching for. The nodes of such node type should contain all the keywords in the subtrees and not to be deeply nested in the XML. Secondly, it tries to decide which type is most likely to be the correspondent of each keyword. It's similar to the previous step except that it does not require the node type to contain all keywords and not to be deeply nested. After that a formula is proposed to compute the similarity between an XML node and the query, which is utilized to do the ranking.

Most of the techniques proposed so far are making use of Dewey labeling scheme for query evaluation. Recently, some studies [108, 109] point out that the comparison operation for Dewey labels is one of the most time-consuming operations in XML keyword query evaluation. Some efficient methods for calculating LCA/SLCA/ELCA [108, 109] are proposed to pre-compute some possible common ancestor nodes in order to avoid the comparison operation on Dewey labels.

Figure 2.2 shows some main techniques in XML keyword search and the relationships among them. Figure 2.3 shows a time line for some main approaches.

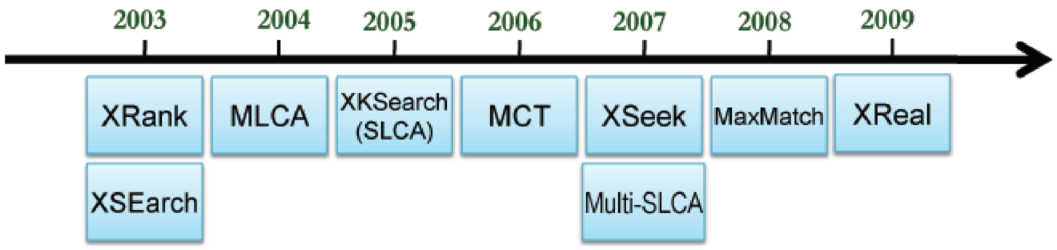


Figure 2.3: Timeline for Main Keyword Search Techniques

2.3.2 Graph Model

An XML document considering ID references is usually modeled as a directed graph (digraph) [37]. Keyword search on a digraph is usually reduced to the Steiner Tree problem or its variants: given a digraph $G = (V, E)$, where V is a set of nodes and E is a set of edges, a keyword query result is defined as a minimal directed subtree T in G such that the leaves or the root of T contain all keywords in the query. The Steiner tree problem is NP-complete [28], and many works are interested in finding the "best" answers of all possible Steiner trees, i.e. finding top-k results according to some criteria, like subtree size (sum of length of all edges in the subtree), diameter (maximum distance between any two nodes in the subtree), etc. Backward expanding strategy is used by BANKS [12] to search for Steiner trees in a digraph. It starts the searching from the nodes which directly contain the query keywords. Then it concurrently runs multiple threads to traverse from those nodes until they find some common nodes which connect to all query keywords. To improve the efficiency, BANKS-II [44] proposed a bidirectional search strategy to reduce the search space, which searches as small portion of digraph as possible. It starts a backward searching from the nodes directly containing the keywords. Meanwhile, it also conducts a forward searching starting from the nodes which have been visited during backward searching. Later [26] designed a dynamic programming approach (DPBF) to identify the top-k Steiner trees containing all query keywords. With some slightly modification on DPBF, a variant of DPBF to output the top-k results in increasing weight order is also proposed in the work. BLINKS [35] proposes a bi-level index and a partition-based method to prune and accelerate searching for top-k results in a digraph. It first divides the XML nodes into several blocks. Then it builds intra-block index and inter-block index for all the nodes. With the index which conveys the connectivity information among and within the blocks, it can prune some unnecessary search space. XKeyword [37] presented a method to optimized the query evaluation by making use of the schema of the XML document. It infers the possible schema structure of the potential results such that it can avoid some search space which will not lead to any results complied with that structure.

2.4 Query Refinement

In this section, we will have a literature study for existing query refinement techniques. We will first study three main techniques in query refinement: query cleaning, query relaxation and query substitution. They are designed to handle different query refinement problems. MisMatch problem is one problem which can be handled either by query relaxation or query substitution. In the end of this section, we will talk about how the MisMatch problem is handled by existing research works in structured data and unstructured data, while there is no work on such a topic on semi-structured data yet.

2.4.1 Query Cleaning

Query cleaning is to correct spelling errors with different kinds of techniques. It is usually done by measuring the difference between wrong keywords and correct keywords in a dictionary.

[80] proposed a method which considers not only the spelling error but also the TF/IDF feature of the data while correcting keywords. It handles both the segmentation problem and spelling correction. Each keyword in a keyword query is corresponding to a set of similar keywords which is already in the database. Picking up one new keyword from each set can form a new query. However, computing all combination is too costly. The paper proposed a dynamic algorithm to get the combinations which is with high quality. To measure the quality of the new queries, three factors are considered: 1) TF-IDF feature of each new keyword 2) length of keyword segments 3) spelling error.

[69] extends the traditional query cleaning techniques to the context of XML considering the structural feature of the XML data. Each keyword in a keyword query is corresponding to a set of similar keywords which is already in the database. Every combination of the new keywords from all these sets is an alternative query. This paper propose a new way to measure the quality of the alternative queries. The quality function is based on a list of factors borrowed from the probabilistic theory and some existing language model: 1) typographical errors (single edit errors); 2) probability of users being interested in a particular query by *unigram language model* and *dirichlet smoothing*; 3) XML tree structure and keyword query semantics by the formula proposed in XReal [8], which help to identify the units of information in the XML data and measure the quality of query in a finer granularity. Then the authors combine all the above factors into one final formula to measure the quality of each alternative query.

2.4.2 Query Relaxation

Query relaxation is to delete some of the query constraints to get more results. [70, 76] proposed methods that measure the importance of each attribute in the database with the help of approximate functional dependencies, which is computed by data samples in the database. First it makes a dependence graph between attributes and perform a topological sort over the graph. Functional dependencies can be used to derive the attribute dependence graph that is needed. But, full functional dependencies with 100% support between all pairs of attributes are often not available. Therefore approximate functional dependencies is used between attributes to develop the attribute dependence graph with attributes as nodes. Attributes that can determine more attributes will have higher importance. Then according to the importance of each attribute, they try to reduce the constraints on an existing query.

[73, 74] proposed another method which adopts the machine learning way to learn some rules from the database. It learns decision rules that express the implicit relationships among the various domain attributes; then it uses nearest neighbor techniques to identify the learned rule that is most similar to the failing query; finally, it uses the attribute values from this most-similar learned rule to relax the constraints from the failing query. For each failed query, with the predicted attribute whose value is more likely not able to be satisfied, it comes up with another similar query which ensures to get nonempty results. The target of all query relaxation methods is to ensure more relevant results are retrieved.

2.4.3 Query Substitution

Query substitution is to replace some keywords of the original query which gets no result. [43] proposed a method which identified related queries and phrases by

	triggering condition	processing phase	goal	major solution	get sample result without one more query
Query Cleaning	typos exists/no result	post processing	correct typos	to find similar-spelling words	no
Query Relaxation	no result	post processing	get more results	to delete some keywords	no
Query Substitution	no result	post processing	get more results	to replace with synonyms	no

Figure 2.4: Comparison of Query Refinement Approaches

user query logs. The training data used comes from logs of user web accesses. This data contains web searches annotated with user ID and timestamp. A candidate reformulation is a pair of successive queries issued by a single user on a single day. Because it is believed that successive queries from a single user is a reformulation of the same query. Then pair independence hypothesis likelihood ratio is proposed to measure the importance of each query pair. Pairs with high importance suggests that there is a strong dependence between the pair of queries. Then a coming new query will be broken up into segments, and replaced by some segments which is statistically significant related. In other words, this approach will try to modify the query by some pre-computed queries and phrases based on similarity, which is given by the machine learning model. Later, [107] proposed methods to improve query substitution by selecting a better training set for the machine learning model. It selects the most informative samples to train the relevance model for query rewriting according to a new linear regression model.

To summarize, we compare the main features of the available keyword query refinement techniques according to some criteria, as shown in Figure 2.4. We can see that all the above techniques operate at a shallow level. I.e., whether there is any typo or whether the result is an empty set. They do not consider users' search intention, nor do they consider the mismatch between the results and such intention.

2.4.4 MisMatch Problem in Structured and Unstructured Data

When users issue a query to a database, they have expectations about the results. If what they search for is unavailable in the database, the system will return an empty result or erroneous mismatch results. This is called the MisMatch Problem. Users will need some help and suggestions in such a case. How to detect such a problem and generate useful suggestion becomes a research problem, which may involves multiple query refinement techniques. To the best of our knowledge, so far there is no work on the MisMatch problem in XML keyword search. So we will look at how the MisMatch problem is handled in either structured data and unstructured data in information retrieval.

MisMatch Problem in Structured Data. When structured queries are issued over structured data (relational tables), the MisMatch problem (i.e. what users search for is unavailable in the database) leads to empty result. E.g., a structured query $select * from\ T\ where\ A =' 11'\ and\ B =' 20'$ will return empty result if no tuple in table T is with attribute A being 11 and B being 20. Users need some help on how to revise the query to get some results, either by dropping or modifying some constraints. It has attracted a lot of research efforts such as [38, 76, 73, 74], where the problem is also known as failing queries, non-answer queries. [76] proposed a method to remove some constraints of the query with the help of approximate functional dependencies, and then execute the new queries to finally find some alternative queries. [73, 74] proposed another method which adopts the machine learning way to solve the problem. Given a structured query, it will first learn some decision rules related to the query from database. Those decision rules are about the condition for a particular query constraint to hold learned from the data. E.g., a rule could be like $AttributeA > 60$ *and* $AttributeB \neq$ "$female$" $\Longrightarrow AttributeC > 200$. Then according to those decision rules, it will change the constraints in the query in order to make it not lead to empty result. Recently, some research works [38, 18, 92] have been done trying to pinpoint the constraint in the structured query which causes the empty result, such that we can explain to users why empty result is returned. [38] tried to identify the attribute of the tuples which are excluded from the final result set (empty set). It is done by changing the attributes of the excluded tuples until they are included in the result set. [18] proposed an approach to explain the empty result by pinpointing the manipulation operation(s) in the query plan that excludes the missing tuples. Later [92] pointed out that it will be more helpful if the database can even provide a refined query, which is formed by using some optional predicates to replace the original predicates in the query.

MisMatch Problem in Unstructured Data. When keyword queries are issued over unstructured data (in web search), the MisMatch problem will lead to a list of mismatch results. E.g., a keyword query "Vaio W red" in order to find a laptop model $Vaio\ W$ with red color may not get any meaningful result if such a color is unavailable for such a model. In web search, results being returned could be a lot of web pages containing those three keywords. However, "Vaio W" could be in one part of the web page while "red" appears in another part of the web page. Those are some mismatch results which are not expected by users. Detecting such a MisMatch problem requires to analyze whether the keywords are ‘semantically’ related in the results. Such analysis is challenging because the data is unstructured. One way to alleviate the problem is to mine some similar and popular queries from query log, like “Vaio W blue” could be a similar query in the query log as “Vaio W red”. The drawback of such an approach is also obvious: those popular queries mined could also lead to mismatch results. [43] tried to modify the query by some pre-computed queries and phrases based on user query log and similarity, which is given by a machine learning model. Later, [107] proposed methods to improve query substitution by selecting a better training set for the machine learning model. However, such similar queries mined from query log cannot be guaranteed to have meaningful results themselves.

Since the results of XML keyword search are very different, which are some subtrees with structure, none of the above techniques consider tree structure and can be used to detect MisMatch problem in XML keyword search.

2.5 Query Results Visualization

Query results visualization is an indispensable part of a keyword search engine. It takes charge of how to present the query results to users, which will directly affect the readability of the query results and the usability of the search engine. Meanwhile, it also offers possibilities for user interaction, which allows users to further manipulate and explore the query results. Therefore, it has attracted a lot of research efforts [87, 25, 24, 53, 34, 16, 17, 72, 19].

Query results visualization is closely related to the form of data and the form of results. Different forms of data and query results require different ways for optimal visualization. There is no single way which can work well on all forms of data.

Although it has attracted a lot of research efforts in web search, in terms of XML keyword search, so far there is very few works on results visualization for XML keyword search.

There are mainly three categories of visualization techniques for web search, namely list-based approach, graphics-based approach and hybrid approach of the pervious two. The list-based approach keeps the traditional ordered list visualization adding visual aids such as bolding words in the paragraphs [45] or clustering web pages and presenting a tree view [53, 19] along with the list. The graphics-based approach represents search results in a graphical environment where the visualization can either be 2D [19, 87] or 3D [16, 72]. The hybrid approach integrates the previous two approaches. For example, Google Maps [3] is a typical application which equipped with the hybrid approach for result visualization. When presenting the query result, it shows the traditional result list as well as an interactive graphical interface.

Most of the research work is focusing on the list-based approach and graphics-based approach since the hybrid approach is obvious and intuitive. Some of the works are based on the idea of presenting the results in a spiral figure. RankSpiral [87] focuses on displaying the results of one single query from multiple search engine to users in a graphical way. It exploits a spiral figure and displays all the titles of the retrieved documents on the spiral. It allows users to browse a large amount of information in one screen and examine document clusters in more detail. [25] tries to apply a similar approach to an image search engine for image browsing. NIRVE [24] is proposed to handle the visualization task where there are a lot of search results. It draws the icon of each search result on a 3D spiral with the highest ranked documents in the center. It also provides a tool called Keyword Sliders which allow the user to accentuate documents which contain the keywords considered to be especially important.

Besides, some of the works are presenting the results in an interactive work space. Lighthouse [53] is hybrid approach which integrates the traditional ranked result list and the clustering visualization. It visualized the documents as clustered circles which can reflect the relevance among the documents. Meanwhile, it also accepts users' relevance feedback to further refine the approach. It is claimed to be able to help users locate the interesting documents among all the results. Sparkler [34] is a graphics-based approach to present multiple result sets of multiple queries on the same screen. It makes use of a bulls eye layout with star plots, where a document is plotted on each star spoke based on its rankings by the queries or search engines. Users can progressively improve their queries by looking at the difference between different result sets. Such an approach also provides an interactive component for users to manipulate and explore the results. WebBook/WebForage [16] organized the web search results in a 3D workspace, which enables users to manipulate the results in an interactive way. Users can group the retrieved documents to a group in the form of a book. Order is also adjustable for all books and documents. WebQuery [17] aims to visually show the pages related to the result set in a graphical way. I.e., pages which are hyper-linked by the search results will be shown to users. Users can view the connectivity among the web pages and explore the related pages following the graphical connectivity links. CardVis [72] introduces card metaphor to visually displays web search results with additional related terms. It is suitable for the case that the results can be organized as a collection of disconnected graphs. The visualization presents the details of the structure and contents of the focus graph. MetaSpider [19] tries to improve the precision of query results by further processing the retrieved documents from primary search engines. All the operations are post-processing. It includes validation, indexing, and categorizing. Then it will draw the cluster results on a self-organizing 2D map.

CHAPTER 3

MISMATCH PROBLEM IN KEYWORD SEARCH OVER XML WITHOUT ID REFERENCES

3.1 Introduction

When users issue a query to a database, they have expectations about the results. If what they search for is unavailable in the database, due to reasons like product removed from shelves, clothes size unavailable, etc., the result they seek may not be found in the database. In such a case, the system will return an empty result or erroneous mismatch results. We call this the *MisMatch Problem*.

For example, a user wants to search for a laptop. She wants the model Vaio W with color being red. If red color is unavailable for laptop Vaio W in the database, then obviously the user will not get what she wants no matter how the data is organized or what kind of query it is.

The MisMatch problem is a natural and common problem. It can happen in any form of information retrieval over data of any structure, i.e. can be either structured query or keyword query on structured, unstructured and semi-structured data. Such a problem has attracted a lot of research effort in the context of structured queries on structured data [38, 18, 76, 73, 74], with descriptions such as failing queries and non-answer queries. However, no such work has been done in the context of keyword search on semi-structured data. This is an important area to address. According to our experiments conducted on XClear, an XML keyword search engine available at [104], users suffered from such a problem for 27% of their queries. This is our central concern in this chapter.

What can we offer to help the user? Ideally, we can get the following help if we are interacting with a human:

1. Notification: "Sorry, we do not have such a product."
2. Explanation: "Because red color is unavailable for Vaio W."
3. Suggestion: "You can choose some other available colors: black, blue and white."

When structured queries are issued over structured data (relational tables), the MisMatch problem (i.e. what users search for is unavailable in the database) leads to empty result. Detecting the problem is trivial because empty result is obvious. A message (notification part) will be given to users. Some existing works [38, 18] try to explain the non-answer queries by pinpointing the constraint causing the empty result (explanation part). Some works [76, 73, 74] focus on generating some alternative constraints to come up with some suggested queries (suggestion part).

When keyword queries are issued over unstructured data (in web search), the MisMatch problem will lead to a list of mismatch results. It is even difficult to detect

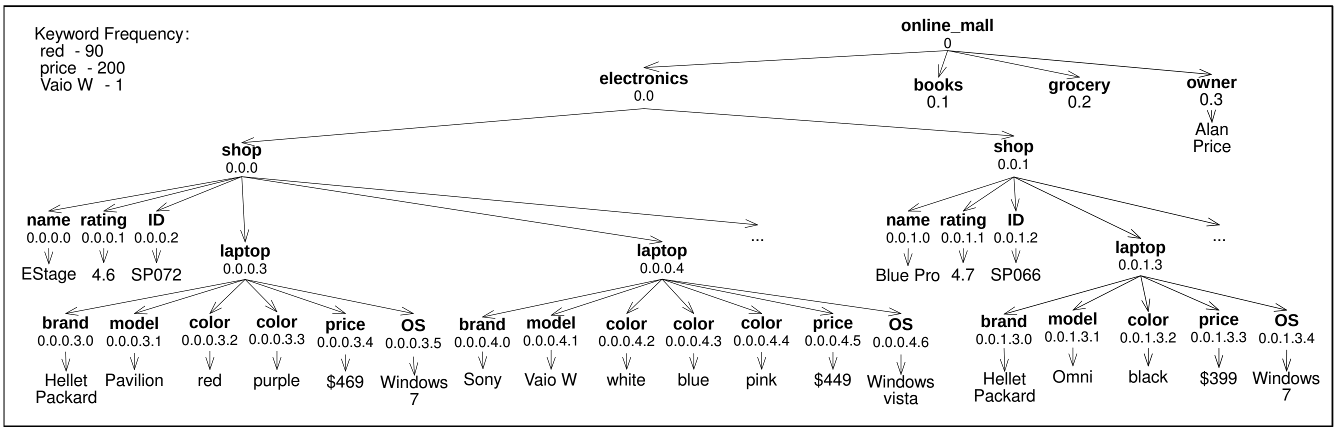


Figure 3.1: Sample XML Document about an Online Shopping Mall

the problem in the first place. Because most likely the results being returned are not empty. It could be the case that the query keywords appearing in one document are far away from each other and not semantically related. E.g., for a keyword query 'Vaio W red', if color red is not available for laptop Vaio W, there still can be many webpages being returned, where 'Vaio W' appears in one part of the webpage while 'red' appears in another part of the webpage. It leads to mismatch results. Therefore, we need to analyze whether the keywords are 'semantically' related in the results. Such analysis is challenging because the data is unstructured. A limited solution to a part of the problem (only the suggestion generation part) is to mine some similar and popular queries from query log [43, 107] and show them to users (suggestion part). But the downside is that such popular queries do not guarantee to have reasonable results.

In this chapter, we focus on identifying and solving the MisMatch problem in the context of keyword search over XML data which is without ID references. For XML data with ID references, we will study it later in Chapter 4. Now, let us take a look at how the MisMatch problem behaves in XML keyword search without ID references.

Example 3.1. *An XML data tree without ID references in Figure 3.1 describes* *the item information of an online shopping mall. Suppose a user wants to buy a laptop. She prefers Sony's Vaio W with red color, and wants to know how much it is. Then she may issue a query* $Q = \{$*'Vaio','W','red','price'*$\}$ *to search for a laptop. Unfortunately, no laptop can meet all her requirements. Vaio W only has three colors: white, blue and pink. Existing keyword search methods, such as LCA (Lowest Common Ancestor) [85], SLCA [99], ELCA [31] or even the most recent variant [51] of LCA, still can find some results containing all query keywords. One of the query results is the subtree rooted at shop:0.0.0, where keyword 'red' matches one laptop while the rest keywords match another laptop. Obviously, the subtree rooted at* shop *is not expected by the user, as it contains too much irrelevant information, i.e. all laptops. What is worse, there could be hundreds of shops selling Viao W and therefore hundreds of mismatch results are returned. In this case, imagine if the user was interacting with a salesman, she would be informed of the unavailability of the product and suggested with some other available colors for the laptop Vaio W.* □

As we can see, the MisMatch problem in XML keyword search also leads to a list of mismatch results. It poses three challenges for a search engine to help users: (1) how to design a detection method to distinguish queries with the MisMatch problem from those without; (2) how to explain why the query leads to mismatch results; (3) how to find good suggestions, and what should be a good way to present them to users.

Our solution to the MisMatch problem is to run a small post-processing job at the end of the query evaluation, consisting of two components, namely *detector* and *suggester*. The former addresses the first challenge above, and the latter addresses the remaining two. The reason for a post-processing solution is that we want to make our solution as general as possible and can be applied to any existing XML keyword search method.

The central idea of our technique for mismatch detection is based on the notion of *Target Node Type* (see Section 3.3 for the formal definition). Intuitively, Target Node Type denotes the type of node a query result r intends to match. We calculate it at schema level. Meanwhile, the actual root of result r is calculated at data level by existing techniques. If r's root does not match its Target Node Type, we claim that r misses the target. We can perform a similar check on all results of a query Q. If all results of a query Q miss their targets, then we say that Q has the MisMatch problem.

Once a mismatch is detected, we propose a concept called *Distinguishability* to find 'important' keywords in the original query, and use these to explain the reason for the mismatch and to suggest possible relaxations. Distinguishability is inspired by the *tf*idf* scoring measure proposed in IR [83] while taking the structural property of XML data into account. Then based on each query result r we try to find some 'approximate' query results, which contain these 'important' query keywords and are structurally consistent with r, while having reasonable replacement for the rest 'less-important' query keywords. Finally, the explanation and suggested queries can be inferred from the approximate results. To further improve the user experience, our suggester also generates a sample result for each suggested query Q' even without evaluating the query Q', which helps users to judge whether Q' is helpful.

Putting these together, we have our complete algorithm. The input of our algorithm is a (ranked) list of all results returned by search engine. For a user query that has the MisMatch problem, the output of our algorithm consists of three parts:

1. An explicit notification to user: "what you search for is not available".

2. An explanation on which keyword(s) in the query leads to mismatch results.

3. Some data-driven suggested queries, which guarantee to have reasonable results.

Note that there are many possible relaxations of a given query, and many of these may themselves also be empty (result in mismatch). It is important to ensure that the suggestions given have at least some results and are not mismatch themselves.

As discussed in the related work section, there is a great body of work on query relaxation and on generating partial match answers. These systems, while valuable, do not address all three of the challenges we described above, and hence are not suited for our problem context. In particular, many of them generate large lists of possible partial match answers that the user has to wade through even to realize that there is a mismatch at all.

In summary, our major contributions in this chapter include:

1. We identify the MisMatch problem in XML keyword search. We detect the MisMatch problem by investigating into the query results and inferring the Target Node Type for each query result. It is *portable* as it can work with any LCA-based matching semantics and is orthogonal to the choice of result retrieval method.

2. We design a *data-driven* approach to generate explanation and suggested queries by finding approximate query results, which contain important keywords in the original query Q while having consistent structure with the results of Q. We propose Distinguishability, which is a structure-aware tf*idf scoring measure, to quantify the importance of keywords.

3. We propose a novel bitmap-based labeling scheme to accelerate finding approximate results. As a result, the MisMatch detector and suggester is

lightweight: it takes only 4% of the whole query processing time.

4. We build a search engine called XClear [104] which embeds the MisMatch problem detector and suggester. Extensive experiments have verified the effectiveness, efficiency and scalability of our method.

We present preliminaries in Section 3.2. Detecting the MisMatch problem is in Section 3.3. Section 3.4 discusses how to find the explanations and suggested queries. Section 3.5 presents our labeling scheme for efficient approximate results detection. Section 3.6 presents indices and algorithms. Experiments are in Section 3.7 and Section 3.9 summarizes this chapter.

3.2 Preliminaries

3.2.1 Semantics and Data Model

We will first make the assumption of semantics we have for an XML document and also define the data model we use for an XML document without ID references. In our solution, we assume that there is no outer semantics provided, which is also the assumption for most of the existing works in XML keyword search. In other words, we assume that we only have the XML document itself with the accompanied schema specification, like DTD or XML Schema. The following information from DTD or XML Schema will be used in our solution: 1) which attributes in the XML document are the ID attributes or IDREF attributes (to identify ID reference links); 2) the number of possible occurrences of a sub-element/attribute appearing under a parent node (to be used in our MisMatch solution). Outer semantics, such as ER model [20], ORA-SS model [27, 63], can help identify in the XML data that which nodes represent objects, object attributes, relationships, relationship attributes, etc. Such information is not conveyed in an XML document or its accompanied schema specification. Instead, it only stores the data in a nested tree structure. Therefore, based on the information in an XML document without ID references, we model it as a rooted, labeled and ordered tree. Each node of the tree corresponds to an element of the XML data, and it has a tag name and (optionally) some value. Each edge of the tree corresponds to the containment relationship between a parent node and a child node. An edge means a parent node and a child node is related, while it is not specified how they are related. E.g., a parent node with tag name *house* and a child node with tag name *person* just means these two nodes are related while it is unknown that whether the person owns the house or the person rents the house.

Without loss of generality, we simply use the word "node" to mean the node in an XML tree. To accelerate the keyword query processing, all existing works adopt the dewey labeling scheme [93]. As shown in Figure 3.1, for a node n its dewey label consists of a sequence of components that implicitly contain all ancestor nodes on the path from the document root to n. E.g., from laptop:0.0.0.3, it is easy to find that the label of its parent is 0.0.0.

Definition 3.1. ***Node Type.*** *The type of a node n in an XML tree, denoted as $n.type$, is the tag name path from root to n.*□

In the rest of the chapter, the tag name of n is used to represent the node type of n if no ambiguity is caused.

Definition 3.2. ***Keyword Match Node.*** *A node n is called a* keyword match node *for a keyword k if the tag name or the value part of n contains k.* □

Definition 3.3. ***Subtree-contain.*** *A node n is said to* subtree-contain *another node m if n equals to m or there exists a directed path from n to m. n is also said to* subtree-contain *the keywords in m's tag name or value part.*

E.g., in Figure 3.1, the node type of *laptop*:0.0.0.3 is *online_mall/electronics/shop /laptop*; *color*:0.0.0.3.2 is a *keyword match node* w.r.t. keyword 'red'; *laptop*:0.0.0.3 is said to *subtree-contain* node *color*:0.0.0.3.2; *laptop*:0.0.0.3 is also said to *subtree-contain* keyword 'red' as 'red' is the value part of *color*:0.0.0.3.2.

3.2.2 General Query Result Format

To define a general format to represent the query results, let us look at the existing matching semantics first. All existing matching semantics so far, such as SLCA [99, 36], ELCA [31], entity-based SLCA [64] are all based on the concept of lowest common ancestor (LCA). Let $lca(m_1, ..., m_n)$ be the lowest common ancestor of nodes m_1,...,m_n. For a given query $Q = \{k_1,...,k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q) = \{v \mid v = lca(m_1, ..., m_n), m_i \in L_i (1 \leq i \leq n)\}$. Both SLCA and ELCA define a subset of $LCA(Q)$, and we refer readers to Chapter 2 for detailed definitions of SLCA and ELCA, and their relationships with LCA.

Definition 3.4. ***Query Result Format.*** *For a keyword query Q=$\{k_1, ...,k_n\}$, we define the format of a query result r as:*

$$r = (v_{lca}, \left\{ m_1, m_2, ..., m_n \right\})$$

where m_i is a keyword match node *w.r.t. keyword k_i ($i \in [1, n]$), and v_{lca} is the lowest common ancestor of nodes m_1,..., m_n, i.e. $v_{lca} = lca(m_1, ..., m_n)$.* □

Defn. 3.4 is highly general in two aspects: (1) it is *compatible* with any existing LCA-based matching semantics adopted by search engines, because one necessary condition for a node v to be an SLCA (or ELCA) node of a query Q is: v must be a lowest common ancestor of a set of *keyword match nodes* m_i w.r.t. Q. (2)

Our query result format forms the skeleton for both Path Return (returning the paths in the XML tree from each LCA node to its *keyword match nodes*) [36, 56] and subtree Return (returning the subtree rooted at each LCA node) [31, 99]. This observation is important in explaining the *portability* feature of our solution to detect and resolve the MisMatch problem later in Section 3.4.4.

3.3 Detecting the Mismatch Problem

In this section, we would like to present how to detect the MisMatch problem.

First, the detector should infer user's possible search target(s) based on the query results. Since a keyword can match different types of nodes, user's search target may be various for a certain query. E.g., keyword "price" can match an owner's name (node *owner*:0.3) or the price of a product (node *price*:0.0.0.3.4) in Figure 3.1. But a certain query result r corresponds to a unique search target. Because each query keyword has a unique corresponding *keyword match node* in a given query result r. Therefore, we introduce a concept called *Target Node Type (TNT)* to denote the node type which a query result r intends to match.

To infer the TNT of a result r, we propose to use node types to simulate the semantics of each *keyword match node.*

Example 3.2. *For the query* $Q = \{$*'Vaio', 'W', 'red', 'price'*$\}$ *in Example 3.1, if the user is interacting with a salesman, the salesman will know that the user is finding a laptop because the salesman knows the meaning of each query keyword. Here for XML keyword search, one result is* $r = (0.0.0, \{0.0.0.4.1, 0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5\})$. *We use node types to simulate the semantics of each* keyword match node. *The node types of each distinct* keyword match node *are ('Vaio' and 'W' match the same node):*

0.0.0.4.1: {online_mall/electronics/shop/laptop/model}

0.0.0.3.2: {online_mall/electronics/shop/laptop/color}

0.0.0.4.5: {online_mall/electronics/shop/laptop/price}.

Then we can know that the user inputs three kinds of information: laptop model, laptop color and laptop price. The user's search intention, i.e. a laptop, corresponds to the node type "online_mall/electronics/shop/laptop", which is closely related to the above three node types. □

Following a similar philosophy of LCA, which finds the lowest/smallest nodes connecting all query keywords as the most relevant results, we define the lowest node type which connects to all the above node types at schema level as the Target Node Type, where the formal definition will be introduced later. It is the most relevant node type connecting to users' input information. E.g., in Example 3.2, {*online_mall/electronics/shop/laptop*} is the lowest node type connecting to laptop model, laptop color and laptop price at schema level even though no laptop can meet all the requirements at data level.

However, an XML document actually comes with some constraints on how many nodes of a type t_a can be subtree-contained by another node of type t_b. E.g., a laptop node (of node type *online_mall/electronics/shop/laptop*) can subtree-contains only one laptop model node (of node type *online_mall/electronics/shop/laptop/model*) while it can subtree-contain more than one laptop color node (of node type *online_mall/electronics/shop/laptop/color*). Similarly, a shop node (of node type *online_mall/electronics/shop*) can subtree-contains multiple laptop node and therefore a shop node also can subtree-contains multiple laptop model nodes.

Such constraints will affect the inferring of Target Node Type when we try to find the lowest node type connecting to users' input information at a schema level.

Example 3.3. *Suppose a user wants to find a shop selling two laptop models, both* *model Pavilion and Omni produced by Hewlett Packard, she may issue a query* $Q = \{$*'Hewlett', 'Packard', 'Pavilion', 'Omni'*$\}$ *in Figure 3.1, which contains two different laptop model names. If the user is interacting with a salesman, the salesman will know the user is not finding a particular laptop but something related to two different laptops, e.g. a shop selling those two laptops. Here in terms of XML keyword search, one query result is a subtree rooted at an eletronics node: r=(0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1}). The node types of each distinct* keyword match node *are ('Hewlett' and 'Packard' match the same node):*

0.0.0.3.0: {online_mall/electronics/shop/laptop/brand}

0.0.0.3.1: {online_mall/electronics/shop/laptop/model}

0.0.1.3.1: {online_mall/electronics/shop/laptop/model}.

The user's keywords are describing one laptop brand and two different laptop models, i.e., the user inputs two different laptop names matching two different laptop models. In such a case, assuming one laptop can only have one laptop model name in the data, the lowest node type connecting the above three node types is no longer online_mall/electronics/shop/laptop. Because there are two different laptop model nodes here while a laptop node can subtree-contain only one laptop model node. Instead, the lowest node type connecting the above three node types is online_mall/electronics/shop because a shop node can subtree-contain multiple laptop model nodes. Therefore, we can infer that the search intention is to find a shop selling two laptop models rather than finding a particular laptop model. □

The containment constraints among different types of nodes can be easily inferred from the schema of the XML document (if any). E.g. DTD is a commonly used XML schema language, where operator * (zero or more occurrences), + (one or more occurrences), ? (zero or one occurrence) are used to specify the number of occurrences of sub-elements or attributes under a particular type of node.

If the schema of the XML document is unavailable, we can still infer such constraints simply by scanning the XML document to summarize a DataGuide [29]. Let $t_1.maxContain(t_2)$ be the maximum number of nodes of type t_2 which can be subtree-contained by another node of type t_1. The range is $[0, +\infty]$. E.g., in Figure 3.1, if a laptop node can subtree-contain at most one laptop model node, then we have $laptop.maxContain(model) = 1$; if a shop node can subtree-contain multiple laptop nodes, we have $shop.maxContain(laptop) = +\infty$; besides, since shop node is the parent of laptop node and laptop node is the parent of model node, we can further infer $shop.maxContain(model) = +\infty$ by multiplying the above two values. Such a calculation can be done offline based on either the schema (if any) or the DataGuide (summarized by scanning the XML document).

Now we need to count the number of occurrences of each different node type for the *keyword match nodes*. Let $\mathbb{T} = \{t_1, t_2, ..., t_x\}$ be a set of different node types for the *keyword match nodes*. As some of the *keyword match nodes* could be of the same node type, let $count(t_i)$ be the number of *keyword match nodes* which are of type t_i. E.g. for the query result in Example 3.3, there are three distinct *keyword match nodes*: two of them are of type *online_mall/electronics/shop/laptop/model* and one of them is of type *online_mall/electronics/shop/laptop/brand*. In this case, $\mathbb{T} = \{brand, model\}$, $count(brand) = 1$ and $count(model) = 2$.

We are trying to find the most relevant node type connecting to users' input as the TNT, i.e., to find the lowest node type which can connect to all *keyword match nodes*' node types at schema level. Next we will define the Target Node Type of a result r formally.

First, TNT should be related to and connecting to each node type in $\mathbb{T}$, i.e. the TNT should be a common prefix of the node types in $\mathbb{T}$. Second, a node of the Target Node Type should be able to subtree-contain all occurrences of each node type in $\mathbb{T}$. Last, TNT should be as low as possible such that it can connect to each node type in $\mathbb{T}$ as closely as possible. So we define the extended TNT formally as follows:

Definition 3.5. ***Target Node Type (TNT) for a single query result.*** *Given a query $Q = \{k_1, k_2, ..., k_n\}$ and a query result $r = (v_{lca}, \{m_1, m_2, ..., m_n\})$ on an XML document D, let $\mathbb{T} = \{t_1, t_2, ..., t_x\}$*[1] *be the set of different node types for m_1 to m_n, the Target Node Type $TNT(r)$ for result r is defined as:*

$$TNT(r) = t$$

such that t satisfies the following 3 conditions

- Condition 1: $t \in commonPrefix(t_1, t_2, ..., t_x)$;

- Condition 2: $t.maxContain(t_i) \geq count(t_i), i \in [0, x]$;

- Condition 3: $\nexists t'$ such that t' is a descendent of t and t' also satisfies condition 1 and condition 2,

where $commonPrefix(t_1, t_2, ..., t_x)$ represents all possible common prefixes for a set of node types; $t.maxContain(t_i)$ represents the maximum number of t_i type nodes which can be subtree-contained by a t type node; $count(t_i)$ represents the number of different keyword match nodes *in m_1 to m_n which are of node type t_i.* □

TNT is the lowest node type which can connect to all *keyword match nodes*' node types at schema level. It is defined at the schema level by making use of node types, no matter whether what users search for exist in the XML document at data level or not. To calculate the TNT for a given result, we check the prefixes of each node type in $\mathbb{T}$ from the lowest one upwards, see whether it satisfies condition 2.

In the following two examples, we will infer the TNT according to the above definition for two sample queries, both of which are with the MisMatch problem,

[1] t_i may not necessarily be a one-to-one mapping to m_i. Because two *keyword match nodes*, say m_i and m_j, could be of the same node type.

i.e., what users search for is unavailable in the data. In Example 3.4, all *keyword match nodes* are of different node types; in Example 3.5, some *keyword match nodes* are of the same node type.

Example 3.4. *For the query $Q = \{$'Vaio','W','red','price'$\}$ in Example 3.1, one of the results is $r = (0.0.0, \{0.0.0.4.1,\ 0.0.0.4.1,\ 0.0.0.3.2,\ 0.0.0.4.5\})$, where the node types of each distinct* keyword match node *are ('Vaio' and 'W' match the same node):*

0.0.0.4.1: $\{online\_mall/electronics/shop/laptop/model\}$ (denoted as t_1)

0.0.0.3.2: $\{online\_mall/electronics/shop/laptop/color\}$ (denoted as t_2)

0.0.0.4.5: $\{online\_mall/electronics/shop/laptop/price\}$ (denoted as t_3).

The set of distinct node types $\mathbb{T} = \{t_1, t_2, t_3\}$, where $count(t_1) = 1$, $count(t_2) = 1$ and $count(t_3) = 1$.

Then we check the prefixes of all node types in $\mathbb{T}$. The lowest one is $t =$ "online_mall/electronics/shop/laptop". Suppose we have the following constraints (either inferred from the XML schema or by scanning the XML document): one laptop node can subtree-contain one model node, one price node and multiple color nodes. In other words, model and price are both single-value attributes for laptop, and color is a multi-valued attribute of laptop. Then it will satisfy: $t.maxContain(t_1) = 1 \geq count(t_1) = 1$, $t.maxContain(\ t_2) = +\infty \geq count(t_2)$ and $t.maxContain(t_3) = 1 \geq count(t_3)$.

Therefore, $TNT(r) = t =$ "online_mall/electronics/shop/laptop" even though no laptop can meet all the user's requirements at data level. It is the lowest node type which can connect to all keyword match nodes*' node types at schema level.* □

Example 3.5. *For the query $Q = \{$'Hewlett','Packard','Pavilion','Omni'$\}$ in Example 3.3, since there is no shop selling both of these models in Figure 3.1, the results being returned are not shops as expected by the user. One of the results*

is $r = (0.0, \{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1\})$*, where the node types of each distinct* keyword match node *are ('Hewlett' and 'Packard' match the same node):*
0.0.0.3.0: $\{online\_mall/electronics/shop/laptop/brand\}$ *(denoted as* t_1*)*
0.0.0.3.1: $\{online\_mall/electronics/shop/laptop/model\}$ *(denoted as* t_2*)*
0.0.1.3.1: $\{online\_mall/electronics/shop/laptop/model\}$ *(denoted as* t_2 *which is the same as the previous one).*

The set of distinct node types $\mathbb{T} = \{t_1, t_2\}$*, where* $count(t_1) = 1$ *and* $count(t_2) = 2$*.*

Then we check the prefixes of all node types in $\mathbb{T}$*. The lowest one is* $t =$ "*online_mall/electronics/shop/laptop*"*. Suppose we have the following constraints (either inferred from the XML schema or by scanning the XML document): one laptop node can only subtree-contain one brand node and one model node. In other words, brand and model are both single-value attributes for laptop. Then we will have:* $t.maxContain(t_1) = 1 \geq count(t_1) = 1$ *but* $t.maxContain(t_2) = 1 \ngeq count(t_2) = 2$*. As we can see,* t *is not the TNT as a laptop node cannot subtree-contain two model nodes.*

Then we will check another prefix $t' =$ "*online_mall/electronics/shop*"*, which is just above* t*. Suppose we have the following constraints in the XML document (either inferred from the XML schema or by scanning the XML document): one shop node can subtree-contain multiple brand nodes while it can subtree-contain multiple model nodes, then we have:* $t'.maxContain(t_1) = +\infty \geq count(t_1) = 1$ *and* $t.maxContain(t_2) = +\infty \geq count(t_2) = 2$*.*

Therefore, Target Node Type of result r *is* $TNT(r) = t' =$ "*online_mall/electronics/shop*"*. It is the lowest node type which can connect to all* keyword match nodes*' node types at schema level.* □

Our solution assumes there is no outer semantics provided. Because usually

XML data exists without such information, so that we use node types to simulate semantics, where two nodes of the same type will be with the same semantics. If we do have outer semantics, like thesaurus, ontology, etc., we can further improve our approach such that we can even tell that node types "/laptop/color" and "/notebook/color" are with the same semantics while node types "/owner/name" and "/product/name" are with different semantics. This will be one of our future work.

3.3.1 Detecting The MisMatch Problem based on Target Node Type

With the Target Node Type of a query result r being inferred, the detector should figure out whether there is a mismatch between the TNT (see Defn. 3.5) of r and the actual root of r, namely v_{lca}.

Definition 3.6. *Given a query* $Q = \{k_1, k_2, ..., k_n\}$ *and a query result* $r = (v_{lca}, \{m_1, m_2, ..., m_n\})$ *on the XML data D, if* v_{lca} *is not of the same node type as* $TNT(r)$*, the query result r* ***misses the target****.* □

For result r in Example 3.4, $v_{lca}.type$ = *shop*$\neq$*laptop* = $TNT(r)$, so we say r misses the target. Now, we can formally define the MisMatch problem.

Definition 3.7. ***MisMatch Problem.*** *Given a query Q and its results* $\mathbb{R}$ *retrieved from the keyword search engine, Q has the MisMatch problem if all* $r \in \mathbb{R}$ *misses the target.* □

Here we choose to take a conservative approach: we only judge a query to have the MisMatch problem when there is a mismatch for all possible search intentions. Such a conclusion holds for all users with different intentions. E.g., for the result r in Example 3.4, we inferred that it misses the target. In a similar way, we will also calculate a TNT for each of the other results (if any). We will claim that the query has the MisMatch problem only if all the results miss their corresponding target.

Moreover, users usually investigate the retrieved results starting from the *top-ranked* ones. Therefore, without loss of generality, we can also easily extend Defn. 3.7 by considering the top-K retrieved results of Q.

Time Complexity of the detector is O($|\mathbb{R}|$), which is very efficient. As discussed in Sec. 5.2.2 later, we store the type information of each node when building the keyword inverted list. Thereby for each $r \in \mathbb{R}$, $TNT(r)$ can be computed in O(1) time assuming the number of keywords in a query and the depth of the XML tree are bounded by some constants.

3.4 Finding Explanations and Suggested Queries

As discussed in Section 3.3, the main feature of the MisMatch problem is: there does not exist a single *TNT node* that *subtree-contains* all query keywords. So the query keywords have to scatter in more than one TNT node and then lead to a mismatch result. As a result, the root of the returned subtree is always an ancestor of the TNT nodes which are expected by the user. Given a user query Q=$\{k_1, k_2, ..., k_n\}$ and a mismatch query result r=(v_{lca},$\{$ $m_1, m_2, ..., m_n\}$), where m_i is a *keyword match node* for k_i, the basic idea to find the explanations and some promising suggested queries can be illustrated in three steps.

Step 1: Since each *keyword match node* m_i in r may contain several keywords $\mathbb{K}$ in Q, we first propose a *tf*idf*-inspired heuristic called *distinguishability* to score the importance of such $\mathbb{K}$.

Step 2: We then try to find the approximate query results, i.e. $r' = (v'_{lca}$,$\{m'_1, m'_2, ..., m'_n\})$, which are some subtrees containing the 'important' keywords (derived by Step 1).

An ideal approximate result r' should satisfy the following properties: (a) the node type of r' should be the same as $TNT(r)$; (b) for each *keyword match node* m_i in original result r, there always exists a node m'_i that has the same node type as m_i ($i \in [1, n]$). By such properties, it can ensure at least the structure of r' and r are consistent with each other.

Step 3: Then, we can pinpoint which keyword(s) in the user's query lead to the mismatch results, i.e. the query keywords not contained by the approximate results. This is the explanation part. We can further infer the suggested queries by replacing those keywords with the keywords associated with the aforementioned m'_i (in approximate result) in step 2.

Step 1 is illustrated in Section 3.4.1, and the last two steps are described in Section 3.4.2. Lastly, we complement our suggester by discussing how to rank the suggested queries in Section 3.4.3.

3.4.1 Distinguishability

In this section, we will present a concept to measure the importance of query keywords, namely *distinguishability*. We find that the importance of query keywords is closely related to what type of nodes they match. E.g., in Figure 3.1, keyword 'blue' can match either a shop name *name*:0.0.1.0 or a laptop color *color*:0.0.0.4.3. When it matches a shop name, most likely it is important since few shop names contain the keyword 'blue'; when it matches a laptop color, it may be less important since many color nodes contain the keyword 'blue'. Therefore, we propose the concept of *distinguishability*.

Distinguishability $D(\mathbb{K}, t)$ represents the importance of the query keywords $\mathbb{K}$ when $\mathbb{K}$ matches a node of type t, which also means this node of type t *subtree-contains* each keyword in $\mathbb{K}$. Large $D(\mathbb{K}, t)$ means $\mathbb{K}$ is important with respect to

t.

Recall Step 1 in Section 3.4, $\mathbb{K}$ actually represents the query keywords derived from the *keyword match node*(s). To quantify $D(\mathbb{K}, t)$, we propose a scoring measure inspired by Term Frequency * Inverse Document Frequency (*tf***idf*) [83], which is widely used in information retrieval.

For *tf*, we can simply count the keyword frequency in an XML node. In this work we focus on data-centric XML documents, where each XML node does not contain long text and in most cases keyword frequency is 1. The same problem is also pointed out by [33], so we follow [33] and do not consider *tf* in the formula.

For *idf*, it tells that the keywords contained by fewer documents are more important. Similar to *idf*, we have Intuition 1 in the context of XML. Let f_t be the number of nodes of type t, and $f_t^{\mathbb{K}}$ be the number of nodes which are of node type t and subtree-contain each keyword in $\mathbb{K}$.

Intuition 1. *idf*$(\mathbb{K}, t)$. *If few nodes of type t contain keywords $\mathbb{K}$, $\mathbb{K}$ should be important with respect to the node type t. Formally, the smaller the $f_t^{\mathbb{K}}$ is as compared to f_t, the larger the idf$(\mathbb{K}, t)$ should be.*

As there are many variants of *idf* to follow Intuition 1, we define *idf*$(\mathbb{K}, t) = 1 - \frac{f_t^{\mathbb{K}}}{f_t}$. In this way, *idf*$(\mathbb{K}, t)$ is normalized in [0,1).

The *tf***idf* works by assuming there is only one type of (flat) document, but in the context of XML data there is more than one type of node. The type of the node alone may also contribute to the importance of the keywords that match the node. Let us look at a motivating example first.

Example 3.6. *Consider a keyword 'price' in Figure 3.1. It can match both an* owner *node and all* price *nodes. When 'price' matches a price node, it may not be important as there are many price nodes and all of them contain 'price'. Accordingly, idf({'price'},price)=0 because $f_t^{\mathbb{K}}$=f_t. When it matches the owner node,* *it should be important as there is one and only one owner across the whole XML data. But since $f_t^{\mathbb{K}}$=f_t=1, idf({'price'},owner)=0 as well. As we can see, simply by* tf*idf, *we cannot distinguish these two cases (*idf *is 0 for both cases). Because the idea of* tf*idf *assumes there is only one type of node while we have nodes of different types and we need to consider the weight of different node types.* □

So we have Intuition 2 to cater for the *node type weight (ntw).*

Intuition 2. *ntw(t). The weight of a node type t is inversely proportional to f_t within the XML data.*

Therefore, We define $ntw(t) = \frac{1}{f_t}$. Finally, we can define $D(\mathbb{K}, t)$ to capture the concept of distinguishability as:

$$D(\mathbb{K}, t) = idf(\mathbb{K}, t) + ntw(t) = 1 - \frac{f_t^{\mathbb{K}}}{f_t} + \frac{1}{f_t} \quad (1 \leq f_t^{\mathbb{K}} \leq f_t) \tag{3.1}$$

It is easy to verify that the range of distinguishability is (0,1].

3.4.2 Two-phase Solution

In order to find the explanation and suggested queries, we first need to find some 'important' query keywords (in terms of distinguishability) from the result r of the original query. So first of all, we need to set a threshold τ[2], say τ=90%. Those keywords whose distinguishability is higher than τ are considered as 'important' and must be kept. Besides, we find that those 'important' keywords $\mathbb{K}$ are indeed derived from the *keyword match node(s)* of r, thereby we may need to consider two independent cases at the same time:
(1) $\mathbb{K}$ is derived from a single *keyword match node* of r;
(2) $\mathbb{K}$ is derived from multiple *keyword match nodes* of r, i.e., combing the keywords

[2]The choice of an appropriate τ will be discussed in the experimental study.

from multiple *keyword match nodes* could achieve high distinguishability.

Then the remaining task is to find the approximate results, each containing the important keywords $\mathbb{K}$, from which suggested queries are inferred.

Phase 1: based on single *keyword match node*

In Phase 1, we derive important keywords from a single *keyword match node* and find the approximate results as follows:
Given a user query Q and a mismatch query result r=(v_{lca},$\{m_1, m_2, ..., m_n\}$), each *keyword match node* m_i contains some keyword(s) $\mathbb{K}_i$ in Q. For each distinct m_i, we calculate the distinguishability $D(\mathbb{K}_i, m_i.type)$. If it is larger than the threshold, then we try to find a TNT node containing m_i as an approximate result. Let the path from v_{lca} to m_i be $(v_{lca}/p_1/p_2/.../p_j/m_i)$, where p_1,p_2,....,p_j are the nodes between v_{lca} and m_i. Then we proceed to traverse each node v'_{lca} from p_1 down to m_i (i.e. $v'_{lca} \in \{p_1, p_2, ..., p_j, m_i\}$), and verify whether the subtree rooted at v'_{lca} can form an approximate query result r'=(v'_{lca},$\{ m'_1, m'_2, ..., m'_n\}$) w.r.t. r.

Definition 3.8. ***Approximate Result.*** *Given a query result* r=(v_{lca},$\{m_1, m_2, ..., m_n\}$) *for a query* Q, $r' = (v'_{lca}, \{m'_1, m'_2, ..., m'_n\})$ *is an approximate result if* r' *have the following two properties:*

- ***P1****:* $v'_{lca}.type = TNT(r)$
- ***P2****:* $m'_i.type = m_i.type$*, for* $i \in [1, n]$.

P1 is to ensure v'_{lca} of r' should have the same node type as the TNT that result r intends to match (but fail to do so). P2 is to ensure a consistency of the internal structure of r and r' in the way that, each node type appearing in the *keyword match node* of r must also appear in those of r'. Intuitively speaking, the node type of each *keyword match node* implicitly reflects the constraint that user intends to specify for the desired query result. Therefore we need to keep all of them in the approximate result. As an analogy, it is an implicit representation of predicates specified in a structured query, whereas the difference is that in a keyword query you have no way to specify constraint on the structural relationship among keywords.

Since there can be many approximate results, a promising approximate result should be with minimal changes to the original result while keeping those m_i which contains the important query keywords. That is why we need to calculate the distinguishability for each m_i. If m_i contains important query keywords (in terms of distinguishability), we will find the approximate result on the path from v_{lca} to m_i, such that we can make sure m_i will be kept within the approximate result. We want m_i to be the same node as m'_i if possible, as we want to make minimal changes. In other words, only if m_i is not in the subtree rooted at v'_{lca}, it will be replaced by a distinct node m'_i.

In this section we focus on how to find approximate results and suggested queries first. How to rank them will be discussed in Section 3.4.3.

Suggested Query and Sample Query Result. After the approximate query results are found, the explanation and suggested query can be inferred easily by the following way: 1) for each different *keyword match node* m_i which is not the same node as m'_i, the query keyword(s) in m_i is the reason for the mismatch results; 2) the suggested query can be generated by replacing the keywords in m_i with the associated value of m'_i, highlighted by an underline. Besides, the approximate query result will be used as a sample query result for the corresponding suggested query.

Next, we will use two running examples to illustrate how we find the suggested queries and sample query result. The following two running examples correspond to the queries in Example 3.4 and Example 3.5 respectively.

Example 3.7. *For query Q ={'Vaio','W','red','price'} in Example 3.4, one query result is r=(0.0.0, {0.0.0.4.1, 0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5}), where there are only three distinct* keyword match nodes*. So we calculate three distinguishability values w.r.t. the query keywords in the three* keyword match nodes*:* $D(\{\text{'}Vaio\text{'}, \text{'}W\text{'}\}, model) = 100\%$, $D(\{\text{'}red\text{'}\}, color) = 68.2\%$, $D(\{\text{'}price\text{'}\}, price) = 0.5\%$.

Since D({'Vaio', 'W'},model)>τ=90%, it is important and must be kept. Then we check the path from shop:0.0.0 (v_{lca}) to model:0.0.0.4.1 (m_i), which is (shop:0.0.0/ laptop :0.0.0.4/model:0.0.0.4.1). In Example 3.4 we know $TNT(r)$ = *laptop, so we check the subtree rooted at laptop:0.0.0.4. For each* keyword match node m_i *in the original result r, within the subtree rooted at* 0.0.0.4, *we can always find a node* m'_i *with the same type. E.g. for the* keyword match node 0.0.0.3.2 *in r, we can find node* 0.0.0.4.2 *with the same node type:* (0.0.0.4.2)*.type* = *color* = (0.0.0.3.2)*.type. As a result, the set of* m' *nodes is: {0.0.0.4.1, 0.0.0.4.1, 0.0.0.4.2, 0.0.0.4.5}. Therefore, an approximate query result* r' *is constructed:*

$$r' = (0.0.0.4, \{0.0.0.4.1, 0.0.0.4.1, 0.0.0.4.2, 0.0.0.4.5\})$$

Compared to r, keyword match node *color:0.0.0.3.2 is changed to color:0.0.0.4.2. Node color:0.0.0.3.2 contains keyword 'red' and the content of color:0.0.0.4.2 is 'white'. So the keyword 'red' in user's query leads to the mismach results. The suggested query can also be inferred as {'Vaio', 'W',<u>'white'</u>, 'price'} by changing 'red' to 'white', and* r' *is its corresponding sample result. Similarly, we can also find suggested queries by changing 'red' to 'blue' or 'pink'.* □

Example 3.8. *For query Q={'Hewlett','Packard','Pavilion','Omni'} in Example*

3.5, where the user wants to search for a shop selling both the laptop model 'Pavilion' and 'Omni'. However, there is no such shop which sells both of the laptop models. One query result is a subtree rooted at an electronics node: r=(0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1}), where there are only three distinct keyword match nodes*. So we calculate three distinguishability values w.r.t. the query keywords in the three* keyword match nodes*:* $D(\{\text{'Hewlett'}, \text{'Packard'}\}, brand) = 75.5\%$, $D(\{\text{'Pavilion'}\}, model) = 100\%$, $D(\{\text{'Omni'}\}, model) = 100\%$.

Since both D({'Pavilion'},model) and D({'Omni'},model) are larger than the threshold τ *(90%), both of them are important. So we will check the following two paths for finding approximate results: path from electronics:0.0 (*v_{lca}*) to model:0.0.0.3.1 (*m_i*); path from electronics:0.0 (*v_{lca}*) to model:0.0.1.3.1 (*m_i*). Here we will take the first path as an example to illustrate how to check the path, which is (electronics:0.0/shop:0.0.0/laptop:0.0.0.3/model:0.0.0.3.1). In Example 3.5 we know* $TNT(r)$ = *shop, so we check the subtree rooted at shop:0.0.0. For each* keyword match node m_i *in the original result r, within the subtree rooted at* 0.0.0*, we can always find a node* m'_i *with the same type. E.g. for the* keyword match node 0.0.1.3.1 *in r, we can find node* 0.0.0.4.1 *with the same node type:* $(0.0.0.4.1).type = model = (0.0.1.3.1).type$*. Thus the set of* m' *nodes is: {0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.0.4.1}. Therefore, an approximate query result* r' *is constructed:*

$$r' = (0.0.0, \{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.0.4.1\})$$

Compared to r, keyword match node *model:0.0.1.3.1 is changed to model:0.0.0.4.1. Node model:0.0.1.3.1 contains keyword 'Omni' and the content of model:0.0.0.4.1 is 'Vaio W'. So the keyword 'Omni' in user's query leads to the mismach results. The suggested query can also be inferred as {'Hewlett','Packard','Pavilion',$\underline{\text{'Vaio','W'}}$}*

by changing 'Omni' to 'Vaio W'. Because shops selling these two models are available. r' is the corresponding sample result. □

Note that, if we set the threshold τ to a very low value, say zero, which means all keywords are with acceptably high distinguishability, then we will examine all the TNT nodes containing at least one of the *keyword match nodes*. This can cover all possibilities but of course more time will be consumed. We will show in the experiment (Section 3.7) that most likely it is not necessary.

Phase 2: based on multiple *keyword match nodes*

When the important keywords are derived from multiple *keyword match nodes* m_i, i.e., combing the keywords from multiple *keyword match nodes* could achieve high distinguishability, we need to compute the lowest common ancestor of these m_i, denoted by v, in order to calculate distinguishability. This is the only difference as compared to Phase 1. Let $\mathbb{K}$ be the query keywords subtree-contained by v. Then the rest job is similar to *Phase 1*, where we calculate $D(\mathbb{K}, v.type)$ and if it is acceptably high, we will check the path from v_{lca} to v to find the approximate result(s). Please refer to Algorithm 1 for details on our two-phase solution.

However, it requires 2^n times of calculation to get all possible lowest common ancestors of any subset of the n *keyword match nodes*. But we find Property 1 to help fulfill it in linear time.

Property 1. *Let $M = \{m_1, m_2, ..., m_n\}$ be the set of distinct* keyword match nodes *for a query result ($m_i \neq m_j$ if $i \neq j$), sorted by their Dewey labels. Then all possible lowest common ancestors (LCA) for any subset S of M, where $|S| \geq 2$, are in the set*

$$\{lca(m_1, m_2), ..., lca(m_i, m_{i+1}), ..., lca(m_{n-1}, m_n)\}.$$

Proof. (By Induction) Step 1: For $n = 2$, this property obviously holds. Step 2: We assume that for n=$k-1$, all LCAs of any subset of M_{k-1}=$\{m_1,m_2,...,m_{k-1}\}$ are in $\{lca(m_1, m_2),\ lca(m_2, m_3),...,lca(m_{k-2},m_{k-1})\}$. We will show that for a set of k nodes M_k=$\{m_1,m_2,...,m_{k-1},m_k\}$, all possible LCAs are in the set L= $\{lca(m_1,m_2),lca(m_2,m_3),...,\ lca(m_{k-1},m_k)\}$. Suppose $Dewey(\ m_{k-1})$=$a_1.a_2...a_j.a_{j+1}...$ and $Dewey(m_k)$=$a_1.a_2\ ...a_j.a'_{j+1}...$, let $m' = lca(m_{k-1}, m_k)$, then $Dewey(m')$=$a_1.a_2...a_j$. As nodes are sorted by Dewey label, there does not exist another node m_i in M_k such that $lca(m_i, m_k)$ is a descendant of m'; otherwise, $Dewey(m_i)$ should be of the form $a_1.a_2...a_j.a'_{j+1}...$ and m_i should appear between m_{k-1} and m_k. So for any subset containing m_k, namely $\{m'_1, m'_2, ..., m_k\}$, their LCA must not be a descendant of m'. If the LCA node equals to m', it is in L; if the LCA node is an ancestor of m', we can get the following because finding LCA is equal to finding the longest common prefix of Dewey labels of a set of nodes: $lca(\{m'_1, m'_2, ..., m_k\})$=$lca(\{m'_1, m'_2, ..., m'\})$= $lca(\{m'_1, m'_2,\ ..., m_{k-1}\})$, which is also in L according to the assumption. Besides, for subsets not containing m_k, their LCAs will also be in L according to the assumption. □

With Property 1, for a query that has MisMatch problem, we only need to conduct at most $n-1$ times of LCA computations to find all possible approximate results. We will use Example 3.9 to illustrate how we infer suggested queries for Phase 2.

Example 3.9. *Suppose a user wants to find a laptop which is of brand Hewlett Packard with purple color running windows vista. She may try to issue a query Q=\{'Hewlett', 'Packard', 'purple', 'windows', 'vista'\} in Figure 3.1. One of the query results is r=(0.0.0,\{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.3, 0.0.0.4.6, 0.0.0.4.6\}). By Defn. 3.6 we know that* $TNT(r)$ = *laptop but the result is a subtree rooted at a shop node. Therefore it misses the target.*

Suppose Hewlett Packard only has two models with purple color. The keywords matching brand:0.0.0.3.0, color:0.0.0.3.3 and OS:0.0.0.4.6 are not of high distinguishability (90%) in Phase 1: D({'Hewlett','Packard'},brand)=75.5%; D({'windows', 'vista'}, OS)=42.5%; D({'purple'},color)= 80.7%. Now in phase 2, by Property 1, all possible lowest common ancestors of the keyword match nodes *are 0.0.0.3 and 0.0.0. Take 0.0.0.3 as an example, we will find that the keywords subtree-contained by laptop:0.0.0.3 have high distinguishability:*

D({'Hewlett', 'Packard', 'purple'},laptop)=98.4%

Note that the above three keywords are actually from two keyword match nodes, *i.e., brand:0.0.0.3.0 and color:0.0.0.3.3.*

Then similar to Phase 1, we will try to find an approximate query result along the path from r's v_{lca} *to laptop:0.0.0.3, i.e. (shop:0.0.0/laptop:0.0.0.3). Finally we find the approximate result rooted at laptop:0.0.0.3 and get a suggested query by changing keywords 'windows vista' to 'windows 7'.* □

3.4.3 Ranking the Suggested Queries

After all suggested queries are generated, we build a preliminary ranking model to judge the quality *score* of a suggested query with the following factors:

1. Number of keywords (in original query) that need to be changed, denoted as *cn*. The larger *cn* is, the lower *score* should be.

2. Distance between the approximate query result root v'_{lca} and original query result root v_{lca}, denoted as *dt* (*dt* is equal to the length difference of their Dewey labels). The larger *dt* is, the higher *score* should be. Because a more compact subtree is preferred.

3. Sum of distinguishability of the keywords that need to be changed, denoted

as $\sum D$. The larger $\sum D$ is, the lower *score* should be. Because we prefer not to replace keywords those are with high distinguishability.

To sum up the above ranking factors, we calculate the ranking score by taking a product of them:

$$score = \frac{1}{e^{cn}} \times (1 - \frac{1}{e^{dt}}) \times \frac{1}{e^{\sum D}} \tag{3.2}$$

3.4.4 Summary of Features of Our Approach

To summarize, our MisMatch detector and suggester have the following features. First, it is *portable*: by capturing the LCA commonality among existing search semantics in defining the format of query result (Defn. 3.4), our approach can work with any LCA-based matching semantics (recall Section 3.2.2); since our approach is a post-processing of the query evaluation, it is orthogonal to the result retrieval method adopted. Second, it is *result-driven*: our approach accepts the results of the original query as input, and recall Section 3.4.2 the suggester finds the important keywords (to be kept in suggested queries) from each result, to guarantee the empirical quality of suggestions. Third, it is *lightweight*: it occupies a small proportion of the whole query evaluation time, as discussed in Section 3.5 later.

3.5 Efficient Approximate Results Detection

Recall Defn. 3.8, to check whether a TNT node is an approximate query result, the core operation is to verify whether the two properties **P1** and **P2** hold. Checking P1 is trivial, so we aim to achieve an efficient check of P2 by designing a novel node labeling scheme and the corresponding logical operations.

3.5.1 Node Labeling

Since our suggester needs to frequently access the type of a node along the way to finding suggested queries, we first collect all node types in XML data. By simply scanning the XML file, we can get a **schema tree** which contains all node types using DataGuide [29]. E.g., for the XML data in Figure 3.2, we can construct a schema tree as shown in Figure 3.3(a), where each node in the schema tree represents a unique node type. Note that each node in Figure 3.3 should be a node type represented as a path (according to Defn. 5.2), but for simplicity we use a tag name instead because there is no ambiguity.

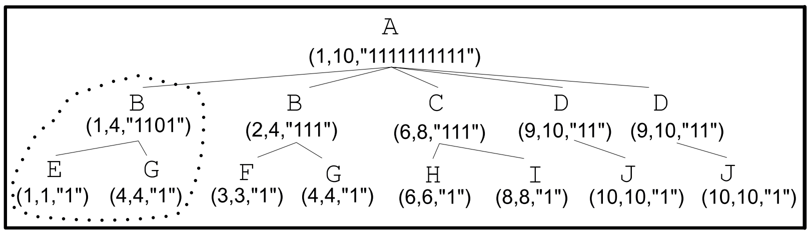


Figure 3.2: An XML Tree with Nodes Labeled by exLabels

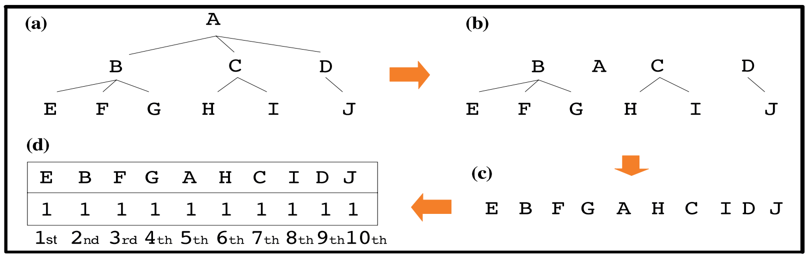


Figure 3.3: Schema Tree Flattening and Virtual Bitmap Construction

Then, we use a *bitmap* to denote all node types in the schema tree, where each bit in the bitmap corresponds to a specific type. We purposefully decide which bit corresponds to which type as follows:

- Flatten the schema tree level by level in a top-down manner. Suppose a node n has k children, then n will be inserted into a place between its $\lfloor \frac{k}{2} \rfloor$th and

$(\lfloor \frac{k}{2} \rfloor + 1)$th children. As a result, n will maintain its position between its neighbors and neighbors' children. Figure 3.3(a), (b) and (c) show such a process of flattening.

- Construct a virtual *bitmap* as shown in Figure 3.3(d). Each distinct node type has a unique **position number** in *bitmap*. E.g., F's position number is 3.

Such a bit-to-type mapping has a nice property: *the bits of all node types that appear in a specific subtree in XML will stay together.* As we can see later, this property helps ensure the label size as compact as possible.

For a node n in the XML tree, the subtree rooted at n may contain different types of nodes. To indicate which node types appear in its subtree ST_n, we assign n a label (a, b, bm), called **exLabel**. Here, a is the smallest position number (in the *bitmap*) of the node type appearing within ST_n; similarly, b is the largest position number of the node type appearing within ST_n. bm is a sub-sequence of the *bitmap* (of the schema tree) from position a to b, indicating which type of nodes can be found in the subtree rooted at n. In particular,

- $bm[i]$=1, if the node type at position $a+i-1$ in *bitmap* appears in the subtree rooted at n;
- $bm[i]$=0, otherwise. (i∈[1,b-a+1])

Example 3.10. *In Figure 3.2, for the subtree rooted at node B circled by the dotted line, it contains nodes of types E, B and G. According to the bitmap in Figure 3.3(d), the position number is 1 for E, 2 for B and 4 for G. Among the four node types ranging from position 1 to 4, bm of node B indicates which of those four node types appear in B's subtree ST_B. As a result, bm=1101 as the 3rd node type F does not appear in ST_B, and B's exLabel = (1,4,'1101'). Note that the exLabel of*

B is compact *because the bits representing E, B and G are staying together, which is the benefit from the aforementioned bit-to-type mapping.* □

3.5.2 Logical Operation

Similar to node labeling, for a query result $r = (v_{lca}, \{m_1,\ m_2, ..., m_n\})$, we can intentionally construct an exLabel to represent its node type information even though it is not a node at all. Let a' (b') be the smallest (largest) position number of the node type for m_i, and the label for the query result is denoted as (a', b', bm').

Having a query result label (a', b', bm') and a subtree root label (a, b, bm), we can verify property **P2** by examining the following containment relation: $(a', b', bm') \subseteq (a, b, bm)$.

This relationship holds only if $a \leq a' \leq b' \leq b$ and all bits that appear in bm' also appear in bm. This can be efficiently done by a logical *AND* operation on bm' and bm.

Example 3.11. *In Figure 3.2, suppose a query result $r = (v_{lca}, \{m_1,\ m_2\})$, where m_1.type = B, m_2.type = G. Then the exLabel for r is (2,4,'101'). If we want to check whether an approximate query result exists in the subtree rooted at the left node B in Figure 3.2, whose exLabel is (1,4,'1101'), then we know the approximate query result exists because (2,4,'101') ⊆ (1,4,'1101').* □

3.6 Algorithms

3.6.1 Data Processing and Index Construction

In the phase of XML document parsing, we collect all distinct node types and generate a bitmap code for each node type as discussed in Section 3.5.1. For each node n visited, we assign a Dewey label $deweyID$ [93] to n; get the node type t_n of n; construct an $exLabel$ for n. To speed up the query processing and refinement, three indexes are built.

The *first* index is called *replacement table*, which is a B+ tree storing each node with (t,$deweyID$) as its key. Such an index has the following property: by scanning rightwards of the position (t,$deweyID$), we can find all the nodes of type t under the subtree rooted at $deweyID$. Recall in Section 3.4.2, after we find an approximate query result r', we need to materialize the replacement nodes within r' in order to infer the suggested query. Since we know the type t of each replacement node and the $deweyID$ of the root node of r', with replacement table, we can easily materialize all such nodes by calling getReplacement (t,$deweyID$). The *second* index is to maintain the exLabel and type info for each node.

To speed up the computation of distinguishability, particularly for parameter $f_t^{\mathbb{K}}$ in Formula 3.1, the *third* index called *inverted index* is built: For each combination of a distinct node type t and a distinct keyword k (in XML data), we build an inverted list containing all nodes of type t where each node *subtree-contains* keyword k. Those inverted lists are grouped by node type t. As a result, $f_t^{\mathbb{K}}$ can be computed by simply computing the intersection of the inverted lists for each keyword in $\mathbb{K}$ under node type t [52]. Operation getDist($deweyID$,$\mathbb{K}$) returns the distinguishability of a set of keywords $\mathbb{K}$ w.r.t. the type of the node with $deweyID$.

Here, we analyze the space complexity of the third index - *inverted index*. To simplify our analysis, we assume the average degree of each node is d, average level of the XML is L and average keyword frequency is F. For a traditional inverted index, every keyword in the data will be counted once for representing the corresponding node in inverted list. In our inverted index, every keyword will be counted at most L times because every keyword appearing in a node is also a keyword for all its ancestor nodes. Let's say root, it is the ancestor of all nodes below, so it subtree-contains all keywords. If keyword frequency is F, every keyword will be counted $1/F$ times for the root level of the tree because the same keywords will be counted as one for one node. And every keyword will be counted $1/(F/d)$ times for level 2 because keywords frequency for subtree rooted at level 2 will become F/d. Similarly, we can know a keyword will be counted at most the following times:

$$\underbrace{\frac{d^0}{F} + \frac{d^1}{F} + ... + \frac{d^{\llcorner log_d^F \lrcorner}}{F} + 1 + 1 + ... + 1}_{\text{totally L terms}}$$

Note that when the keyword frequency becomes 1 at a certain level, all frequencies of levels below it will be 1, which is expressed by the term $\frac{d^{\llcorner log_d^F \lrcorner}}{F}$.

Let the size of a traditional inverted index be $size$, average level be L. Then for the worst case, where average keyword frequency is 1, the space complexity of our index is $O(L * size)$.

3.6.2 Solving the MisMatch problem

The main procedure is presented in Algorithm 1, where the input is the query Q and its retrieved results $\mathbb{R}$. First it checks each result of Q (line 2) and calculates its TNT (line 3). Once one of the results does not miss the target, which means what the user wants is in the retrieved results, it will terminate the process (line 4). Otherwise, it constructs an exLabel for the query result (line 6) as discussed in Section 3.5.2.

For *Phase 1* (in Section 3.4.2), it checks each *keyword match node nd* of the query result (line 8). If the distinguishability is larger than the threshold τ (line 9), the TNT node on the path from the v_{lca} to this node will be checked in order to find an approximate query result (line 10). Whether an approximate query result

```
Algorithm 1: MisMatchResolver($Q$,$\mathbb{R}$)
   input  : user query $Q$=$keywords[m]$, $Q$'s results $\mathbb{R}$
   output : null if no MisMatch problem; $suggestedQueries$ + one sample result $n$
            for each $Q' \in suggestedQueries$ otherwise
 1 suggestedQueries $\leftarrow \emptyset$;
 2 foreach $r \in \mathbb{R}$ do
 3     if $r.v_{lca}.type = getTNT(r)$ then
 4         return $null$;
 5 foreach $r \in \mathbb{R}$ do
 6     $rExlabel$ = constructExlabel($r$);
 7     {Phase 1}
 8     foreach $nd \in r.matchnodes$ do
 9         if $getDist(nd.dewey, nd.keywords) > \tau$ then
10             foreach $n \in$ nodes on the path from $r.v_{lca}$ to $nd$ AND
               $n.type = getTNT(r)$ do
11                 if contain(getExLabel(n.dewey), rExlabel) then
12                     QuerySuggester($n, r, suggestedQueries$);
13     {Phase 2}
14     sort($r.matchnodes$);
15     for $i$ = 1 to ($r.matchnodes.length$-1) do
16         Let $v$ = getLCA($r.matchnodes[i]$, $r.matchnodes[i+1]$);
17         $kwinside$ = getQueryKwsInside($r$, $v$);
18         if $getDist(v, kwinside) > \tau$ then
19             foreach $n \in$ nodes on the path from $r.v_{lca}$ to $v$ AND
               $n.type = getTNT(r)$ do
20                 if contain(getExLabel(n.dewey), rExlabel) then
21                     QuerySuggester($n, r, suggestedQueries$);
22 return $suggestedQueries.sort()$;
```

exists can be easily checked by examining the containment relationship between the exLabels (line 11), as described in Section 3.5.2. If an approximate query result exists, the explanations and suggested queries will be inferred by calling $QuerySuggester()$ (line 12).

For *Phase 2* (in Section 3.4.2), we sort the *keyword match nodes* (line 14) and check the LCA node of every two adjacent *keyword match nodes* (line 16) according to Property 1. Then we need to find which query keywords are *subtree-contained* by this LCA node (line 17). Afterwards, we follow the same steps (line 18-21) as Phase 1. Finally, it returns the suggested queries (line 22) sorted by the ranking formula in Section 3.4.3, attached with one sample result for each suggested query.

Algorithm 2: QuerySuggester(v'_{lca}, r, $sugQueries$)

```
  input  : the approximate result root $v'_{lca}$, the query result being changed $r$ and
           the suggested queries $sugQueries$
  output : new suggested queries + one sample result $v'_{lca}$
1 $i = 0$;
2 foreach $nd \in r.matchnodes$ do
3     if $nd$ is not a descendant of $v'_{lca}$ then
4         replace[i++] = getReplacement($nd.type$, $v'_{lca}.dewey$);
5 foreach $n_1 \in replace[1],...,n_i \in replace[i]$ do
6     $sugQueries = sugQueries \bigcup (r.matchnodes[1] \to n_1,...,$
      $r.matchnodes[i] \to n_i)$;
```

Given the approximate result root and the original query result, Algorithm 2 presents how to infer the suggested queries. *Keyword match nodes* which are not in the subtree rooted at v'_{lca} will be replaced by nodes in v'_{lca} that have the same node type according to property P2 in Defn. 3.8 (line 2-4). For a *keyword match node* that needs to be changed, there may be more than one replacement node to replace it. Such nodes can be retrieved from index by calling the function *getReplacement*() (line 4). Note that there might be more than one *keyword match node* needed to be changed, so suggested queries will be inferred by considering all possible cases (line 6).

Algorithm 3: contain(elx, ely)

```
  input  : exLabel $elx$ and exLabel $ely$
  output : a boolean indicating whether $elx$ contains $ely$
1 if ($elx.a \leqslant ely.a$ and $ely.b \leqslant ela.b$)==false then
2     return $false$;
3 $bmTemp$ = subset of $elx.bm$ from position $ely.a$ to $ely.b$;
4 if ($bmTemp$ & $ely.bm$)==$ely.bm$ then
5     return $true$;
6 return $false$;
```

Algorithm 3 presents the function, *contain*(), to examine the containment relationship between two exLabels, i.e., the first contains the second. As discussed

Table 3.1: 10 of the Sample Queries on IMDB

IMDB:90MB			
#	Query	suggested queries	best-3 suggested queries (Format: explanation → suggested options)
Q1	Gladiator Spanish	5	(language): Spanish → English / Japanese / French
Q2	Spielberg DiCaprio Action movie	6	(genres): Action → Biography / Crime / Drama
Q3	Neo hacker phonebooth	3061	(keyword): phonebooth → computer / software / programmer
Q4	Joel Ethan	0	None
Q5	Italy Betty Fisher	12	(country): Italy → France / Canada / USA
Q6	Spielberg Schwarzenegger	58	(cast name): Schwarzenegger→ Meredith Brooks / Jim Conroy / Dean Spunt
Q7	Terminator 3 cast Sarah	19	(cast name): Sarah → Nick Stahl / Claire Danes / Kristanna Loken
Q8	Panic Room 2001	11	(year): 2001 → 2002 (title): Panic Room → Promised Land / Nowhere Road
Q9	Ettore The Man movie	1189	(director name): Ettore → Ethan Coen / Salvatore Maira / Massimo Sani
Q10	boy death ghost love	992	(keyword): love → orphanage / bully / bomb

in Section 3.5.2, one condition for the relationship to be held is that the range of the second label should be contained by the first (line 1-2). After that, we need to make sure every bit that appears in the second label also appears in the first. Since the bitmap length of the two may not be the same, we shrink the first bitmap as the same length as the second (line 3). Then bit checking can be done by only doing a logical AND operation on two bitmaps (line 4). Finally, every element in the second label's vector should be less than or equal to the corresponding element in the first label's vector (line 5).

3.7 Experiments

We have conducted extensive experiments to verify the effectiveness, efficiency and scalability of our approach. For expository convenience, we refer to our MisMatch Detector & Suggester as *MisMatch D&S*. We have also built an interactive XML keyword search engine called XClear [104], where the *MisMatch D&S* is incorporated.

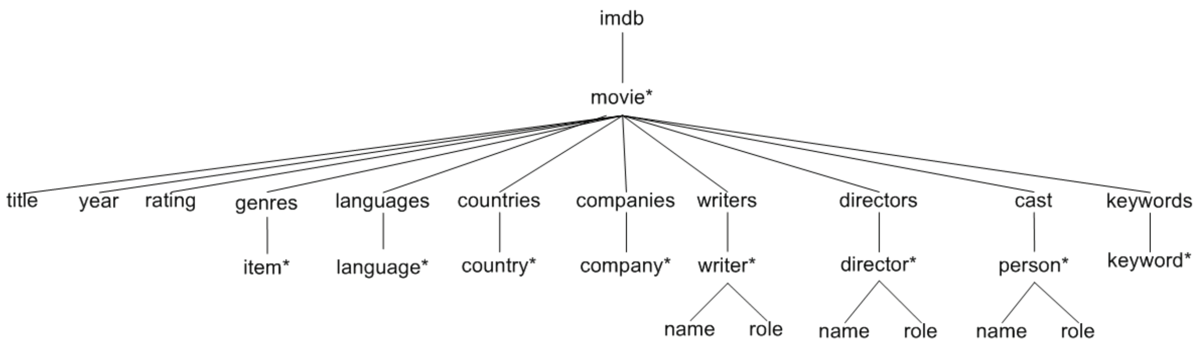


Figure 3.4: Schema Graph of IMDB Dataset

3.7.1 Experimental Settings

All experiments are conducted on a 2.83GHz Core 2 Quad machine with 3GB RAM running 32-bit windows 7. All codes are implemented in Java. Berkeley DB Java Edition [1] is used to store all indexes for our algorithms.

Data Set. Three real datasets are tested: (1)IMDB[3] 90MB, where around 200,000 movies of recent years are selected in our dataset. Each movie contains information like title, rating, director, etc. (2) DBLP 520MB, which contains publications since 1990. (3) IEEE Publication 90MB from INEX[4].

Query Set. Our query set contains 18 queries for each of the datasets, all of which are collected from the real-world user log data of our system. 10 sample queries for IMDB and their best-3 suggested queries (if any) are shown in Table 3.1. For better understanding of the queries, the schema tree of the IMDB dataset is given in Figure 3.4. Besides, 1000 random queries are generated for each dataset as well (see Section 3.7.7), where the max (average) number of results is 2691 (169).

User Study Methodology and Ground Truth. For each dataset, we employ 15 assessors to pick up the queries with the MisMatch problem, and their judgements are based on both the queries given and their respective results [5]. We obtained the

[3]http://www.imdb.com/interfaces

[4]https://inex.mmci.uni-saarland.de/

[5]Since different users could have many different search intentions even for the same keyword

ground truth by judging a query to have the MisMatch problem if at least 8 of the 15 assessors agree on that. Eventually, 9 (10, 10) out of the 18 queries for IMDB (DBLP, IEEE) have the MisMatch problem.

Keyword Search Method. Here we choose SLCA [99], which is one of the most efficient ones so far. Since no SLCA-based search method proposed so far has result ranking component, for the experiment we adopt the result ranking scheme of XRank [31].

Table 3.2: Sensitivity of the MisMatch Detector

	IMDB dataset	DBLP dataset	IEEE dataset
Precision	90%	91%	100%
Recall	100%	100%	100%

3.7.2 Frequency of the MisMatch Problem

We have done a survey among 15 participants. Each participant is required to issue 30 queries in XClear [104], an XML keyword search engine, to find some movies they are interested in the IMDB dataset. Each participant is asked to judge whether her queries have the MisMatch problem according to the query results. The same experiments are also conducted on DBLP and IEEE datasets. We find that, averagely users suffered from such a problem for 27% of their queries.

3.7.3 Sensitivity of the MisMatch Detector

With the ground truth obtained from the human assessors, as discussed in Section 3.7.1, we study the precision and recall of our MisMatch detector. Let A be the set of queries that do have MisMatch problem. Let B be the set of queries that

query, we do not want to confine the search intentions to some pre-defined options. So we did not show any pre-defined search intentions to users for reference or let them choose, which could affect users' decision.

our detector claims to have MisMatch problem. Then the precision=$|A \cap B|/|B|$, while recall=$|A \cap B|/|A|$. The result for queries on each dataset is shown in Table 3.2. We find:

(1) Our detector achieves a perfect recall, i.e. we do not miss any query that does have MisMatch problem. This is because the detector checks *all* the results of Q before deciding whether Q has MisMatch problem (by Definition 3.7).

(2) A non-perfect precision tells that we may accidentally identify some queries without MisMatch problem as problematic. E.g. for Q_4 'Joel Ethan' in Table 3.1 issued on IMDB, no person in database has such a name. For such a query, it is ambiguous that whether the user intends to find a movie related to two persons, or to find a person with that name which does not exist. In this case, our approach infers *movie* as the TNT, but some users may think it is to find one person but with the name wrongly input. Note that in fact no existing approach can solve the ambiguous query thoroughly [8].

3.7.4 Quality of the Suggested Queries

We first have a glance at how explanations and suggestions look like for real-world queries in Table 3.1. For Q8, 'Panic Room' ('2001') is associated with the node of type *title* (*year*), but no single movie contains all keywords. Naturally, one suggestion is to find a movie with the same title but different year (e.g. '2001'→'2002'), or to find a movie with the same year but different title (e.g. 'Panic Room'→'Promised Land'). Note that we do not replace the keyword(s) directly, instead we first replace the *keyword match node*, then derive the keywords as replacement. The term inside the parenthesis in Table 3.1 indicates *the type of the node* in which the replacement is involved. The left hand side of the arrow is the keyword(s) which lead to the mismatch problem (explanation part). Q3 has 3061 suggestions, because Q3 has a large number of results, and our suggester works by checking each result to generate suggestions (if any).

Evaluation Method

We select the queries with the MisMatch problem for each dataset to conduct a user study.

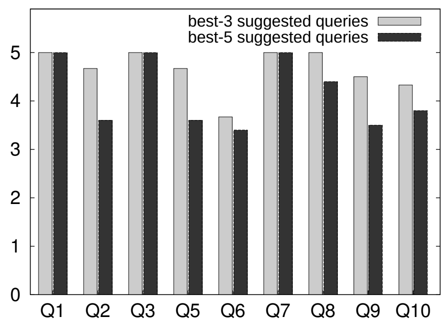
To conduct a fair evaluation, we are aware of two things. *First*, we invite both experts and novices to participate the task of scoring the suggested query. For DBLP and IEEE, we ask three CS research students and three undergraduates in other faculties; for IMDB, we ask three movie fans and three non-fans. The participants are shown the matching results of each query, the best-5 suggested queries together with the corresponding sample query results. *Second*, the participants are asked to score the quality of each suggested query by using the Cumulated Gain-based evaluation (CG) metric [40] (from 0 to 5 points, 5 means best while 0 means worst). In contrast to traditional metrics like precision and recall which adopt a binary judgement (yes or no), CG is aware of the fact that all results are not of equal relevance to user.

Evaluation of Overall Quality

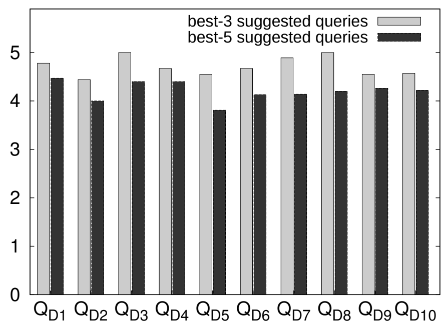
The average scores for best-3 and best-5 suggestions are shown in Figure 3.5 [6]. We can find for queries with the MisMatch problem, our approach is able to find reasonable suggested queries for them, and subsequently it leads to more meaningful results; the scores for best-3 suggestions are always higher than those of best-5, which also shows the effect of our query ranking scheme.

Although our suggested queries can lead to better query results, some are still

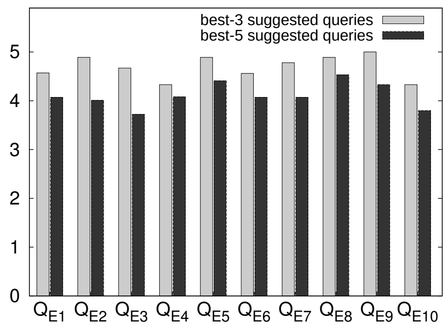
[6]Here by default we adopt $\tau = 0.9$. Experiment on effects of threshold setting is discussed in Section 3.7.4.



(a) IMDB



(b) DBLP



(c) IEEE

Figure 3.5: Average Quality Measure of Suggested Queries

given low scores by some participants because new keywords and old keywords are not semantically similar, such as the replacement for Q10 in Table 3.1. But considering semantics is out of the scope of this chapter.

Most likely, the best-3 suggested queries will be viewed by the struggling users. So in the rest of the chapter, when we talk about the quality of the suggested queries, we mean the average score of the best-3 suggested queries.

Study of the query ranking scheme

We further study how the proposed ranking factors for ranking suggested query affect the overall quality of suggested queries. The ranking factors include cn, dt and $\sum D$, as discussed in Section 3.4.3. The scores for the suggested queries of each case are shown in Table 3.3. Please ignore the choice of τ for the time being.

By comparing the scores in a columnwise way, we find:

(1) The model taking all ranking factors always outperforms any models that miss one of the three ranking factors.

(2) Without considering the distinguishability of the keywords to be replaced (i.e., $\sum D$), the suggested query quality decreases more than the case without any of the other two factors. It shows that distinguishability plays an important role.

Table 3.3: Suggestion Quality w.r.t. different τ and ranking factors

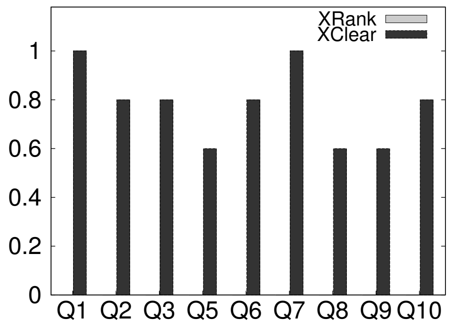
	τ	all ranking factors	no cn	no dt	no $\sum D$
IMDB	0.9	4.63	4.30	4.37	4.13
	0.6	4.63	4.30	4.37	4.13
	0.3	4.63	4.30	4.37	4.13
	0.0	4.63	4.30	4.37	4.13
DBLP	0.9	4.71	4.39	4.39	4.13
	0.6	4.71	4.36	4.42	4.18
	0.3	4.71	4.36	4.42	4.18
	0.0	4.71	4.36	4.42	4.18
IEEE	0.9	4.68	4.34	4.41	4.18
	0.6	4.68	4.34	4.42	4.19
	0.3	4.68	4.34	4.42	4.19
	0.0	4.68	4.34	4.42	4.19

Study of distinguishability threshold

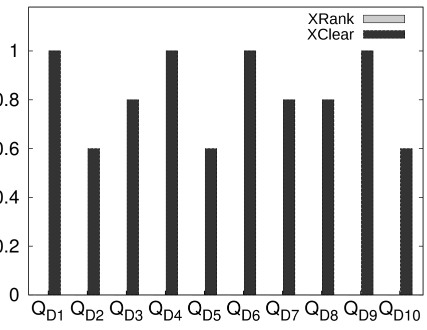
Impact of parameters on Effectiveness

Besides the query ranking scheme, recall Section 3.4.2, the choice of the distinguishability threshold τ will determine what 'important' keywords to keep in suggestions, thereby may lead to different candidates for suggested queries Q's, which in turn may affect the overall quality of Q's. Therefore, we adopt 4 choices of τ, from strong (0.9) to weak (0), as shown in Table 3.3.

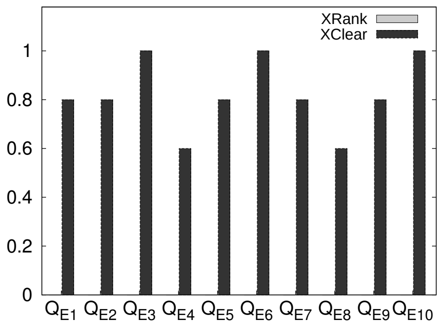
By comparing the scores in a rowwise way, we can see that the best suggested queries usually do not change even when we set a smaller threshold τ. It is because we have already found the best suggested queries when we set a high τ like 0.9, since preserving the keywords with high distinguishability is more reasonable as discussed in Section 3.4. Later we will also study the impact of τ on the efficiency of our approach in Section 3.7.7.



(a) IMDB (XRANK's precision = 0)



(b) DBLP (XRANK's precision = 0)



(c) IEEE (XRANK's precision = 0)

Figure 3.6: Precision for Top-5 results of XClear vs. XRANK

3.7.5 Comparison to XRank

To further verify the importance of the *MisMatch D&S*, we compare our XClear system that incorporates the *MisMatch D&S* with a well-known LCA-based search engine XRank [31]. For queries with MisMatch problem, XRank may still return a ranked list of query results while XClear returns a ranked list of suggested queries. Therefore, for a fair comparison, we retrieve the results for each suggested query produced by XClear, rank them using XRank's result ranking scheme, and then pick the top-5 results to compare with the top-5 results of XRank. A result is regarded as *relevant* if 8 of the 15 assessors agree on that; otherwise it is regarded as *irrelevant*. Figure 3.6 shows the precision of top-5 results of queries on our three datasets, which is calculated as *(number of relevant results in top-5 results)*/5. We find for queries with MisMatch problem, XRank cannot find any relevant result, leading to a precision of zero. Because XRank is not aware of the fact that what user searches for may not exist, but return the full matches as 'perfect results', which are usually the whole XML data tree.

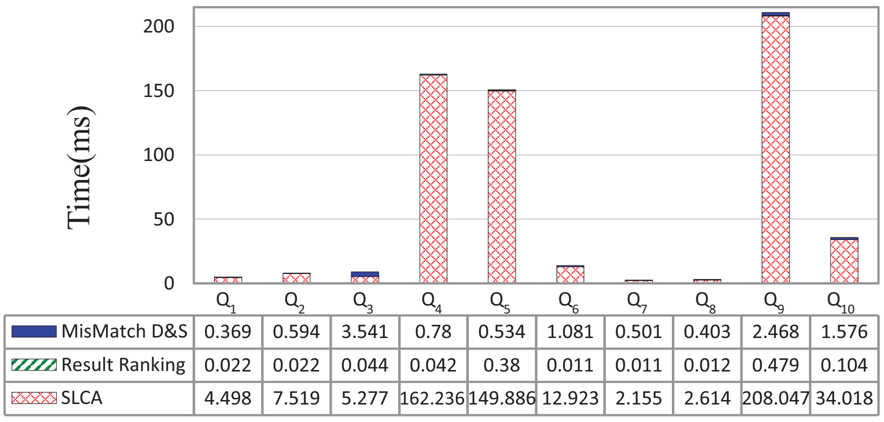
3.7.6 Sample Query Processing Time

For each query in Table 3.1, we run our algorithm 10 times and collect the average processing time on hot cache, as shown in Figure 3.7(a). The query result ranking time is too small to display. Moreover, we record the time used by the *MisMatch D&S* part. We have three observations from Figure 3.7(a):

(1) The *MisMatch D&S* only takes a small portion of the whole query processing time. On average, it is around 4% for our query set. For the queries on which MisMatch D&S spends less than 1ms, it is too small to display in Figure 3.7(a). Besides, on average the detector spends about 1/40 time of the suggester because it only needs to check the node type of the results as discussed in Section 3.5.

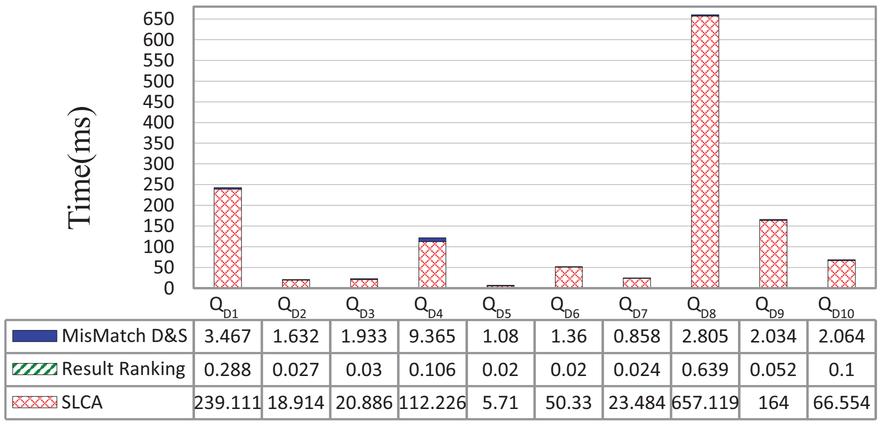
(2) When more suggested queries are generated, the processing time of MisMatch D&S is relatively longer. E.g., as we can see in Table 3.1, Q3 generates more suggested queries than the other queries, so MisMatch D&S consumes more time.

(3) For the query that has no MisMatch problem, MisMatch D&S introduces a negligibly small time as compared to the query evaluation time. Because it will terminate once it finds a query result without the MisMatch problem. E.g. for Q4 which intends to find the movie by two directors, since there exist such kind of



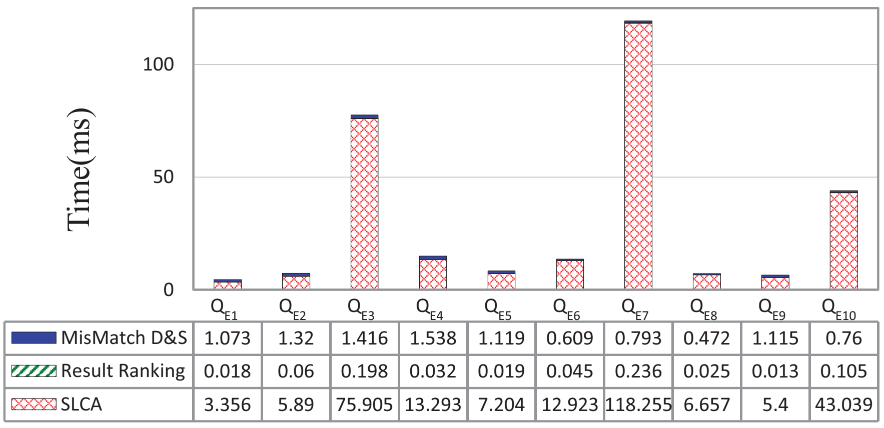
	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}
■ MisMatch D&S	0.369	0.594	3.541	0.78	0.534	1.081	0.501	0.403	2.468	1.576
▨ Result Ranking	0.022	0.022	0.044	0.042	0.38	0.011	0.011	0.012	0.479	0.104
▩ SLCA	4.498	7.519	5.277	162.236	149.886	12.923	2.155	2.614	208.047	34.018

(a) IMDB (MisMatch D&S = MisMatch Detector & Suggester)



	Q_{D1}	Q_{D2}	Q_{D3}	Q_{D4}	Q_{D5}	Q_{D6}	Q_{D7}	Q_{D8}	Q_{D9}	Q_{D10}
■ MisMatch D&S	3.467	1.632	1.933	9.365	1.08	1.36	0.858	2.805	2.034	2.064
▨ Result Ranking	0.288	0.027	0.03	0.106	0.02	0.02	0.024	0.639	0.052	0.1
▩ SLCA	239.111	18.914	20.886	112.226	5.71	50.33	23.484	657.119	164	66.554

(b) DBLP



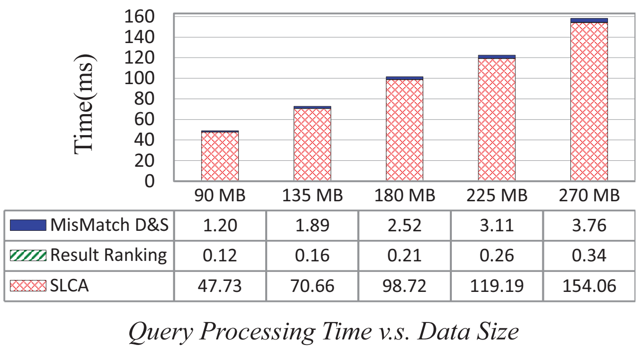
	Q_{E1}	Q_{E2}	Q_{E3}	Q_{E4}	Q_{E5}	Q_{E6}	Q_{E7}	Q_{E8}	Q_{E9}	Q_{E10}
■ MisMatch D&S	1.073	1.32	1.416	1.538	1.119	0.609	0.793	0.472	1.115	0.76
▨ Result Ranking	0.018	0.06	0.198	0.032	0.019	0.045	0.236	0.025	0.013	0.105
▩ SLCA	3.356	5.89	75.905	13.293	7.204	12.923	118.255	6.657	5.4	43.039

(c) IEEE

Figure 3.7: Processing Time for some Sample Queries

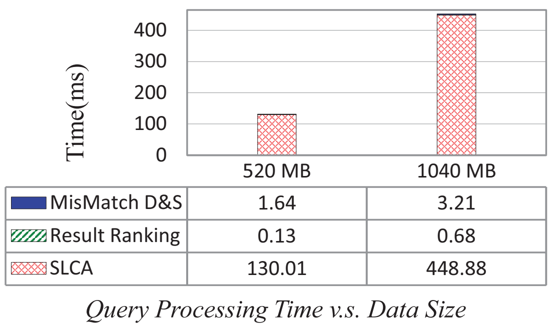
movies, Q4 does not have the MisMatch problem, and our MisMatch D&S takes only 0.05ms.

Figure 3.7(b) and 3.7(c) show the processing time for 10 (out of the total 18) queries on DBLP and IEEE, where we can get similar observations.



	90 MB	135 MB	180 MB	225 MB	270 MB
■ MisMatch D&S	1.20	1.89	2.52	3.11	3.76
▨ Result Ranking	0.12	0.16	0.21	0.26	0.34
▩ SLCA	47.73	70.66	98.72	119.19	154.06

(a) IMDB



	520 MB	1040 MB
■ MisMatch D&S	1.64	3.21
▨ Result Ranking	0.13	0.68
▩ SLCA	130.01	448.88

(b) DBLP

Figure 3.8: Impact of Data Size.

3.7.7 Scalability Test

Recall that our detector checks all results of a query before concluding the existence of the MisMatch problem, and for each query result, our suggester tries to derive suggested query. Therefore, the processing time of the MisMatch D&S should be dependent on the number of suggested queries found, which in turn depends on

- the size of the XML data being queried, and
- the choice of the distinguishability threshold τ, and
- the number of results investigated by MisMatch D&S

Sample Queries

Firstly, we conduct our scalability test by studying the impact of increasing data size on the MisMatch D&S. We run the queries on IMDB and DBLP with different sizes. Figure 3.8 shows the average processing time of one query on the datasets, where we have two observations.

(1) The processing time of the MisMatch D&S increases linearly w.r.t. the data size. Because larger data size leads to possibly larger number of results, and our D&S needs to check all results to decide the MisMatch existence and find suggestions based on each result.

(2) As the query processing time increases w.r.t. the data size as well, the MisMatch D&S only takes around 4% of the whole query processing time regardless of the data size.

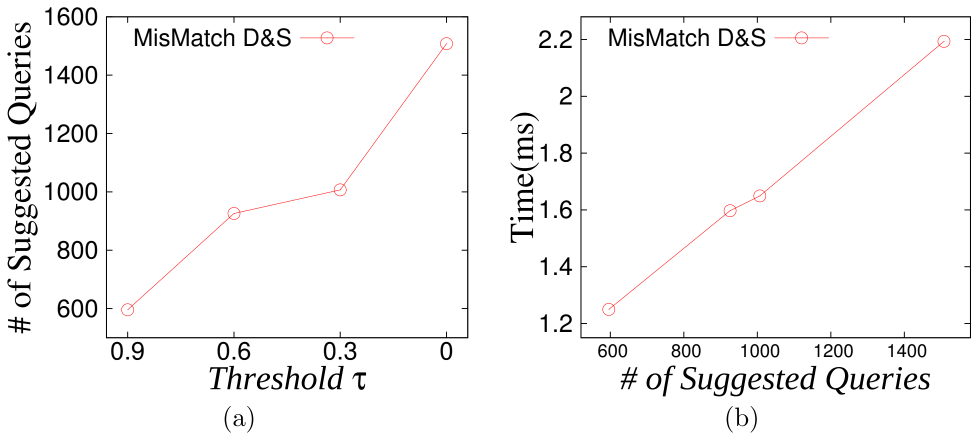
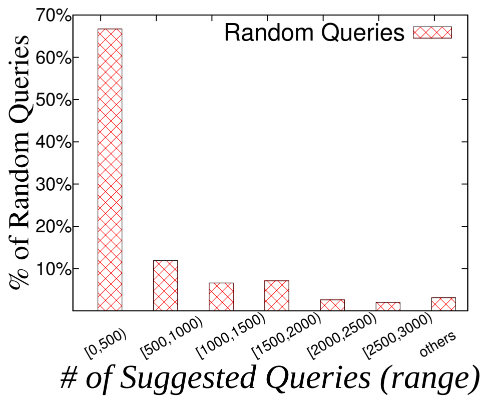


Figure 3.9: Impact of Distinguishability Threshold τ

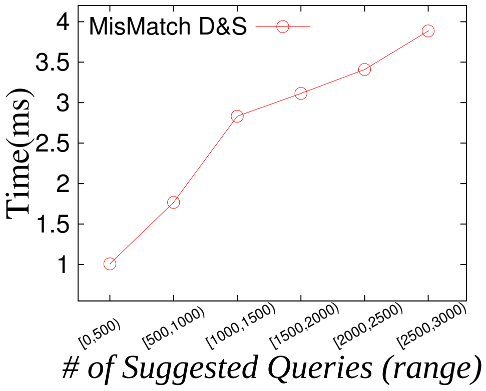
Secondly, we study the impact of the distinguishability threshold τ on the processing time of our MisMatch D&S. Figure 3.9 shows the average number of suggested queries generated for one query w.r.t. different distinguishability threshold τ and the corresponding processing time, where the choice of τ is same as that of the query quality study (in Section 3.7.4). As we can see, more suggested queries will be generated when τ is set to be smaller. Meanwhile, it will take longer to process. Because when threshold τ is set lower, more keywords will be considered as with acceptably high distinguishability, and we will check more TNT nodes and therefore find out more suggested queries. As discussed in Section 3.7.4, most likely, setting τ to 0.9 can find the same best suggested queries as setting τ to 0.6, 0.3 and even 0.0. So we set τ to 0.9 as a balance between efficiency and effectiveness. To summarize, *MisMatch D&S* takes a very small portion of the keyword query processing time, while can come up with some helpful suggested queries to users for possible MisMatch problem.

Random Queries

Besides the real-world sample queries, we further study the performance of our D&S over random queries. Keywords in IMDB dataset are randomly picked to form queries of length 2∼5 and those with MisMatch problem will be kept. We record the first 1000 of such queries and count the suggested queries output by our D&S. The distribution of these queries with different ranges for the number of suggestions is shown in Figure 3.10(a), from which we find most queries will result in suggested queries no larger than 500. Similar to our findings on sample queries, Figure 3.10(b) reports the linear relationship between the D&S time and the number of suggested queries on random queries.



(a) Distribution



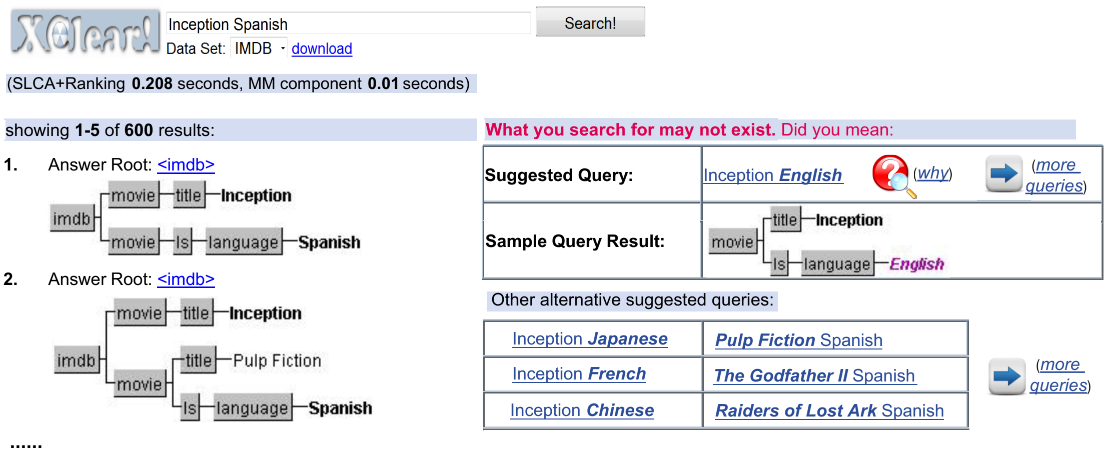
(b) D&S Time

Figure 3.10: Scalability Test of Random Queries

3.8 XClear Demo System

Addressing the MisMatch problem, We have built an interactive XML keyword search engine called XClear [104]. It can detect the MisMatch problem and show users why the MisMatch problem exists, as well as provide result-driven suggested queries to bridge the mismatch gap.

Figure 3.11 shows a screenshot of XClear for a query Q='Inception Spanish' in order to find the Spanish version of a movie Inception. On the left hand side it shows the query results returned by a widely adopted matching semantics SLCA [99]. As we can see from the results, there is no movie Inception with language Spanish. Therefore, help is needed for the user.


Inception Spanish

Data Set: IMDB download

(SLCA+Ranking **0.208** seconds, MM component **0.01** seconds)

showing **1-5** of **600** results:

1. Answer Root: <imdb>

2. Answer Root: <imdb>

......

What you search for may not exist. Did you mean:

Suggested Query:	Inception ***English***	(why)	(more queries)
Sample Query Result:			

Other alternative suggested queries:

Inception ***Japanese***	***Pulp Fiction*** Spanish
Inception ***French***	***The Godfather II*** Spanish
Inception ***Chinese***	***Raiders of Lost Ark*** Spanish

(more queries)

Figure 3.11: Suggested Queries & Sample Query Result

Addressing such a problem, as shown in the right part of Figure 3.11: (1) XClear gives a notification "What you search for may not exist" to the user. This is a crucial part to form a complete solution to the MisMatch problem. Because without the notification, users have to struggle with reading throughout the results until realizing what they search for may not exist. (2) It provides the best suggested query and its sample result. (3) A "*why*" button (next to the suggested query) is provided for users to get further *reasoning* on why we generate this suggested query.

Please refer to Appendix A for more details of the XClear system.

3.9 Conclusion

In this chapter, we first identified and defined the MisMatch problem, in which what users intend to search for does not exist in the XML data. In such a case, existing keyword search methods will still return a list erroneous mismatch results. All existing keyword search methods for XML tree are LCA-based, which try to find some subtrees containing all the query keywords as query results, regardless of users' search intention. We proposed a low-cost post-processing algorithm on the results of query evaluation to detect the MisMatch problem and generate helpful suggestions to users. The detection is done by inferring users' possible search target, called *Target Node Type*, based on each query result. We choose to take a conservative approach: we only judge a query to have the MisMatch problem if none of the query results matches its corresponding Target Node Type. Such a conclusion holds for all users with different intentions. Our detection method can also be easily extended to detect whether a query has MisMatch problem for a particular search intention: we can first classify the query results by different TNT; For the results of a particular TNT, if all of them miss the target, then we can judge the query has MisMatch problem for that particular search intention.

After detection, if the query is without the MisMatch problem, the original results will be returned without any suggestion. Otherwise, query suggestion will be generated for the queries with MisMatch problem. First, it will measure the importance of the query keywords according to a newly defined measure inspired by Term Frequency * Inverse Document Frequency (*tf*idf*), called distinguishability. Second, approximate results containing the important keywords will be discovered in the XML data. Third, to form the suggested queries, new keywords will be found within the approximate results to replace the less important keywords in the original query. Both of these keywords are required to be from the same type of nodes in order to make sure the semantics of the keywords are the same. Finally, a score function is proposed to rank the suggested queries taking the following three factors into consideration: the number of keywords which need to be replaced, the sum of distinguishability of the keywords that need to be replaced and the compactness of the approximate result. To discover the approximate results efficiently, a novel bitmap labeling scheme is also proposed. The empirical study on three real datasets in experiments demonstrates the effectiveness and efficiency of our approach. It evaluates the detection accuracy and suggestion quality, as well as the efficiency and scalability.

Our approach has four main features: (1) both detector and suggester are result-driven; (2) it adopts explanations, suggested queries and their sample results as the output to users, helping users judge whether the MisMatch problem is solved without reading all query results; (3) it is portable as it can work with any LCA-based matching semantics and orthogonal to the choice of result retrieval method; (4) it is lightweight as it occupies a very small proportion of the whole query evaluation time.

CHAPTER 4

MISMATCH PROBLEM IN KEYWORD SEARCH OVER XML WITH ID REFERENCES

4.1 Introduction

In Chapter 3 we have discussed the solution to the MisMatch problem for keyword search over XML tree, where ID references are not considered. When we consider the ID references in the XML, an XML document will be modeled as a digraph rather than a tree, which we called *XML IDREF digraph*.

In this chapter, we are trying to extend our solution for XML tree model onto XML IDREF digraph with ID references considered. We propose a novel method to transform an XML IDREF digraph to a tree model such that we can exploit existing XML tree search methods. Our solution can outperform the traditional

XML IDREF digraph search methods by orders of magnitude in efficiency while generating a similar set of results as existing XML IDREF digraph search methods. What is more, in such a way, our proposed mismatch solution dedicated for XML tree in Chapter 3 can be applied to XML IDREF digraph.

For keyword search over XML IDREF digraph, it poses new challenges for solving the MisMatch problem. Because the matching semantics and keyword search methods for XML IDREF digraph are different from those for XML tree. In order to solve the MisMatch problem in XML IDREF digraph, we will first compare the difference between keyword search over XML IDREF digraph and XML tree.

XML documents usually contain some ID nodes and IDREF nodes to represent reference relationships among the data. For example, Figure 4.1 shows an XML document about a company with *project*, *part* and *supplier*. Each node is assigned a unique Dewey label [93]. Every *part* used by each *project* has a reference indicating its *supplier*. An XML document with ID/IDREF is usually modeled as a digraph, where the keyword query results are usually computed by graph traversal [37, 26, 44, 35]. Then the problem is reduced to the problem of finding Minimal Steiner Tree (MST) or its variants in a digraph, where an MST is defined as a minimal subtree containing all query keywords in either its leaves or root. Since this problem is NP-complete [28], a lot of works are interested in finding the "best" answers of all possible MSTs, i.e. finding top-K results according to some criteria, like subtree size, etc. However, the matching semantics, i.e. MST, is also defined without considering users' search intention. Therefore, mismatch results are still possible to be returned by existing methods in keyword search over XML IDREF digraph.

As a comparison, if we do not consider ID/IDREF, an XML document can be modeled as a tree. Keyword search on an XML tree can be much more efficient based on the tree structure. The results are defined as minimal subtrees containing

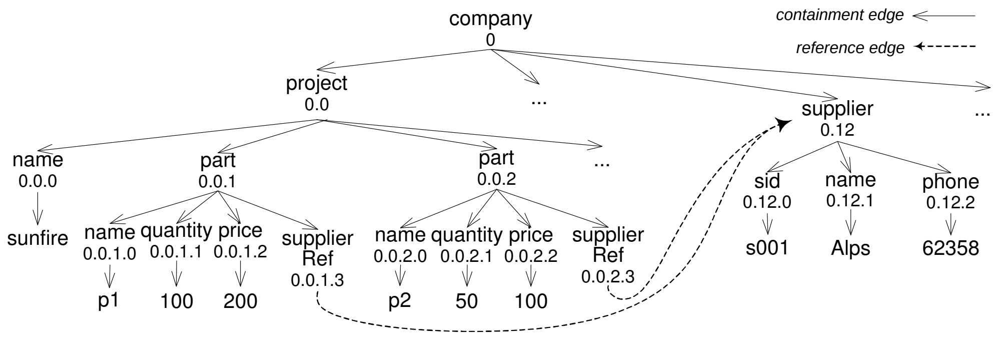


Figure 4.1: An Example XML Document (with Dewey Labels)

all query keywords, which is actually a variant of MST adapted to XML tree. Because in a tree, finding an MST for a set of nodes reduces to finding the lowest common ancestor (LCA) of that set of nodes, which can be efficiently addressed by node label computation. For example, if we do not consider the ID/IDREF in Figure 4.1, given a query Q="p1 price", a node labeled 0.0.1.0 matches keyword "p1" and another node labeled 0.0.1.2 matches keyword "price", then the MST connecting these two nodes is actually the subtree rooted at their lowest common ancestor (LCA), i.e. node 0.0.1. Calculating the LCA simply requires calculating the common label prefix of those two nodes, i.e. 0.0.1 is the prefix of 0.0.1.0 and 0.0.1.2. It is very efficient and does not need any graph traversal.

There are abundant efficient XML tree search methods available, which efficiently calculate the query results based on node labels rather than graph traversal. They build inverted lists of query keywords, in the form of ($keyword$: $dewey_1, dewey_2, dewey_3, ...$), where $dewey_i$ represents the label of a node containing the *keyword*. More over, we have already proposed a solution for the MisMatch problem for keyword search over XML tree in Chapter 3, which makes use of the concept of Target Node Type and Distinguishability.

We observe that, an XML IDREF digraph is indeed a tree with a portion of reference edges. Such an observation offers a great opportunity to solve the Mis-

Match problem in keyword search over XML IDREF digraph, as well as speeding up the processing the keyword query on XML IDREF digraph. In this chapter, instead of adopting traditional graph search methods, we propose a novel approach to transform an XML IDREF digraph to a tree model, such that we can exploit XML tree search methods to evaluate keyword queries on XML IDREF digraph. Then our solution to the MisMatch problem in XML tree still applies to XML IDREF digraph. Meanwhile, such a method can also accelerate the query evaluation for keyword search over XML IDREF digraph. The rest of the chapter is organized as follows. Preliminaries are in Section 4.2. We discuss how to transform an XML IDREF digraph to a tree model for efficient query evaluation in Section 4.3 and how it works on complex reference patterns in Section 4.4. Further extension of our approach is in Section 4.5. Solving the MisMatch problem in XML IDREF digraph is discussed in Section 4.6. The algorithm is presented in Section 4.7. Experiments are in Section 4.8 and we conclude in Section 4.9.

4.2 Preliminaries

4.2.1 Semantics and Data Model

Similar to most of the existing works in XML keyword search, we assume that there is no outer semantics provided. In other words, we assume that we only have the XML document itself with the accompanied schema specification, like DTD or XML Schema. The following information from DTD or XML Schema will be used in our solution: 1) which attributes in the XML document are the ID attributes or IDREF attributes (to identify ID reference links); 2) the number of possible occurrences of a sub-element/attribute appearing under a parent node (to be used in our MisMatch solution). Outer semantics, such as ER model [20], ORA-SS model

[27, 63], can help identify in the XML data that which nodes represent objects, object attributes, relationships, relationship attributes, etc. Such information is not conveyed in an XML document or its accompanied schema specification. Instead, it only stores data as a nested tree structure with some special reference attributes being able to point from one node to another. Therefore, we model an XML document with ID references as a digraph, where each node of the graph corresponds to an element of the XML data, with a tag name and (optionally) some value. Each containment relationship between a parent node a and a child node b in the XML data corresponds to a containment edge in the digraph, represented as $a \rightarrow b$. Each ID/IDREF reference in the XML data corresponds to a reference edge in the digraph, represented as $a \dashrightarrow b$, where a is the IDREF node and b is the ID node. Thus an XML IDREF digraph is denoted as $G = (V, E, R)$, where V is a set of nodes, E is a set of containment edges and R is a set of reference edges.

We use T_n to denote the query result rooted at node n. A node n is usually represented by its label or *tag:label*, where tag is the tag name of n. To accelerate the keyword query processing on XML tree model, existing works adopt the dewey labeling scheme [93], as shown in Figure 4.1.

4.2.2 Reference Types

If the reference edges are not considered, an XML IDREF digraph will reduce to an XML tree. There are three types of reference edges in an XML IDREF digraph: *basic references* (as mentioned in our data model), *sequential references* and *cyclic references.* When an object a references an object b, while b also references a third object c, sequential references occur. Cyclic references happen when containment edges and reference edges form a cycle in an XML IDREF digraph.

Example 4.1. *Figure 4.4(a) shows an example of sequential references: one* part *has a reference to one* supplier*, which in turn has a reference to an* employee *as her manager. The references among* part*,* supplier *and* employee *form the sequential references. Figure 4.5(a) shows an XML IDREF digraph with cyclic references: a* book *has some references to its authors, while each* author *has some* recommend *references to some books. If an author recommends its own book, a cycle is formed.* □

4.3 Transforming Query Processing over XML IDREF Digraph to XML Tree for Basic References

In order to fully exploit the power of tree search methods over the XML digraph, we realize two challenges to tackle: (1) how to transform an XML IDREF digraph to a proper tree model, which can work with different reference patterns; (2) how to apply existing tree search techniques onto such a tree model. We start addressing these challenges by focusing on the case of *basic references* first, then discuss how the proposed solution can handle sequential and cyclic cases in Section 4.4.

4.3.1 Naive Approach: Real Replication

As shown in Figure 4.1, every IDREF node in an XML IDREF digraph points to a particular object with a unique ID value. An object is in the form of a subtree. Therefore, a straightforward yet naive transformation is to just to make a real replication of all such subtrees being referenced. For every reference edge $a \dashrightarrow b$ in the XML IDREF digraph, we make a replication of the subtree T_b rooted at b and put it under a. Figure 4.2 shows a transformed XML tree based on the

XML IDREF digraph in Figure 4.1, where the subtrees in dotted circles are the replication of the subtree $T_{0.12}$.

The transformed XML tree is identical to the original XML IDREF digraph in the sense that they can infer each other. As a result, any existing keyword search method designed for XML tree can be applied on it. They can now find the query results which previously can only be found by graph search methods.

Example 4.2. *Suppose a user wants to find the parts provided by supplier Alps by issuing Q="Alps part" on the XML IDREF digraph in Figure 4.1. There are a lot of possible MSTs connecting the keywords, and two of them are as below:*
(1) an MST rooted at part:0.0.1, i.e. 0.0.1→0.0.1.3-->0.12→0.12.1;
(2) an MST rooted at part:0.0.2, i.e. 0.0.2→0.0.2.3 -->0.12→0.12.1;
According to our real replication method, we will transform the XML IDREF digraph to a tree by replication, which is shown in Figure 4.2. Then we can apply an XML tree search method, say ELCA [100], to the transformed tree. ELCA will first get the following inverted lists of dewey labels for each query keyword:
(1) "Alps": 0.0.1.3.1, 0.0.2.3.1, 0.12.1
(2) "part": 0.0.1, 0.0.2, ...

Then it will scan the inverted lists and compute the following results: (1) the subtree $T_{part:0.0.1}$, which is the LCA computed from node 0.0.1.3.1 and node 0.0.1; (2) the subtree $T_{part:0.0.2}$, which is computed from node 0.0.2.3.1 and 0.0.2. They are the same as the two sample results found in the XML IDREF digraph. □

However, even though the real replication approach can work well for the case of basic references, it is **not usable in practice** because:

- The number of nodes will increase due to the replication of subtrees. We will show in Section 4.4 that, in the worse case, the number of nodes will increase

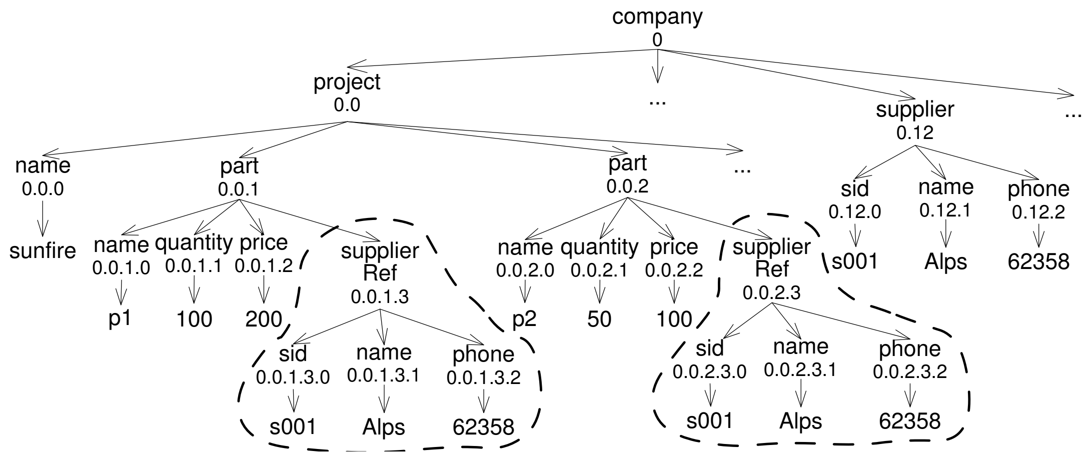


Figure 4.2: Naive Method: Real Replication

exponentially for the case of sequential references and cyclic references. The space cost is unacceptable in practice.

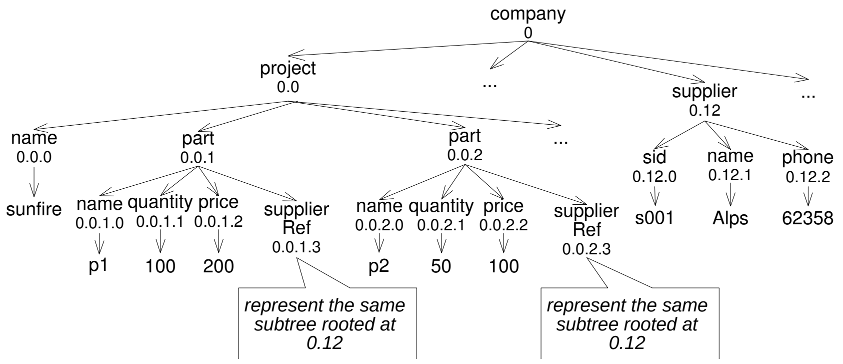
- Some duplicate results may be generated (as shown in Example 4.3).

Example 4.3. *If we issue a query Q="Alps phone" to find the phone number of supplier Alps in Figure 4.1, the real replication method will get the transformed XML tree in Figure 4.2 and do the keyword search on it. By ELCA search method, we get three results: $T_{supplier:0.12}$, $T_{supplierRef:0.0.1.3}$ and $T_{supplierRef:0.0.2.3}$ respectively. The last two results, which are the same as the first one, are actually redundant. Because they are found within the replicated subtrees, while the same results should have already been found in the original subtree.* □

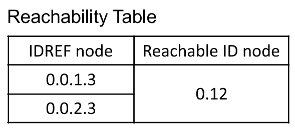
4.3.2 Our Approach: Virtual Replication

As discussed in the previous section, real replication is not usable in practice. From Example 4.3 we observe that, a result is **redundant** if it is found within the replicated subtrees, because it must have been found in the original subtree as well. Thus, a result is **non-redundant** only if the root of the result is not within any replicated subtree. Based on this observation, we find that the cost of replicating the subtrees is not necessary because we do not need to search within any replicated subtree.

Instead, we propose to use a special node, i.e. the IDREF node, to virtually represent the whole replicated subtree (without inducing any new node), which is able to find the same set of non-redundant results as the real replication method. This is what we call **virtual replication**. For instance, Figure 4.3(a) shows the idea of using one node to represent the whole replicated subtree. As compared to Figure 4.2 of real replication, here we use node supplierRef:0.0.1.3 in Figure 4.3(a) to represent the whole replicated subtree under it.



(a) Part 1: XML tree


Reachability Table

IDREF node	Reachable ID node
0.0.1.3	0.12
0.0.2.3	

(b) Part 2: Reachability Table

Figure 4.3: Advanced Method: Virtual Replication (Two Parts)

Example 4.4. *Recall the query Q= "Alps part" in Example 4.2, the real replication method gets the following non-redundant results in Figure 4.2:* $T_{part:0.0.1}$ *and* $T_{part:0.0.2}$ *These two results are non-redundant because their roots, part:0.0.1 and part:0.0.2, are not within any replicated subtree.*

Now by virtual replication, keyword "Alps" will no longer match the node 0.0.1.3.1 and 0.0.2.3.1 in Figure 4.2. Instead, it will match two IDREF nodes 0.0.1.3 and node 0.0.2.3 in Figure 4.3(a), because we use these two IDREF nodes to represent *the whole replicated subtrees. But the final results are still the same: (1)* $T_{part:0.0.1}$*, which is computed from node 0.0.1.3 (matching keyword "Alps") and node 0.0.1 (matching keyword "part") in Figure 4.3(a); (2)* $T_{part:0.0.2}$*, which is computed from node 0.0.2.3 (matching keyword "Alps") and 0.0.2 (matching keyword "part").* □

In this manner, we do not induce any new node while it is able to get the same set of non-redundant results as the real replication method.

In other words, *Virtual Replication* will find the same set of non-redundant results as *Real Replication*. Following is the proof.

Proof. ($a \prec b$ denotes that node a is an ancestor of b. $a \preceq b$ denotes that $a \prec b$ or $a = b$.) Step 1: to prove that every non-redundant result found by *real replication* can also be found by the *virtual replication*. Let any non-redundant result found by the *real replication* be T_r, which is a subtree rooted at node r. It should be an LCA of a set of nodes $M_{real} = \{n_1, n_2, ..., n_k, \hat{n_1}, , \hat{n_2}, ..., \hat{n_l}\}$ matching the query keywords, where $\hat{n_j}$ is a match node appearing in a replicated subtree and n_i is a match node not in any replicated subtree. Each match node corresponds to a keyword in the query. The LCA relationship can be represented as two properties: ① $r \preceq n_i (1 \le i \le k)$, $r \preceq \hat{n_j} (1 \le j \le l)$; ② $\nexists r' \prec r$ s.t. $r' \preceq n_i (1 \le i \le k)$ and $r' \preceq \hat{n_j} (1 \le j \le l)$. In the *virtual replication* method, suppose we use an IDREF node $\hat{N_j}$ to represent the replicated subtree which $\hat{n_j}$ is in, we have ③ $\hat{N_j} \preceq \hat{n_j}$. Then we can prove that the same result T_r can also be calculated based on the following set of match nodes $M_{virtual} = \{n_1, n_2, ..., n_k, \hat{N_1}, , \hat{N_2}, ..., \hat{N_l}\}$. Formally, we need to prove r is the LCA of $M_{virtual}$. Since T_r is a non-redundant result, we have ④ $r \prec \hat{N_j} (1 \le j \le l)$. So from ① and ④, we have ⑤ $r \preceq n_i (1 \le i \le k)$, $r \preceq \hat{N_j} (1 \le j \le l)$. Next we need to prove ⑥ $\nexists r' \prec r$ s.t. $r' \preceq n_i (1 \le i \le k)$ and $r' \preceq \hat{N_j} (1 \le j \le l)$ by contradiction. If ⑥ is not true, with ③ we can infer that $\exists r' \prec r$ s.t. $r' \preceq n_i (1 \le i \le k)$ and $r' \preceq \hat{N_j} \preceq \hat{n_j} (1 \le j \le l)$, which contradicts with

②. So with ⑤ and ⑥ being true, r is the LCA of $M_{virtual}$ as well. Step 1 is finished. Step 2: to prove that every non-redundant result found by *virtual replication* can also be found by *real replication*. The proof is similar to step 1, which is omitted here due to space limitation. □ □

In order to know which IDREF node represents which subtree, we need a data structure to keep track of the information that which subtree will be replicated under which IDREF node. For such a purpose, we maintain a table called **reachability table**, as shown in Figure 4.3(b). The table is based on a concept called **reachable**.

Definition 4.1. ***Reachable.*** *Given an IDREF node n, if there is a directed path from n to a node m in the XML IDREF digraph, where the last edge of the path is an reference edge, then we say m is a reachable ID node of n.*

Example 4.5. *Given the XML IDREF digraph in Figure 4.1, we can find that from the IDREF node 0.0.1.3, there is a directed path from it to node 0.12, where the path ends with a reference edge. So node 0.12 is a reachable ID node for node 0.0.1.3. Similarly, node 0.12 is a reachable ID node for node 0.0.2.3.* □

For every pair of *(IDREF node, reachable ID node)*, we store it as a tuple into a table called reachability table, indexed by the attribute "reachable ID node". Every pair of *(IDREF node, reachable ID node)* means the subtree rooted at the *reachable ID node* will be replicated under the *IDREF node*. E.g., the reachability table inferred from the XML IDREF digraph in Figure 4.1 is shown in Figure 4.3(b). The reachability table can be computed offline by a breadth-first search based on each node and the algorithm is presented in Section 4.7.

4.3.3 Query Evaluation

So far we have completed the transformation from an XML IDREF digraph to an advanced tree model. Given an XML IDREF digraph $G = (V, E, R)$, we transform G to a novel tree model consisting of two parts:

1. An XML tree $GT = (V, E, \emptyset)$, which is exactly the same as G with all the reference edges dropped.

2. A reachability table *table*, which maintains the information of which subtree will be virtually replicated under which IDREF node.

Now, we will present how to make an efficient keyword query evaluation based on our transformed tree model.

As discussed in Section 4.1, existing keyword search methods on XML tree do not traverse the tree to search query results. Instead, they compute results based on nodes' labels, e.g., the dewey label. Such labels are stored in an inverted list index in form of $keyword : dewey_1, dewey_2, dewey_3, ...$, where $dewey_i$ represents a node containing the *keyword*. Any LCA-based keyword search method for XML tree will build such an index. Given a keyword query $Q = \{k_1, k_2, ..., k_n\}$, they will retrieve an inverted list for each keyword k_i, and then compute the results based on the inverted lists.

Similarly, after we transform an XML IDREF digraph to tree model in *virtual replication*, we will also build such an inverted list index. Our tree model consists of an XML tree and a reachability table. The inverted list index will be built on the XML tree, while later the reachability table will help to expand the inverted lists to handle ID/IDREF.

With the index ready, we exploit the existing XML tree keyword search methods and evaluate a keyword query on our tree model in three steps:

1. Retrieve the inverted lists for each keyword in a query.

2. Expand the inverted lists retrieved in step 1.

3. Apply an existing XML tree keyword search method to the expanded inverted lists.

Step 1. Given a query $Q =$ "$k_1k_2...k_n$", one inverted list will be retrieved from the index for each keyword. E.g., given a query Q="Alps part", based on our tree model in Figure 4.3, we will first retrieve the inverted lists as follows:
"Alps" : 0.12.1
"part" : 0.0.1, 0.0.2, ...

Take note that the keyword "Alps" only matches one node, i.e. 0.12.1, because the inverted list is built on the XML tree in Figure 4.3(a). So in this step, we only find out the nodes in the XML tree matching the keywords before replication.

Step 2. With the help of the reachability table, we will try to find out whether there is any node in the replicated subtree matching the keywords as well. We can do it in the following way: for each dewey label retrieved in step 1, we check each of its ancestors to see whether the ancestor appears in the *reachable ID node* column of the reachability table. If yes, we add the corresponding IDREF nodes to the inverted list.

E.g., for the dewey label 0.12.1 retrieved in step 1 in the above example, 0.12.1 has two ancestor (prefix): 0.12 and 0. We can find that its ancestor 0.12 appears in the *Reachable ID node* column of the reachability table in Figure 4.3(b). This means the subtree $T_{0.12}$ is reachable by some IDREF nodes and it should be replicated under those IDREF nodes. So the keyword should match those IDREF nodes as well. Then we add the corresponding IDREF nodes to the inverted list. But its ancestor 0 does not appear in the *Reachable ID node* column. After that, the expanded inverted list will be:

“Alps” : 0.0.1.3, 0.0.2.3, 0.12.1

After we do the same thing to the “part” inverted list, it will become:

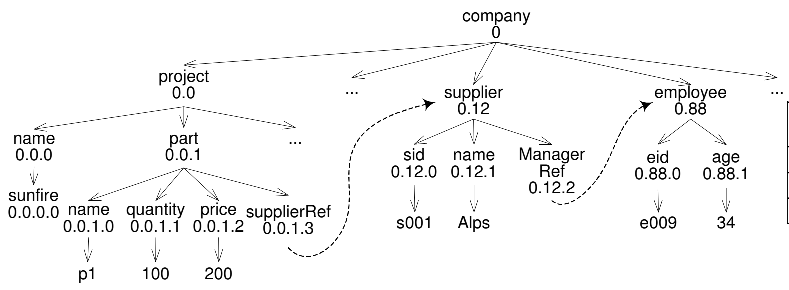
“part” : 0.0.1, 0.0.2, ...

The reachability table is organized in a B+ tree and indexed by the column *Reachable ID node*. So given the dewey label of a reachable ID node, the corresponding IDREF nodes can be retrieved efficiently.

Step 3. After step 2, the final inverted lists are ready. Now we can apply any existing keyword search methods designed for XML tree, like SLCA, ELCA, etc., as all of them operate on the inverted lists for result computation.

4.4 Sequential References and Cyclic References

Section 4.3 presents our solution on transformation and query evaluation for basic references case which does not involve sequential and cyclic references. In this section we would like to discuss how they are capable of handling the cases of sequential references and cyclic references as well.



(a) XML IDREF Digraph with Sequential References

IDREF node	Reachable ID node
0.0.1.3	0.12
0.0.1.3	0.88
0.12.2	

(b) Its Reachability Table

Figure 4.4: Constructing Reachability Table for Sequential References

4.4.1 Sequential References

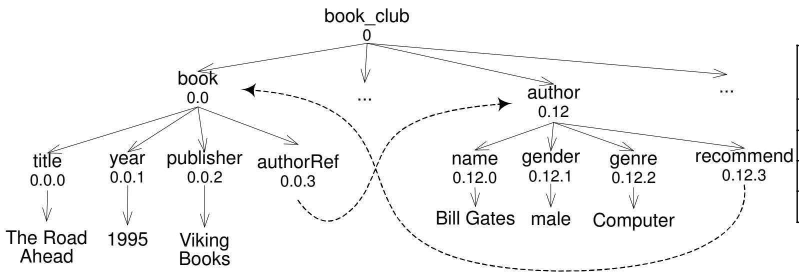
In this case, e.g. in Figure 4.4(a), to make a complete replication, the subtree rooted at employee:0.88 should be replicated not only under managerRef:0.12.2, but also supplierRef:0.0.1.3. Therefore, if we adopt the real replication approach in Section 4.3.1, the number of nodes may increase exponentially in terms of the number of levels of sequential references, as one subtree could have multiple references to some other subtrees.

For the virtual replication in Section 4.3.2, we do not need to induce any new nodes into the XML IDREF digraph. For the XML IDREF digraph in Figure 4.4(a), according to Definition 4.1, we just construct a reachability table as shown in Figure 4.4(b). E.g., there is a directed path from supplier:0.0.1.3 to employee:0.88, where the path ends with a reference edge. So node 0.88 is a reachable ID node for node 0.0.1.3.

4.4.2 Cyclic References

In the case of cyclic references, our reachability concept in Definition 4.1 is still able to handle it. E.g., in Figure 4.5(a), there is a directed path from node authorRef:0.0.3 to node author:0.12, where the path ends with a reference edge. So node 0.12 is a reachable ID node for node 0.0.3. There is also a path from node authorRef:0.0.3 to node book:0.0, where the path ends with a reference edge. So node 0.0 is also a reachable ID node for node 0.0.3. So we will have a reachability table shown in Figure 4.5(b), and we can find that every ID node in a cycle is reachable by all the IDREF nodes in that cycle.

One thing to take note here is that, due to the cyclic references, the XML IDREF digraph will be transformed in to an infinite tree. But our tree model by virtual replication is still capable to handle it because the virtual replication only cares about the reachability relationship from an IDREF node to another ID node. Even the tree is an infinite tree, the reachability table is still finite. Because the number of nodes are finite and reachability is a binary relationship. So there are at most $n(n-1)$ reachability relationships given the number of nodes n.



(a) XML IDREF Digraph with Cyclic References

IDREF node	Reachable ID node
0.0.3	0.0
0.12.3	
0.0.3	0.12
0.12.3	

(b) Its Reachability Table

Figure 4.5: Constructing Reachability Table for Cyclic References

4.4.3 Reachability Table Space Complexity

Let the number of IDREF nodes in an XML IDREF digraph be L, where each IDREF node corresponds to one reference edge, then there could be at most L different ID nodes which are referenced by a reference edge. In the worst case, if every IDREF node can reach all these L ID nodes, then the space complexity is $O(L^2)$ in the worst case. The worst case only happens when all the ID/IDREF nodes forms a big cycle. Furthermore, the L IDREF nodes only occupy a small portion of all nodes in an XML IDREF digraph in practice (around 5% in real-life data set in our experiments in Section 4.8). This is because every IDREF node must belong to a particular object in the XML, and the attribute information of an object, like ID, name, etc., can only be described by non-IDREF nodes.

4.5 Further Extension and Optimization for Query Evaluation

In this section, we will further extend our transformed tree model to offer more features catering for different applications, like ranking module, etc.

4.5.1 Removing unnecessary checking of the reachability table

For query evaluation on our transformed XML tree model, we need to expand the inverted lists by checking the reachability table. However, we find that many of the checking is unnecessary. E.g., given the reachability table in Figure 4.3(b) and the following inverted lists to be expanded: “Alps” : 0.12.1 and “part” : 0.0.1, 0.0.2, ...

As discussed in Section 4.3.3, in step 2 we need to check the ancestor of each dewey label to see whether their ancestors appear in the *Reachable ID node* column of the reachability table. But the ancestors of 0.0.1, 0.0.2, ... do not appear in that column, thus the checking is in vain. To avoid unnecessary checking, we can add a check bit to each dewey label in the inverted list index, indicating whether we need to check such a dewey in the reachability table. E.g., the above inverted lists will become “Alps” : 0.12.1(true) and “part” : 0.0.1(false), 0.0.2(false), ...

Now we only need to check those dewey labels with the check bit being true. Here, only the ancestors of 0.12.1 will be checked in reachability table. Such a check bit can be computed during offline by checking whether the ancestors of each dewey label appear in the *Reachable ID node* column of the reachability table.

4.5.2 Adding Distance and Path to Reachability Table

Some of the XML tree keyword search methods need to rank the query results by some criteria. For example, one of the common criteria is the size of the results. It is usually measured by the sum of path length from the result root to each match node. To meet such a need, we can extend our virtual replication method (in Section 4.3.2) by adding a column called *distance* and a column called *path* to the reachability table. The *distance* value records the distance from the IDREF node to the reachable ID node. The *path* value records the path from the IDREF node to the reachable ID node. If an IDREF node can reach a reachable ID node by more than one paths, we record the distance/path of the shortest one. Because substructure with minimal size is in favor in both XML tree search and XML IDREF digraph search.

Take the reachability table in Figure 4.4(b) as an example. We can extend the table with a *distance* column and a *path* column. E.g., for the second tuple, the distance value is 3 because the IDREF node supplierRef:0.0.1.3 need to go through a path 0.0.1.3--→0.12→0.12.2--→0.88 to the reachable ID node employee:0.88. We can store such a path in the *path* column and its length in the *length* column. Therefore, the distance values for all the three tuples will be 1, 3 and 1 respectively.

With the *path* column and *length* column in the reachability table, we can find the path or path length from the answer root to a match node of some query keywords. The path consists of three parts: (1) path/distance from the answer root to the IDREF node; (2) path/distance from the IDREF node to the reachable ID node; (3) path/distance from the reachable ID node to the match node. The first and the third part can be found in the XML tree, the second part can be found in the reachability table *path*/*distance* column.

For example, given a result root part:0.0.1 and a match node eid:0.88.0 in Figure 4.4, the path/distance from the result root to the match node is the sum of three parts: (1) path/distance from part:0.0.1 to supplierRef:0.0.1.3 (length is 1); (2) path/distance from supplierRef:0.0.1.3 to employee:0.88 (length is 3), which can be found in the reachability table *path/distance* column; (3) path/distance from employee:0.88 to eid:0.88.0 (length is 1). Therefore the total path 0.0.1→0.0.1.3⇢0.12 →0.12.2⇢0.88→0.88.0 and the length is 5.

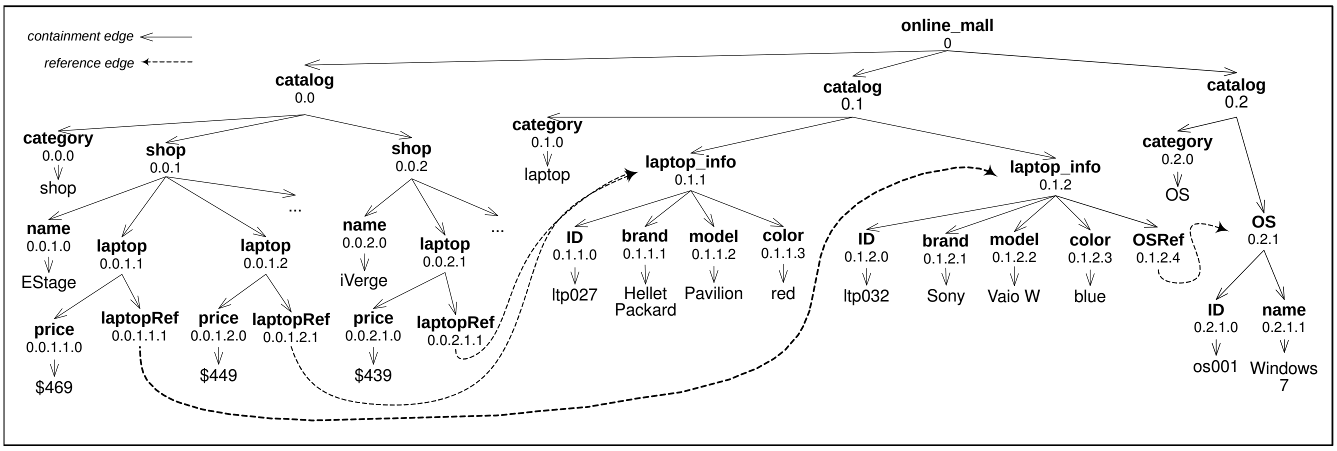


Figure 4.6: Sample XML Document with ID References

4.6 Solving the MisMatch Problem in XML IDREF Digraph

We have transformed the XML IDREF digraph to tree model such that we can adopt the efficient LCA-based XML tree search methods. Even though the keyword search technique over XML IDREF digraph is somehow different from the keyword search technique over XML tree, MisMatch problem could still exist. Because MisMatch problem exists in any form of information retrieval over data of any structure, as discussed in Chapter 3 Section 3.1.

Example 4.6. *Figure 4.6 is an XML document with ID references describing an online shopping mall, where the containment edges and reference edges are presented by solid line (→) and dashed line (⇢) respectively. Each shop sells some laptops. Each laptop node can have some IDREF nodes, i.e. laptopRef nodes, pointing to the laptop information, which can be reused by different shops to avoid duplication. Suppose a user wants to find the price of a laptop with model being Vaio W and red color, she may issue a query Q = {'Vaio','W','red','price'} over the data in Figure 4.6. But red color is unavailable for model Vaio W. Therefore, what will be returned is a list of mismatch results. One of the results is a subtree tree rooted at shop:0.0.1, with three* keyword match nodes*:*

0.1.2.2 for keyword 'Vaio' and 'W',

(following a path 0.0.1→0.0.1.1→0.0.1.1.1⇢0.1.2→0.1.2.2)

0.1.1.3 for keyword 'red',

(following a path 0.0.1→0.0.1.2→0.0.1.2.1⇢0.1.1→0.1.1.3)

0.0.1.1.0 for keyword 'price'.

(following a path 0.0.1→0.0.1.1→0.0.1.1.0)

As we can see, shop is returned because there is no laptop that can meet all the requirements. So MisMatch problem exists and mismatch results are returned.

In this section, we will talk about how we can apply our solution to the MisMatch problem dedicated for XML tree in Chapter 3 here to the transformed tree model. We adopt the LCA-based XML tree search methods on our transformed tree model and our MisMatch problem solution in Chapter 3 can be applied to any LCA-based keyword search methods. In this section we will make the necessary modification to the MisMatch solution for XML tree (as discussed in Section 3) to apply it onto the XML IDREF digraph.

Recall Chapter 3, the solution to the MisMatch problem is based on two novel concepts we proposed: *Target Node Type* (TNT) and *Distinguishability.* The process includes three main steps: 1) Detecting the MisMatch problem by calculating the Target Node Type; 2) Measuring the keywords importance based on the distinguishability; 3) Efficiently discovering approximate results, from which the suggested queries can be inferred.

4.6.1 Target Node Type for Detecting MisMatch Problem for XML IDREF Digraph

Given a result, the central idea to calculate the Target Node Type is getting the node type of each *keyword match node* and count their occurrences, as discussed in Chapter 3 Section 3.3. But in an XML IDREF digraph, there could be more than one paths from one node to another. Therefore, there could have more than one node types for a given node. This is different from XML tree. E.g., for the node $laptop\_info$:0.1.1 in Figure 4.6, there are two possible paths from the document root node to it. One is through ID reference and the other one is not through ID reference. So there are two different node types for that node, i.e., node type "online_mall/catalog/laptop_info" and "online_mall/catalog/shop/laptop/laptopRef/laptop_info". Therefore, to define the query result format, we need to specify which path it goes through from the answer root to each *keyword match node.*

So first of all, we will define the format of search result for XML IDREF digraph, which is slightly different from the format for XML tree in Defn. 3.4:

Definition 4.2. ***Query Result Format for XML IDREF Digraph.*** *For a keyword query* Q=$\{k_1, ..., k_n\}$ *issued on the XML data with ID references, we define the format of a query result* r *as:*

$$r = (v_{lca}, \left\{ m_1(path_1), m_2(path_2), ..., m_n(path_n) \right\})$$

where m_i is a keyword match node *w.r.t. keyword k_i ($i \in [1,n]$); v_{lca} is the root node of the result subtree, which connects to the* keyword match node *m_1 to m_n; $path_i$ is the path from v_{lca} to m_i.* $\square$

Comparing to Defn. 3.4, the only difference is that we need to specify the path from the result root to each *keyword match node*, as there could be more than one path from one node to another in an XML IDREF digraph. E.g. in Figure 4.6, there is multiple paths from node *onine_mall*:0 to node *laptop_info*:0.1.1, either through the reference edges or not through the reference edges. As discussed in Section 4.5.2, with the help of reachability table, the paths from the result root to each keyword match node can be recorded when we calculate the results. Therefore, the query results can be represented in the above format.

Secondly, to calculate the Target Node Type (TNT) of a result r, we need to get the node type of each *keyword match node* and count their occurrences. Given a result $r = (v_{lca}, \{m_1(path_1), m_2(path_2), ..., m_n(path_n)\})$, the node type of m_i consists of two parts: (1) path from document root to v_{lca}; (2) path from v_{lca} to m_i, i.e. $path_i$. We can just combine these two parts to get the node type of m_i.

After the node type for each *keyword match node* is ready, we can now calculate the TNT of a given result and detect the MisMatch problem in the same way in Section 3.3.

Example 4.7. *For a query $Q = \{$'Vaio','W','red','price'$\}$ issued in Figure 4.6, one of the results is*

$r = (0.0.1, \{$

0.1.2.2 *(0.0.1→0.0.1.1→0.0.1.1.1⇢0.1.2→0.1.2.2),*

0.1.2.2 *(0.0.1→0.0.1.1→0.0.1.1.1⇢0.1.2→0.1.2.2),*

0.1.1.3 *(0.0.1→0.0.1.2→0.0.1.2.1⇢0.1.1→0.1.1.3),*

0.0.1.1.0 (0.0.1→0.0.1.1→0.0.1.1.0)$\})$.

The node types of these keyword match nodes *are ('Vaio' and 'W' match the same node):*

0.1.2.2: $\{online\_mall/catalog/shop/laptop/laptopRef/laptop\_info/model\}$ *(denoted as* t_1*)*

0.1.1.3*:* $\{online\_mall/catalog/shop/laptop/laptopRef/laptop\_info/color\}$ *(denoted as* t_2*)*

0.0.1.1.0: $\{online\_mall/catalog/shop/laptop/price\}$ *(denoted as* t_3*).*

The set of distinct node types $\mathbb{T} = \{t_1, t_2, t_3\}$, *where* $count(t_1) = 1$, $count(t_2) = 1$ *and* $count(t_3) = 1$.

Then we check the prefixes of all node types in $\mathbb{T}$. *The lowest one is* $t =$ "*online_mall/catalog/shop/laptop*". *Suppose we have the following constraints (either by examining the XML schema or scanning the XML document):* $t.maxContain(t_1) = 1 \geq count(t_1) = 1$, $t.maxContain(t_2) = 1 \geq count(t_2)$ *and* $t.maxContain(t_3) = 1 \geq count(t_3)$.

Therefore, by Defn. 3.5, $TNT(r) = t =$ "*online_mall/catalog/shop/laptop*" *even though no laptop can meet all the user's requirements at data level.*

4.6.2 Distinguishability for Measuring Keywords' Importance

Recall Chapter 3 Section 3.3 that we have proposed the concept of *distinguishability* to measure the importance of the query keywords contained in a certain *keyword match node* in XML tree model. However, in an XML IDREF digraph, the reference edges can make the structure a bit complex. There could be sequential references (a node a references a node b, and then a descendent of b also references a third node c) and cyclic references (containment edges and references edge form a cycle). Then there can be exponentially many node types. This poses challenges for directly adopting the distinguishability formula (i.e. Equation 3.1 in Chapter 3). But we also notice that many node types in an XML IDREF digraph are actually representing the same type of information. Therefore, in computing the distinguishability in XML digraph, we propose to exploit the node types to replace each other when they are representing the same type of information.

Recall Equation 3.1 in Chapter 3, distinguishability $D(\mathbb{K}, t)$ measures the importance of the query keywords $\mathbb{K}$ when $\mathbb{K}$ match a node of type t. It is defined based on two variables f_t and $f_t^{\mathbb{K}}$. f_t is the number of nodes of type t; $f_t^{\mathbb{K}}$ is the number of nodes which are of node type t and subtree-contain each keyword in $\mathbb{K}$.

In Chapter 3, we store a f_t value for each distinct node type t; to calculate $f_t^{\mathbb{K}}$, we build the following inverted index: for each combination of a distinct node type t and a distinct keyword k (in the XML data), we build an inverted list containing all nodes of type t where each node *subtree-contains* keyword k. As a result, $f_t^{\mathbb{K}}$ can be computed by simply computing the intersection of the inverted lists for each keyword in $\mathbb{K}$ under node type t.

Here for the XML IDREF digraph, it will be good if we could build the same index. However, there could be exponentially many node types in the XML IDREF digraph. So it is not feasible to store a f_t value for each distinct node type t and build an inverted index for each combination of a distinct node type t and a distinct keyword k.

Comparing the node types in an XML tree to those in an XML IDREF digraph, we notice that many node types in an XML IDREF digraph are actually representing the same type of information. For example in Figure 4.6, the node type t_a= "online_mall/catalog/laptop_info" and t_b="online_mall/catalog/shop/laptop/laptopRef/laptop_info" are actually representing the same type of information, i.e. laptop_info. If we extract the schema graph of the XML document, as shown in Figure 4.7, it will be clearer that these two node types actually represent the same type of information, i.e. they correspond to the same schema node in the schema graph. Node type t_a contains no ID reference edge in its path while t_b contains ID reference edge in its path.

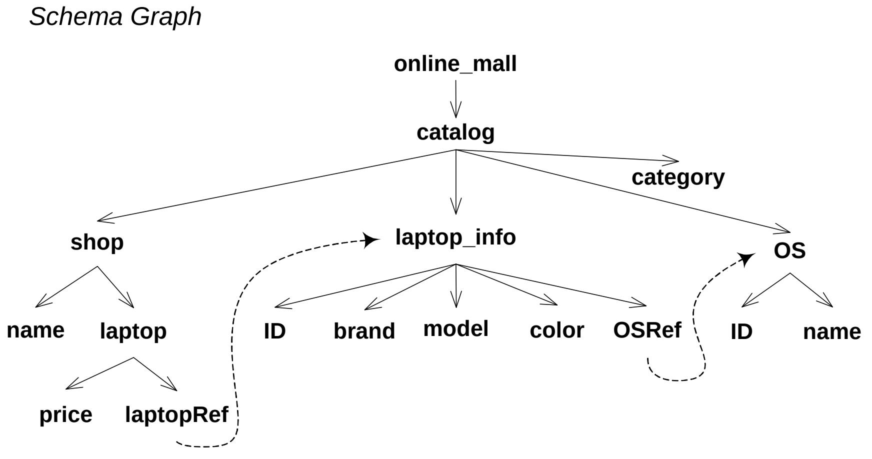


Figure 4.7: Schema Graph of Figure 4.6

Definition 4.3. ***Solid & Virtual Node Type*** *We call a node type which does not contain ID reference edges in its path as* ***solid node type****; a node type which contains ID reference edges in its path as* ***virtual node type****.*

Since *solid node types* do not include ID reference edges, the number of *solid node types* equals to the number of schema nodes in the schema graph. For example in Figure 4.7, the number of *solid node types* is 17 as there are 17 schema nodes in the schema graph, while the number of *virtual node types* can be exponentially many. But it is easy to know from the schema graph that, every *virtual node type* corresponds to a *solid node type*, i.e. they correspond to the same schema node in the schema graph.

Therefore, to calculate the distinguishability, a feasible solution is to use the distinguishability for a *solid node type* to simulate the distinguishability for a *virtual* *node type*, if they are representing the same type of information. Let *solid*(t) be the corresponding *solid node type* for a *virtual node type* t, then we can define the distinguishability as follows:

$$D(\mathbb{K}, t) = \begin{cases} 1 - \frac{f_t^{\mathbb{K}}}{f_t} + \frac{1}{f_t}, & \text{if } t \text{ is a } \textit{solid node type}. \\ D(\mathbb{K}, solid(t)), & \text{if } t \text{ is a } \textit{virtual node type}. \end{cases} \tag{4.1}$$

If a node type t is a *solid node type*, we define it the same way as Equation 3.1; if a node type t is a *virtual node type*, we use the distinguishability for *solid*(t) to simulate its distinguishability value.

So now we can store a f_t value for each distinct *solid node type* and build an inverted index for each combination of a distinct *solid node type* t and a distinct keyword k. Then distinguishability can calculated based on such indexes in the same way as discussed in Chapter 3 Section 3.6.

4.6.3 exLabel for Efficient Approximate Results Detection for XML IDREF Digraph

With distinguishability, we know which keywords are important. The next step is to find the approximate results which contain those important keywords and also have replacement for the less important keywords.

In Chapter 3, the approximate results are found based on exLabel assigned to each node in the XML document, which is discussed in Chapter 3 Section 3.5. The exLabel is bitmap-based. Every node n's exLabel records what types of nodes are subtree-contained by n. Each bit of the exLabel corresponds to a particular node type and the value of the bit indicates whether such a node type appear in the subtree rooted at n.

Similar to the case of calculating distinguishability, there could be exponentially many node types in an XML IDREF digraph (including *solid node types* and *virtual node types*). Every node n in the data could have exponentially many node types appearing in the subtree rooted at n. Therefore, it is not a feasible solution to record all node types in n's exLabel.

Actually the purpose of exLabel is: we want to check the exLabel of a node n to see whether n can be an approximate result. If the exLabel shows that node n subtree-contains replacement for a particular type of nodes, then it could be an approximate result. In XML IDREF digraph, every *virtual node type* corresponds to a *solid node type* representing the same type of information. E.g. in Figure 4.4, the *solid node type* t_a= /company/supplier and the *virtual node type*t_b=/company/project/part/supplierRef/supplier are actually representing the same type of information. There are relatively small number of *solid node types* in an XML IDREF digraph.

So a feasible solution is to use the *solid node types* to represent the *virtual node types* in the exLabel (which is actually a bitmap). For example, if a node n subtree-contains some nodes of *virtual node type* t, then we can set the bit for node type $solid(t)$ in the exLabel because $solid(t)$ represent the same type of information as t. Then the maximum size of an exLabel (number of bits) equals to the number of different *solid node types* in the XML IDREF digraph.

Then the approximate results can be efficiently found based on the exLabel, as discussed in Chapter 3 Section 3.5. Each approximate result subtree-contains replacement nodes for the *keyword match nodes* containing the less important keywords, where such replacement nodes and the *keyword match nodes* represent the same type of information. So the suggested queries can be inferred by replacing those less important keywords with the keywords in the replacement nodes.

One thing to take note here is that, to make sure each suggested query does not have MisMatch problem itself, we need to check whether the TNT of the approximate result is the same as its root's node type before we return the suggested query to users.

Algorithm 4: transformXMLGraphToTree(XT)

```
input  : XML IDREF Digraph $XG$
output : Transformed XML Tree $XT$ and reachability table $RT$
1  // Construct reachability table
2  Table $RT$;
3  assignDeweyLabel($XG$); //regardless of reference edges
4  foreach IDREF node $n \in XG$ do
5      $np$ = the ID node which $n$ references to;
6      Queue $nodesToExplore$={$np$};
7      Set $exploredNodes$ = {$\emptyset$};
8      while $nodesToExplore \neq \emptyset$ do
9          $v$ = $nodesToExplore$.removeFirst();
10         if $exploredNodes.notContains(v)$ then
11             $exploredNodes = exploredNodes \cup v$;
12             foreach IDREF node $m \in$ the subtree rooted at $v$ do
13                 $mp$ = the node which $m$ references to;
14                 $nodesToExplore$.add($mp$);
15     foreach node $r \in exploredNodes$ do
16         $RT$.addTuple($n$.dewey, $r$.dewey);
17 $XT$ = removeAllReferenceEdges($XG$); // Generate the XML tree
18 return $XT$ and $RT$;
```

4.7 Algorithms

In this section, we present Algorithm 4 to transform an XML IDREF digraph to our tree model, which consists of an XML tree and a reachability table. After that, we will present Algorithm 5 for solving the MisMatch problem in XML IDREF digraph.

Given an XML IDREF digraph, the XML tree part can be easily generated by removing all the reference edges (line 17). The main task here is to generate the reachability table. We will first assign a dewey label to each node in the XML IDREF digraph (line 3). Then based on each IDREF node n in the XML IDREF digraph (line 4), we do a breadth-first search to explore the reachable ID nodes until no more new ID node can be further explored (line 5-14). The first node to be explored is the ID node being referenced by n and it will be pushed to a queue (line 5-6). The ID nodes which have been visited will be stored in a set (line 7). Each time we will take a node from the queue to explore until there is no more node in the queue (line 8-9). If the node taken from the queue is not visited before (line 10), we will visit it and mark it as explored (line 11). Then we will further explore within the node. For each IDREF node within it (line 12), we will add the corresponding ID node to the queue (line 13-14), which stores the nodes waiting to be explored. This process will terminate until no more node to explore (line 8). Finally, it will add all reachable ID nodes to the reachability table (line 15-16).

For the algorithm of doing query evaluation based on our tree model, it is similar to the 3 steps discussed in Section 4.3.3 and existing XML tree search algorithms can be easily found in the literature [64, 99, 100]. So the pseudo code will be omitted here.

Next we will present Algorithm 5 for solving the MisMatch problem for XML IDREF digraph. The input is the query Q, its retrieved results $\mathbb{R}$ and the reachability table RT. First it checks each result of Q (line 3) and infer its TNT (line 4-12). This is the major difference comparing to Algorithm 1 in Chapter 3 Section 3.6. For each *keyword match node* (line 6), if the node is not an expanded node, as discussed in Step 2 in Section 4.3.3 (line 7), then the node type of node n is the path from the document root to n (line 8). If the node is an expanded node, then the node type of n consists of 3 parts: path from the document root to the IDREF node; 2) the path from the IDREF node to the ID node, which is recorded in the reachability table; 3) the path from the ID node to n (line 10). After the node types of the *keyword match nodes* are ready, the TNT can be calculated by checking the conditions discussed in Definition 3.5 in Chapter 3 Section 3.3. Once one of the results does not miss the target, which means what the user wants is in the retrieved results, it will terminate the process (line 14). Otherwise, it will generate the suggested queries for each query result as discussed in Chapter 3 Section 3.5.2 (line 16-17).

Algorithm 5: MisMatchResolver(Q,$\mathbb{R}$,RT)

```
   input  : user query $Q$=$keywords[m]$, $Q$'s results $\mathbb{R}$ and reachability table $RT$
   output : null if no MisMatch problem; $suggestedQueries$ + one sample result $n$
            for each $Q' \in suggestedQueries$ otherwise
1  suggestedQueries $\leftarrow \emptyset$;
2  {Detector}
3  foreach $r \in \mathbb{R}$ do
4      {infer TNT of $r$}
5      nodeTypes $\leftarrow \emptyset$;
6      foreach $n \in r.matchnodes$ do
7          if $n.iExpandedNode = false$ then
8              $type$ = pathFromRootTo($n$);
9          else
10             $type$ =
               pathFromRootTo($n.idrefNode$)+$RT$.getRecordedPath($n.idrefNode$,
               $n.idNode$)+solidPathFromTo($n.idNode$, $n$);
11         $nodeTypes$.add($type$);
12     tnt = getTntByCheckingConditions($nodeTypes$);
13     if $r.v_{lca}.type = tnt$ then
14         return $null$;
15 {Suggester}
16 foreach $r \in \mathbb{R}$ do
17     {This part of pseudo code to get suggested query $suggestedQueries$ is the
       same as Algorithm 1 in Chapter 3 Section 3.6.}
18 return $suggestedQueries.sort()$;
```

4.8 Experiments

In this section, we will first present the experimental results comparing our approach with two graph-search-based methods. One is XKeyword [37], which is dedicated for XML IDREF digraph by making use of the XML schema. Another one is BLINKS [35], which is one of the most efficient pure digraph search method so far by building a bi-level index. After that, we will present the experimental study of the MisMatch solution working on XML IDREF digraph.

Experimental Settings. All algorithms are implemented in Java. The experiments were performed on a 2.83GHz Core 2 Quad machine with 3GB RAM running 32-bit windows 7. Berkeley DB Java Edition [1] is used to organize our reachability table in a B+ tree and store the inverted lists. MySQL [75] is used to support XKeyword. BLINKS does not need any database support since it is an in-memory approach.

Data Set. To test the real impact of the keyword search methods, we use a 200MB subset of real-life XML data set with ID/IDREF, ACMDL [1] , in our experiments. It contains publications from 1990 to 2001 indexed by the ACM Digital Library. There are 38K publications and 253K citation (as IDREF) among the publications. Totally 4.5M XML nodes and 4.8M XML edges are included. We can see that IDREF nodes (253K) are 5% of all XML nodes.

4.8.1 Keyword Search on XML IDREF Digraph

Comparing The Results

There are abundant search methods available designed for XML tree. Here we adopt one of them, i.e. ELCA [100], to work on our transformed tree model. Most

[1]Thanks to Craig Rodkin at ACM Headquarters for providing the ACMDL dataset.

Table 4.1: Result Overlap Between Our Approach and Graph Methods

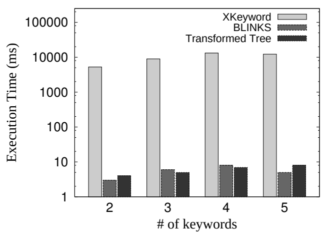
Graph Methods	XKeyword				BLINKS			
# keywords	2	3	4	5	2	3	4	5
Average Result Overlap	77.9%	83.0%	85.4%	82.9%	92.1%	89.0%	90.5%	91.2%

of the XML tree methods focus on finding a meaningful subset of all possible results with regard to users' search intention. However, studying whether these subset of results are meaningful regarding users' search intention is not the main focus of this chapter. So here we study the similarity between the subset found by tree methods and the subset found by graph methods in terms of result overlap rather than users' search intention.

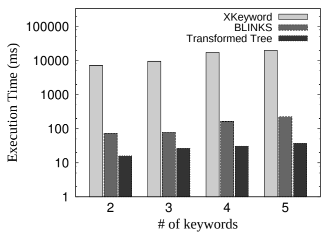
We generate 100 random queries with two keywords, three keywords, four keywords and five keywords respectively. For each group of queries, we compare the top-20 results found by XKeyword and BLINKS on the original XML IDREF digraph, to the top-20 results found by ELCA on our transformed tree model. For a fair comparison, all results are ranked by the size of the corresponding Minimal Steiner Tree, i.e. the sum of the path length.

Table 4.1 shows the average result overlap between our approach and the graph methods, which is calculated as (*# of same results in top-20)/20.* Two results are the same only if the root and each match node are the same.

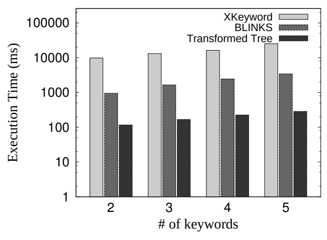
As we can tell from Table 4.1, averagely 16 out of top-20 results are the same between our approach and XKeyword, while averagely 18 out of top-20 results are the same between our approach and BLINKS. Because XKeyword sets a maximum result size to constraint the search space, sometimes it finds less than 20 results. Therefore the result overlap is smaller.



(a) keyword frequency 10

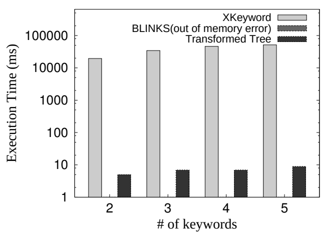


(b) keyword frequency 100

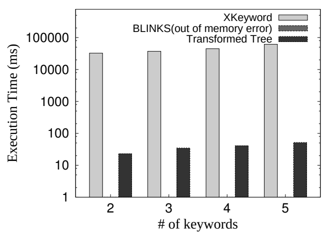


(c) keyword frequency 1000

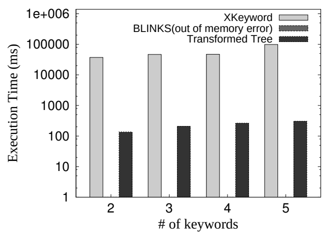
Figure 4.8: Query Execution Time (45MB data Size)



(a) keyword frequency 10



(b) keyword frequency 100



(c) keyword frequency 1000

Figure 4.9: Query Execution Time (200MB Data Size)

Performance

Next we will study the performance of our approach with the transformed tree model. XML tree search methods can be very efficient. E.g., ELCA can compute the results by linearly scanning the inverted lists. Here we will compare our approach with two digraph search methods, XKeyword and BLINKS. However, BLINKS is an in-memory approach, which throws out-of-memory errors when handling the ACMDL date set. In order to be able to compare the performance of these three approaches, we have to downgrade the data set size to 45MB, which is the maximum data size BLINKS can handle on our machine [2]. Later we will compare on the full data set with only our approach and XKeyword.

Figure 4.8 shows the execution time of the three approaches. Our approach performs a full ELCA computation while XKeyword and BLINKS perform a top-

[2]BLINKS throws out-of-memory error when the data set size is larger than 45MB for ACMDL. A recent survey [22] also has a similar conclusion.

20 results computation. We generate 100 random queries for each combination of *keyword frequency* and *# of keywords*. We can see that our approach outperforms XKeyword by orders of magnitude. This is because XKeyword stores the node information in relational tables to accommodate very large graphs, then the results computation is based on table join. Although schema information can help prune some search space, it is still not efficient.

For BLINKS, our approach is faster than it by an order of magnitude when keyword frequency is 100 or 1000. But our approach runs neck and neck with BLINKS when the frequency is around 10. We find that this is because BLINKS is an in-memory approach, which loads the whole graph into memory and does not need to access disk during the whole query evaluation. Yet it is not scalable to large data set. With 1.5 GB heap size assigned to JVM on our machine, 45MB is the maximum data size it can handle without out of memory. A recent survey [22] also has a similar conclusion. For our approach, we store the inverted lists and reachability table in database, so the disk access dominates the query evaluation time when keyword frequency is low.

Now we will compare XKeyword and our approach on the full data set. Figure 4.9 shows the experiment results. As we can see, our approach is still orders of magnitude faster than XKeyword on full data set. Comparing Figure 4.9 to Figure 4.8, we find that XKeyword consumes more time even the keyword frequency in a query is the same. This is because XKeyword is based on table join. Larger data set will lead to larger tables. Therefore XKeyword requires more time to join the tables for results regardless of keyword frequency.

4.8.2 MisMatch Solution on XML IDREF Digraph

In this section, to verify the effectiveness and efficiency of the MisMatch solution working on XML IDREF digraph, we have conducted a set of experiments that are similar to the experiments in Chapter 3 Section 3.7. For expository convenience, we refer to our Extended MisMatch Detector & Suggester as *Extended MisMatch D&S*.

Query Set. Our query set contains 18 queries for the dataset, all of which are collected from the real-world user log data of our system. 10 sample queries and their best-3 suggested queries (if any) are shown in Table 4.2. For better understanding of the queries, the schema tree of the ACMDL dataset is given in Figure 4.10. Besides, 1000 random queries are generated as well (see Section 4.8.2).

User Study Methodology and Ground Truth. We employ 15 assessors to pick up the queries with the MisMatch problem, and their judgements are based on both the queries given and their respective results[3]. We obtained the ground truth by judging a query to have the MisMatch problem if at least 8 of the 15 assessors agree on that. Eventually, 9 out of the 18 queries have the MisMatch problem.

For ACMDL dataset (with ID reference), some of the suggested queries are found involving ID references while some of them are found without involving ID references. E.g. for Q_{A1} in Table 4.2, according to the dataset, Jeffrey Ullman did not publish any paper at INFOCOMM or reference any INFOCOMM paper in his paper. The results being returned are all mismatch results. The suggested options PODS and SIGMOD are found without involving ID references, which are the conferences Jeffrey Ullman has published papers at. Another suggested option KDD is found involving ID references. KDD is suggested because some KDD papers

[3]Since different users could have many different search intentions even for the same query, we do not want to confine the search intentions to some pre-defined options. So we did not show any pre-defined search intentions to users for reference or let them choose, which could affect users' decision.

Table 4.2: 10 of the Sample Queries on ACMDL

ACMDL (with ID references)			
#	Query	suggested queries	best-3 suggested queries (Format: explanation → suggested options)
Q_{A1}	Jeffrey Ullman INFOCOMM	207	(proceeding): INFOCOMM → PODS / SIGMOD / KDD
Q_{A2}	Ling Tok Wang KDD 1993	112	(year): 1993 → 2000 (proceeding): KDD → SAC / ACM Transaction on Database Systems
Q_{A3}	Michael Stonebraker PODS	285	(proceeding): PODS → CHI / OOPSLA / SIGMOD
Q_{A4}	Victor Vianu PODS 1999	89	(year): 1999 → 2000 / 1998 / 1997
Q_{A5}	Hanspeter Pfister database	971	(title): database→ Integrated volume compression and visualization / The VolumePro real-time ray-casting system / VolVis a diversified volume visualization system
Q_{A6}	Michael Franklin 2000	0	None
Q_{A7}	Tan Kian-lee robot	229	(title): robot → A framework for modeling buffer replacement strategies / Sampling from databases using B+-trees / Rule-assisted prefetching in Web-server caching
Q_{A8}	SIGIR England 1985	992	(country): England → Canada (year): 1985 → 1984 / 1980
Q_{A9}	David Dewitt skyline	68	(title): skyline → The 007 Benchmark / A status report on the OO7 OODBMS benchmarking effort / Crash recovery in client-server EXODUS
Q_{A10}	Tan Kian-lee Michael Franklin	52	(author): Michael Franklin → Chua Tat-Seng / Ooi Beng-Chin / Li Jiandong

are referenced in Jeffrey's papers and such paper reference relationship is expressed by XML reference edges. However, such information is not reflected simply by the suggested queries. This is why our MisMatch Module also returns a sample result for each suggested query to help users understand our suggestion.

Evaluation Method

We select the queries with the MisMatch problem for each dataset to conduct a user study.

To conduct a fair evaluation, we are aware of two things. *First*, we invite both experts and novices to participate the task of scoring the suggested query. We ask three CS research students and three undergraduates in other faculties. The participants are shown the matching results of each query, the best-5 suggested queries together with the corresponding sample query results. *Second*, the participants are asked to score the quality of each suggested query by using the Cumulated Gain-

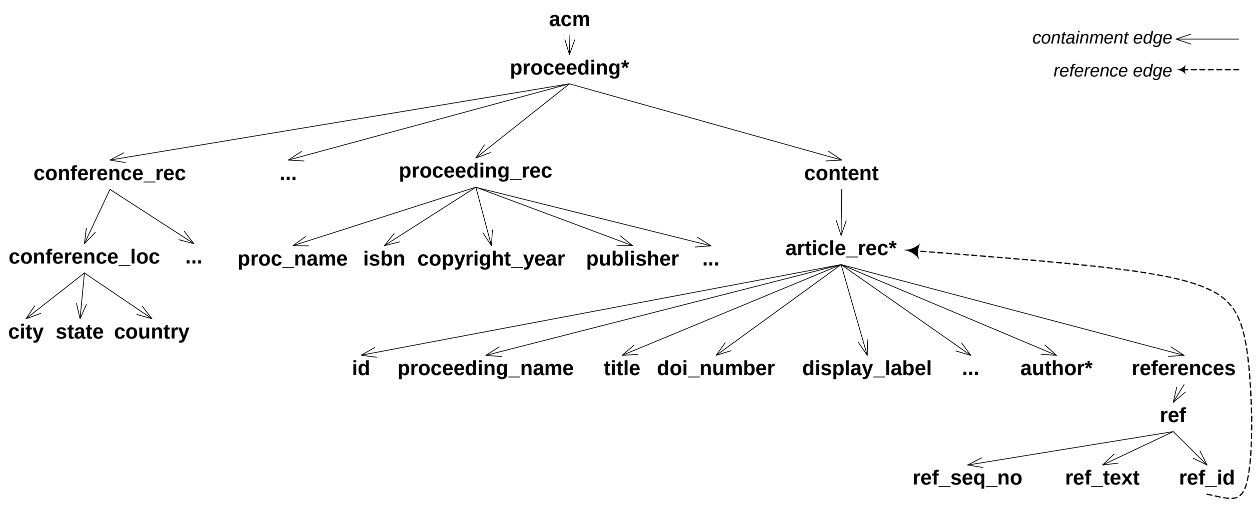


Figure 4.10: Schema Graph of ACMDL Dataset (some parts are omitted because full schema graph is too big to display)

based evaluation (CG) metric [40] (from 0 to 5 points, 5 means best while 0 means worst). In contrast to traditional metrics like precision and recall which adopt a binary judgement (yes or no), CG is aware of the fact that all results are not of equal relevance to user.

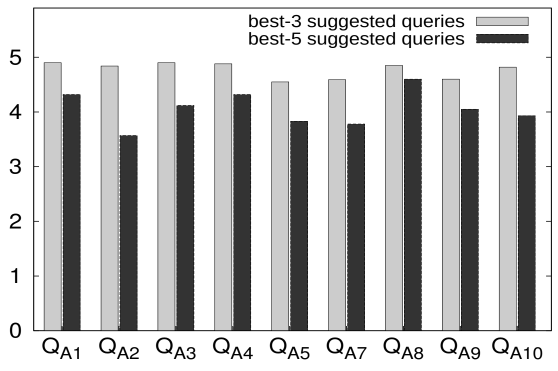


Figure 4.11: Average Quality Measure of Suggested Queries

Evaluation of Overall Quality

The average scores for best-3 and best-5 suggestions are shown in Figure 4.11. We can find for queries with the MisMatch problem, our approach is able to find reasonable suggested queries for them, and subsequently it leads to more meaningful results; the scores for best-3 suggestions are always higher than those of best-5, which also shows the effect of our query ranking scheme.

Most likely, the best-3 suggested queries will be viewed by the struggling users. So in the rest of the paper, when we talk about the quality of the suggested queries, we mean the average score of the best-3 suggested queries.

Study of the query ranking scheme and distinguishability threshold

Table 4.3: Suggestion Quality w.r.t. different τ and ranking factors

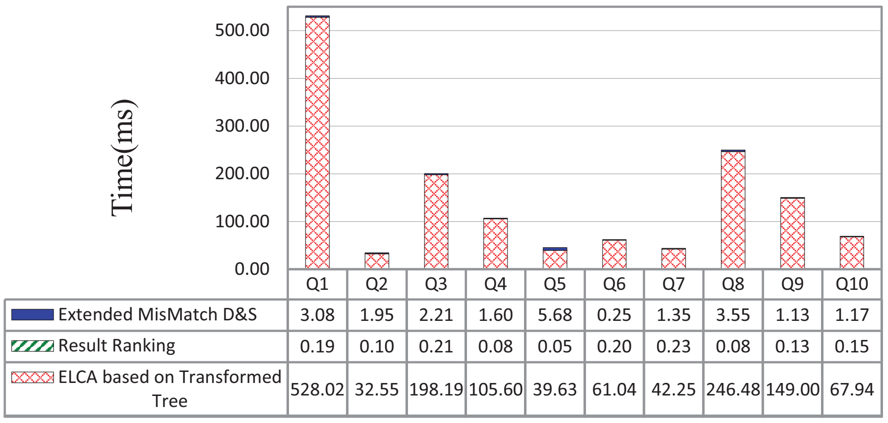
	τ	all ranking factors	no cn	no dt	no $\sum D$
ACMDL	0.9	4.61	4.32	4.35	4.10
	0.6	4.61	4.32	4.35	4.10
	0.3	4.61	4.32	4.35	4.10
	0.0	4.61	4.32	4.35	4.10

We further study how the proposed ranking factors for ranking suggested query affect the overall quality of suggested queries. The ranking factors include cn, dt and $\sum D$, as discussed in Chapter 3 Section 3.4.3. The scores for the suggested queries of each case are shown in Table 4.3. Please ignore the choice of τ for the time being. By comparing the scores in a columnwise way, we find:
(1) The model taking all ranking factors always outperforms any models that miss one of the three ranking factors.
(2) Without considering the distinguishability of the keywords to be replaced (i.e., $\sum D$), the suggested query quality decreases more than the case without any of the other two factors. It shows that distinguishability plays an important role.

By comparing the scores in a rowwise way, we can see that the best suggested queries usually do not change even when we set a smaller threshold τ. It is because we have already found the best suggested queries when we set a high τ, since preserving the keywords with high distinguishability is more reasonable as discussed in Chapter 3 Section 3.4.

Efficiency

For each query in Table 4.2, we run our algorithm 10 times and collect the average processing time on hot cache, as shown in Figure 4.12. The query result ranking time is too small to display. Moreover, we record the time used by the *Extended MisMatch D&S*.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
■ Extended MisMatch D&S	3.08	1.95	2.21	1.60	5.68	0.25	1.35	3.55	1.13	1.17
▨ Result Ranking	0.19	0.10	0.21	0.08	0.05	0.20	0.23	0.08	0.13	0.15
⊠ ELCA based on Transformed Tree	528.02	32.55	198.19	105.60	39.63	61.04	42.25	246.48	149.00	67.94

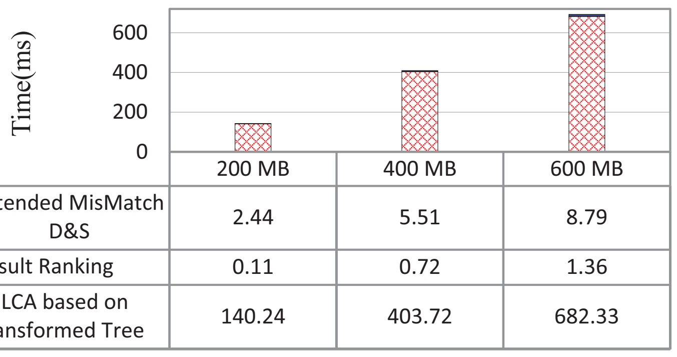
Figure 4.12: Processing Time for some Sample Queries (The result ranking time is too small to display.)

The *Extended MisMatch D&S* only takes a small portion of the whole query processing time, as shown in Figure 4.12. The processing time of the *Extended MisMatch D&S* for all ten queries are less than 10ms. So it could be too small to display in Figure 4.12.

Scalability

Sample Queries

Firstly, we conduct our scalability test by studying the impact of increasing data size on the Extended MisMatch D&S. We run the queries on ACMDL with different sizes. Figure 4.13 shows the average processing time of one query on the datasets. The processing time of the Extended MisMatch D&S increases linearly w.r.t. the data size. Because larger data size leads to possibly larger number of results, and our Extended MisMatch D&S needs to check all results to decide the MisMatch existence and find suggestions based on each result.



	200 MB	400 MB	600 MB
■ Extended MisMatch D&S	2.44	5.51	8.79
▨ Result Ranking	0.11	0.72	1.36
▩ ELCA based on Transformed Tree	140.24	403.72	682.33

Query Processing Time v.s. Data Size

Figure 4.13: Impact of Data Size.

Random Queries Besides the real-world sample queries, we further study the performance of our Extended MisMatch D&S over random queries. Keywords ACMDL datasets are randomly picked to form queries of length 2∼5 and those with MisMatch problem will be kept. We record the first 1000 of such queries and count the suggested queries output by our Extended MisMatch D&S. The distribution of these queries with different ranges for the number of suggestions is shown in Figure 4.14(a), from which we find most queries will result in suggested queries no larger than 500. Similar to our findings on sample queries, Figure 4.14(b) reports the linear relationship between the Extended MisMatch D&S processing time and the number of suggested queries on random queries.

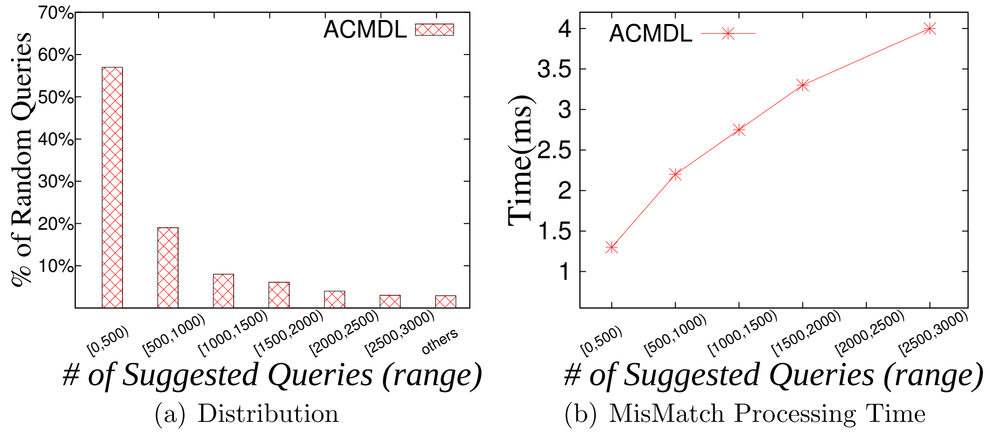


Figure 4.14: Scalability Test of Random Queries

4.9 Conclusion

In this chapter, we observed that an XML IDREF digraph is mainly a tree structure with a portion of reference edges. It motivated us to proposed a novel method to transform an XML IDREF digraph with ID/IDREF to a tree model, such that we can exploit abundant efficient XML tree search methods and our mismatch solution for XML tree can be applied to XML IDREF digraph as well. Existing keyword search methods for XML IDREF digraph compute the query results by graph traversal. Then the keyword search problem on an XML IDREF digraph is reduced to the problem of finding Minimal Steiner Tree (MST) or its variants in a digraph, where an MST is defined as a minimal subtree containing all query keywords in either its leaves or root. Since finding all MSTs in a digraph is an NP-complete problem, efficiency is one of the notable issues.

We proposed to transform an XML IDREF digraph to a tree model by virtually replicating the subtrees being referenced. Our tree model consists of two parts: an XML tree and a table (called reachability table), which is capable of handling different kinds of reference patterns in an XML IDREF digraph. Based on the reachability table, we designed a query evaluation framework on our tree model which includes an extra step than traditional LCA-based XML tree search methods, i.e. expanding the inverted lists with those nodes within the duplicated subtrees. This is done by checking the reachability table. Our approach can work with any existing XML tree search method. The experimental results show that our approach is orders of magnitude faster than the traditional search methods on XML IDREF digraph while generating a similar set of results as existing XML IDREF digraph search methods. Since the traditional search methods on XML IDREF digraph find the results by graph traversal, it could be as inefficient as NP-complete. After we have transformed the XML IDREF digraph to tree model and adopted the XML tree search methods, we further applied our mismatch solution dedicated for XML tree onto the transformed XML tree model. The key steps of the mismatch solution, i.e. MisMatch problem detection and suggestion generation, can be applied here as they can work with any LCA-based XML keyword search methods.

CHAPTER 5

QUERY RESULT PRESENTATION OF XML KEYWORD SEARCH

5.1 Introduction

As we discussed in Chapter 3 and Chapter 4, keyword search is an important way for novice to explore XML documents. It has attracted a lot of research efforts on how to retrieve the proper results for a keyword query. After the results are retrieved from the search engine, the results need to be presented to users, which is an unavoidable yet important topic for XML keyword search, as well as any other form of information retrieval. To further reduce the gap between users' search intention and the query results, how to present the query results in a proper way plays an important part. As discussed in Chapter 2 Section 2.5, different forms of data and query results require different ways for optimal visualization. Existing visualization techniques for other forms of data, like relational data, web data, etc.,

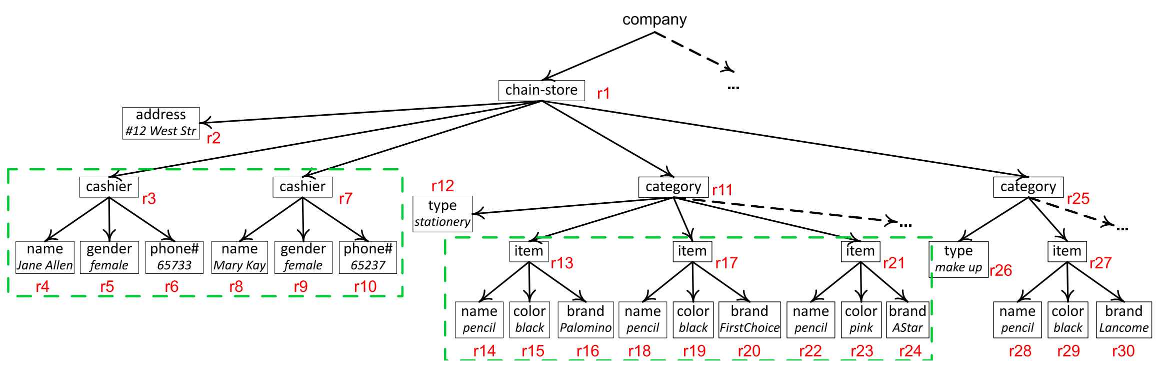


Figure 5.1: Sample XML Document about the Chain-stores in a Company

will not work well on XML as none of them consider the tree structure in XML. So far there is very few works on result visualization for XML keyword search. In this chapter, we will study the query result presentation problem mainly on XML data without ID references. For such a problem on XML data with ID references, it will be one of our future work.

For XML data without ID references, the traditional way on result presentation is to return a list of subtrees as query results. We find that it is insufficient to meet the information needs of users. Such a presentation is imprecise and could be misleading. Next we will see some motivations, based on which we propose a new exploration model on XML database as a complementary component of the XML keyword search engine. It is designed to enhance users' search experience in XML keyword search and bridge the mismatch gap between users' search intention and the query results.

Motivation 1 All the data in an XML tree is inter-connected by the hierarchical structure. Therefore, each query result of XML keyword search is a part of the XML data tree rather than a piece of independent information. Among the query results(subtrees), they may have sibling and containment relationships. Without showing such relationships, the results are imprecise and could be misleading.

Example 5.1. *An XML data tree in Figure 5.1 describes the chain-store information of a company, including information like address, cashier, item etc. Each node in the figure is assigned a unique ID, which will be used in the rest of the chapter. For the XML tree in Figure 5.1, a query "pencil black" will get the following results by LCA:*

1. *subtree rooted at node* $r13$*(item),*
2. *subtree rooted at node* $r17$*(item),*
3. *subtree rooted at node* $r27$*(item).*

Without showing the relationships among the results, it is hard to know that the third result is completely different from the first two results because it is a make-up pencil rather than a normal pencil. It is imprecise and misleading such that users will understand the query results wrongly. As a comparison, if we display the results in the context of the whole XML data tree, this can be easily identified by the user. □

Motivation 1 leads to a demand of showing the relationships among the query results and the context of the query results.

Motivation 2 Users' interaction with a search engine is not a one-time transaction. Many users will issue a serious of queries progressively to further explore the returning query results [86]. Therefore, users need some "means" to make adjustment to the query results in order to further explore what they want. So far, the only means for users to adjust the query results is by changing and re-submitting the keyword query .

Example 5.2. *For a query "Allen female" issued in Figure 5.1, whose search intention is somehow ambiguous, one of the results by LCA is the leftmost subtree* *rooted at node r3(cashier). It meets the information needs for those who are finding information of a cashier.*

Since this is a database for chain-stores, some people may want to go further to find the chain-store with cashier Allen. Then the current results cannot meet their information needs. He may need to take further action. But the crucial thing is that, so far there is only one way for a user to take further action to find what she wants: to change the keywords and re-submit the query. □

However, to change the keywords and re-submit the query is not trivial. For the case in Example 5.2, users who are searching for chain-stores need to add more keywords to make their search intention more specific. Since the user may not know much about the database, she may not know what keywords to add in. There are a lot of choices: "store", "shop", "chain-store", etc. But we can observe that, what the user wants, i.e. the information of a chain-store, is just above the subtree of the cashier. To reach what she wants, the user only needs to adjust the query result to a higher level by some means (if there is any) rather than revising and re-submitting the query.

Therefore, Motivation 2 calls for an easier way for users to further explore the query results to find what they want.

Based on Motivation 1 and 2, we propose our Map-Like Exploration Model on XML Database (XMAP), which can work as a complementary component of the XML keyword search engine or even work independently as a new way to explore XML data. XMAP displays the XML data tree as a whole to users. Users can zoom in/out the display and move left/right/up/down the display. The query results will be highlighted on the global display. Therefore, it provides users an easy way to interact with XML data addressing Motivation 1 and 2.

XMAP explores the XML data tree following a similar way in which we explore a digital map, such as Google Maps [3], Yahoo Maps [7], etc. When users search on a digital map, besides a list of results being returned, an interactive map is also available to help users consume and adjust the query results. The interactive map can be either a component of a geographic data search engine or even as an independent model to explore geographic data.

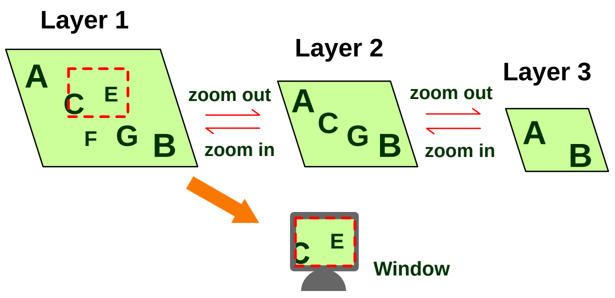


Figure 5.2: Working of A Typical Digital Map System

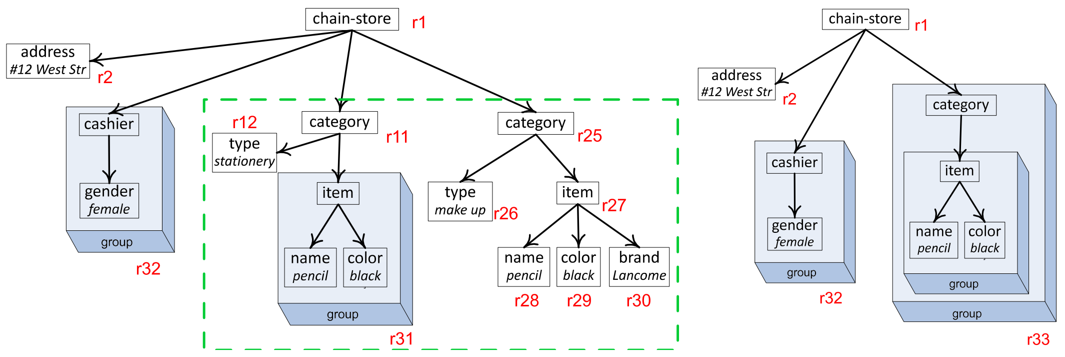
Before we go into the detail of XMAP working on XML data, let us have a quick review on how a typical digital map [71, 47] works. Figure 5.2 shows the working process of a traditional digital map system, which is adopted by XMAP:

- to support the zoom in/out function, the system needs to generate multiple **layers** of data, each of which is a complete map but has different scales of details. Higher level has less details and thereby its size is smaller.

- the area of a user's device where the map can be displayed is called the **window**. To simplify the discussion, we can assume the window size is fixed . The system will extract a "window" of content from a specific layer to the user.

- when a user moves left/right/up/down the map, the missing data to be displayed in the window will be transferred from the system to the user; when a user zooms in (out), the current layer providing data to the user will be changed to a lower (higher) layer. Note that the window size is fixed even the current layer is changed.

In this chapter, we will address the challenges and give solutions in building a map-like exploration model on XML database.



(a) Layer$_2$ of Figure 5.1 (b) Layer$_3$ of Figure 5.1

Figure 5.3: Generating layer$_2$ and layer$_3$ for Figure 5.1

Our main contributions in this chapter include:

1. We point out that returning a list of subtrees as the results of a keyword query on XML is not sufficient to meet the information needs of different users. We address the needs for a way to easily adjust the query results and show the query results in the global context.

2. We propose a novel exploration model, XMAP, to work as a complementary component of the XML keyword search engine, in order to enhance users' search experience. It can even work independently as a new way to explore the XML database. We find a reasonable way to generate different layers for XMAP on XML data. Meanwhile, we also build a practical index indexing all the layers generated.

The rest of the chapter is organized as follows. We present how to generate layers and build index for XMAP in Section 5.2. Section 5.3 discusses how to highlight the query results in XMAP. Section 5.4 presents the algorithms. Experimental evaluation is in Section 5.5. Then we conclude in Section 5.7.

5.2 Building XMAP

As we can see in Section 5.1, the most important tasks for building a digital map system are 1) to generate multiple layers with different scales of details; 2) to build index for retrieving data from different layers. However, due to the difference between XML data and geographic data, these two main tasks pose new challenges to XMAP working on XML database, which we will discuss in the following two subsections.

5.2.1 Generating Layers for XMAP

Let the number of layers be n, which is decided by the *scale factor* s:

Definition 5.1. ***Scale factor*** s *is the width*[1] *ratio of two adjacent layers, i.e.* $s = width(layer_i)/width(layer_{i+1})$.

Scale factor is normally set to a number from 2 to 5. E.g., Google Maps set it to 2.

Layer$_1$ is the most detailed layer; layer$_n$ is the most abstract layer and supposed to be with a size such that it can fit into the users' window. Therefore, the relation between n and s is:

$$width(layer_1)/s^{n-1} = width(window) \tag{5.1}$$

[1]Height can also be used for a digital map system, where each layer has a fixed height:width ratio.

which means layer$_1$, the most detailed layer, will be scaled to generate layer$_2$, layer$_3$, ..., layer$_n$.

Therefore, layer$_1$ is easy to generate. We only need to present the whole XML as a tree without losing any detail on layer$_1$. Then every node in the XML has its own coordinates (x, y) on the layer. Note that these coordinates will be the global coordinates and unchanged in other layers, because the other layers are generated based on layer$_1$.

Now we need to generate layer$_2$ up to layer$_n$, which is the main problem in this subsection. Normally, for geographic data, these layers will be generated by scaling layer$_1$ with fixed height:width ratio. The fixed height:width ratio is required because of the demand of reflection of reality. E.g., the shape of a country should not be changed no matter which layer it is in.

However, an XML database normally stores millions of similar-structured data at the same hierarchical level. E.g., right below a node of a chain-store, there might be a great amount of nodes representing items being sold. Therefore, the height:width ration is almost zero for layer$_1$. It will be problematic when we generate the other layers with fixed height:width ratio. For example, layer$_n$ is supposed to fit into the user's window. But the almost-zero height:width ratio leads to the fact that layer$_n$ will not be readable because it is just like a horizontal line presented in the user's window.

Therefore the challenge is how to scale on the dimension of width when we generate layer$_2$ to layer$_n$. Our solution to the challenge is to **merge** the *compatible* subtrees into groups in the XML data tree, in order to scale the layer on the dimension of width. Before introducing the concept of *compatible*, we will define a term which will be used.

Definition 5.2. ***Node Type*** *The type of a node v in the XML tree, denoted as* *v.type, is the tag name path from root node of the XML tree to v.*

For example in Figure 5.1, the node type of node $r13$(item) is "*company/chain-store/category/item*".

Definition 5.3. ***Compatible*** *Given an XML data tree, two subtrees T_1 and T_2 are said to be compatible if and only if they satisfy the following conditions:*

1. *the root of T_1 and the root of T_2 are siblings;*

2. *the root of T_1 and the root of T_2 correspond to the same node type.*

For example, in Figure 5.1, the subtrees surrounded by a dash line rectangle on the right hand side are compatible. The roots of the three subtrees are: node $r13$(item), node $r17$(item) and node $r21$(item). They are siblings and of the same node type.

With the concept of compatible, to generate layer$_{i+1}$ from layer$_i$, we can merge the *compatible* subtrees into a *group* in order to hide the details and shrink the layer on the dimension of width.

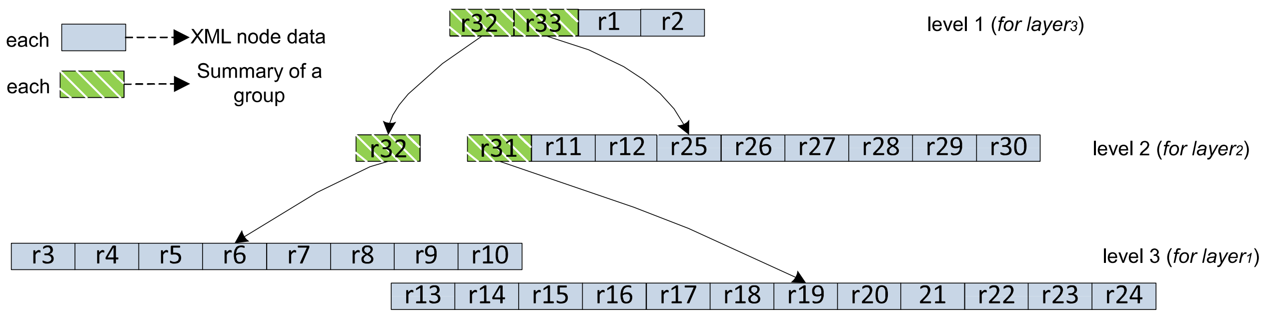


Figure 5.4: Index of the data shown in Figure 5.1

Since some subtrees will be hidden in a group, we need to show a summary of the information in a group. Here we adopt [39] to generate a snippet as a summary for the information of a group. Asides from the snippet, we can also show the value range for some of the important nodes appearing in the group.

Besides, XML data has a hierarchical structure. The data that is nearer to the leaf node tends to be a piece of more-detailed information. E.g., in Figure 5.1, the category information is more abstract than the item information. Hence, when we generate layers, item information should be hidden first and then the category information. So we will group the compatible subtrees in a *bottom-up* manner, which means a subtree T will not be merged with other subtrees until all the compatible subtrees in T have already been merged.

Figure 5.3 shows the merging process for the data shown in Figure 5.1. When generating layer$_{i+1}$ from layer$_i$, how many subtrees will be merged into one group will depend on the scale factor. E.g., suppose there are 100 compatible subtrees and the scale factor is 3. There is no need to merge all the 100 compatible subtrees into one group. Maybe merging 3 compatible subtrees into one group is sufficient to shrink the width by 2/3.

5.2.2 Index of XMAP

With all the layers ready, now we need to organize the index for efficient retrieval of data from different layers. Since R-tree [32] is a popular data structure used for spatial access methods, here we will propose a revised R-tree data structure tailored for indexing the layers generated in Section 5.2.1.

Normally, R-tree is used to index spatial data in a *single* multi-dimensional space. However, in XMAP, there are a number of closely related layers, which means there are a number of two-dimensional space. One possible solution is to build one R-tree index for each layer. But we observed that each layer is closely related to one another. Following we will propose a revised R-tree index, which can index all the layers in one R-tree rather than multiple R-trees.

As we can observe in the Section 5.2.1, XMAP has the following two properties:

1. **P1**: each layer$_i$ of XMAP is generated based on a previous layer$_{i-1}$ by merging compatible subtrees together. Therefore, amongst the layers, there exists an intrinsic hierarchical structure.

2. **P2**: each layer$_i$ shrinks based on the previous layer$_{i-1}$ by a scale factor (see Definition 5.1).

These two properties inspired us to index all the layers in one revised R-tree, where each level of the R-tree corresponds to a layer in XMAP. P1 can ensure the hierarchical structure of the tree index. P2 can ensure that each level of the tree index has more entries than the previous level by a certain scale, i.e. the scale factor.

Figure 5.4 shows our index for the data shown in Figure 5.1. The striped entries, called *group entries*, represent a group in the XMAP layer; the pure-color entries, called *node entries*, represent a regular node in a XMAP layer. Each entry in the index has a pointer pointing to the storage data, which can be either a regular XML node data or the summary information of a group. This is different from a normal tree index where only the leaf entries have such pointers.

As shown in Figure 5.4, the first level of the index keeps track of the data on layer$_n$, the second level of the index keep track of the data on layer$_{n-1}$, and so on and so forth. For example, at the second level of the index, entry $r31$ corresponds to the group $r31$ shown in Figure 5.3(a). More group nodes will appear at level 2 if there are more item information shown in Figure 5.1.

When we index the layers, we need to take note of the following two things:

1. A node entry only needs to appear in the index once while the same group entry could appear several times. It is because we can observe that when we find a node entry at level i, we know that the node entry also appear at level

i+1 and onwards. E.g., entry $r1$ only appear at level 1 in Figure 5.4, which also means it is also a part of level 2 and level 3 in the index. Because node $r1$ in Figure 5.1 never merge with any subtree and therefore will appear at all layers.

2. From the index we can know that, because the outdegree of an entry in the index depends on how many subtrees are merged together. Therefore, when we merge compatible subtrees to generate layers in Section 5.2.1, we prefer to merge as less number of subtrees as possible. E.g., suppose there are 1000 compatible subtrees of node type A, 1000 compatible subtrees of node type B, the scale factor is 2. Then to shrink the layer by scale factor 2, we may have three different choices: 1) merging 1000 compatible subtrees of node type A into one group without making any changes to those of node type B; 2) merging 1000 compatible subtrees of node type B into one group without making any changes to those of node type A; 3) merging every two compatible subtrees of A into a group and merging every two compatible subtrees of B into a group. We prefer the third choice because it will reduce the outdegree of the index entry.

As we can see from Figure 5.4, group entries are intentionally put in the front of the index node, so that the group entries can fit into a disk page. If the scale factor is s, approximately there will be s groups/subtrees merged into one group, so there are s group entries in an index node approximately[2]. These s group entries in an index node can be stored in one disk page. Besides, we can easily arrange the layout of layer$_1$, such that the area represented by the same-level entries will not overlap with each other. Therefore, the complexity of search is at most $O(log_sN)$, where N is the total number of XML nodes.

[2]Figure 5.4 can show this property better if more data is available in Figure 5.1.

5.3 XMAP Working with a Search Engine

When XMAP works as a complementary component of an XML keyword search engine, there is one more task to accomplish: we need to figure out some ways to highlight the query results in XMAP. For example, suppose a search engine will display a list of five query results on a showing page, these five query results will need to be highlighted in XMAP.

The layout of XMAP is generated statically offline based on the XML data regardless of any query. But the query results are generated dynamically based on each coming query from users. Basically, there are two ways to highlight the query results in XMAP. In this section we will study both of them. (1) The first approach is a static approach. We can highlight the query results right in XMAP, i.e. highlight the area for each result in XMAP using colors, lines, etc. However, sometimes those results are far away from each other in XMAP. It is not easy to show the relationship among those results because there are too many irrelevant subtrees in between them. What is worse, those results may not be visible at the same layer in XMAP. These pose challenges to the first approach. (2) The second approach is to dynamically generate a new display based on the query results, i.e., assembling all result subtrees together to form a new display. After that users pick up any result they want to further explore. Then we try to highlight the that particular result subtree in the XMAP. Then we can avoid the drawbacks of the static approach.

5.3.1 Static Approach: Highlight all Query Results in XMAP

An intuitive and easy solution is to highlight all the query results directly in XMAP. If all the nodes of the query results are visitable at the layer at which the

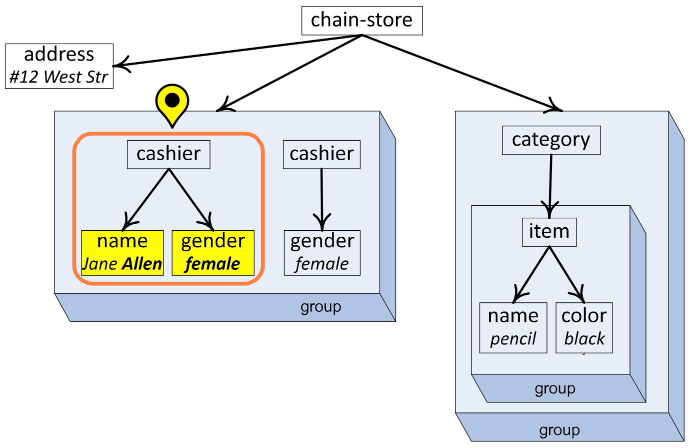


Figure 5.5: Query results highlighted of the query "Allen female" at layer$_3$

user is browsing, it is easy to highlight the results because we only need to highlight the nodes by color, lines, etc. However, the nodes of the query results sometimes are not available at users' current layer. In other words, the nodes will be available only when the user zooms in to an appropriate layer. In this case, the user may not be able to judge which result may be better and which group she should zoom in first.

Therefore, to further enhance users' search experience, we need to find a way to highlight the query results even if the relevant nodes are not available at the user's current layer.

Our solution to highlight the query results whose nodes are not available at the current layer is: we show the important nodes and paths of a query result in the area of the summary information of a group. Figure 5.5 shows a result highlighted at a certain layer.

Now we will define *skeleton* of a query result, which is considered as important nodes and paths of a query result.

Definition 5.4. ***Skeleton*** *For a query result R, where the user's query keywords are matched to a set of nodes $\{v_1, v_2, ..., v_k\}$, the skeleton of R are defined as a* *set of nodes* $\{LCA(v_1, v_2, ..., v_k), v_1, v_2, ..., v_k\}$ *and the paths linking all these nodes. Function* $LCA()$ *returns the lowest common ancestor of a set of nodes*[3].

So when the nodes of a query result are not available at a certain layer, the query result will be highlighted by showing its skeleton at the summary area of the corresponding group at the layer.

Example 5.3. *For a keyword query "Allen female" issued on the XML data shown in Figure 5.1, the leftmost subtree rooted at node* $r3$*(cashier) is one of the results by LCA. The query keywords are matched to the nodes* $\{r4, r5\}$*. Then the skeleton of the query result is the set of nodes* $\{LCA(r4, r5), r4, r5\}$ *and the paths linking them, which is shown in the orange rectangle in Figure 5.5. Therefore, when the user is browsing at layer*$_3$*, its skeleton is shown at the summary area of the corresponding group.* □

Another drawback of the static approach is, if the query results are far away from each other in the XML tree, the results will be divided by the content in between them. In such a case, the relationship among the query results will be hard to show on users' screen. Therefore, we try to overcome this drawback by proposing a dynamic approach as shown in Section 5.3.2.

5.3.2 Dynamic Approach: Generate a New Display by Assembling the Query Results

As discussed in the previous section, one drawback of the static approach is that a lot of irrelevant information will be shown in between the result subtrees if the result subtrees are far away from each other in the XML data. To overcome the drawback, we proposed the dynamic approach: to generate a new display by

[3]Other matching semantics are also applicable. We only use LCA as an example for illustration.

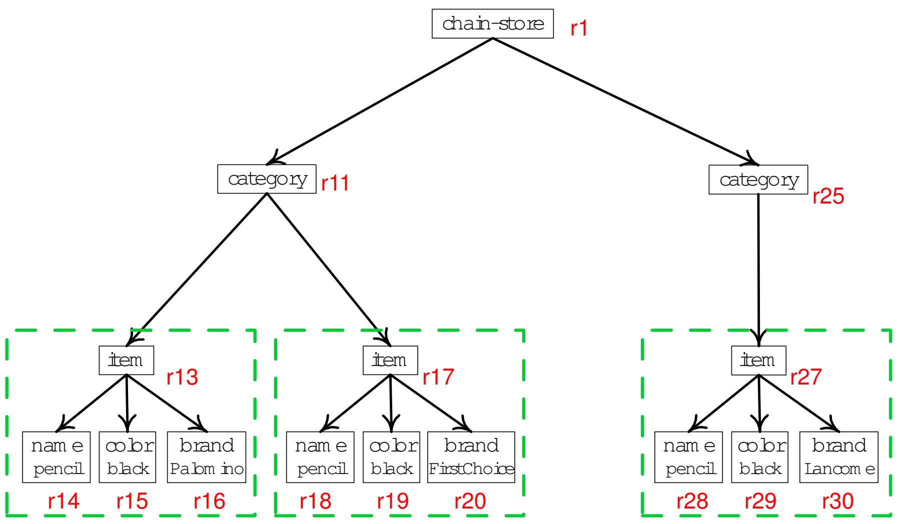
assembling the query result subtrees together. Such a display is generated dynamically catering for a particular query and its results. We call such a display *Context Display*, which is to help users understand the relationship among the result subtrees. Then users pick up any result on the display which they want to further explore. Only after that, we will show users the XMAP with that particular result subtree highlighted.

To generate a display to show the relationship among the query results, we need to show how the result subtrees are connected together in the XML document. An intuitive way is to assemble the result subtrees by showing the paths in the XML document which connect all the result roots. Given a set of result subtrees with their roots being $\{r_1, r_2, ..., r_n\}$, we will include the following information to form the display: (1) the result subtrees; (2) the LCA node l of $r_1, r_2, ..., r_n$; (3) the path from l to r_i for each $i \in [1, n]$.

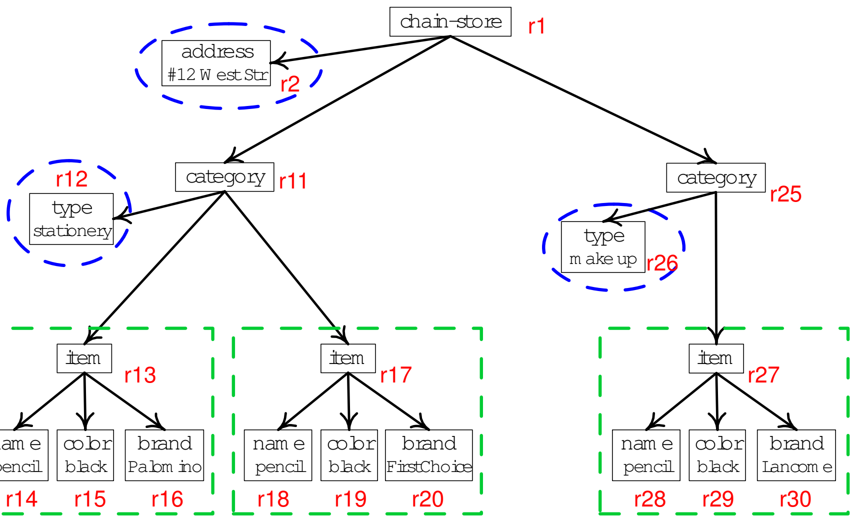
Example 5.4. *For a keyword query "pencil black" issued on the XML data shown in Figure 5.1, it will get the following results by LCA:*

1. *subtree rooted at node* $r13$*(item),*
2. *subtree rooted at node* $r17$*(item),*
3. *subtree rooted at node* $r27$*(item).*

By assembling the results together, we can get a new display as shown in Figure 5.6(a), which is dynamically generated and can show the relationship among the results subtrees. It consists of three parts: (1) the result subtrees, which is in the dashed rectangles. (2) the LCA node $r1$ *of all the result subtree roots. (3) the paths connecting* $r1$ *and all the result subtree roots, which involve node* $r11$ *and* $r25$*. With the context display, users can easily tell that result 1 and result 2 are under the same category while result 3 is under another another category. Meanwhile, all* *these three results are under the same chain-store. In such a way, the relationship among the three results is clearly presented to users.* □



(a) Basic Context Display



(b) Enhanced Context Display

Figure 5.6: Context Display for the Query Results of Query “pencil black”

As we can see from the above example, although we can know that the three result subtrees are under different categories, users do not have more information on what categories they are. Therefore, a very natural way to enhance the context display is to show more information for the nodes which are on the paths connecting the result subtree roots. E.g., for the node $r1$, $r11$ and $r25$ in Figure 5.6, we can show one child node for each of them, where such a child node should identify its parent. Figure 5.6(b) shows an enhanced context display for Figure 5.6(a). The three nodes in the dashed circles are shown because they can well identify their parents.

If the XML data is modeled using some semi-structured data model, like ORA-SS (Object-Relationship-Attribute Model for Semi-Structured Data) [27, 63], we can choose to show the Identifier/Key node of an Object to construct an enhanced context display. If such semantics is unavailable, it can be discovered by some existing algorithms [58] for XML data. Actually ORA-SS can capture the semantics that which nodes in the XML data represent objects, object attributes, relationships, relationship attributes, etc. Such semantics can further help with the query result presentation. E.g., sometimes a result subtree could be very big and contains multiple levels of Objects nested within the subtree. In that case, we can make use of ORA-SS to pinpoint the Object which is searched for by users, and show the Object to users rather than showing the whole result subtree. How to make full use of ORA-SS to further improve the result presentation for XML keyword search is one of our future work.

5.4 Algorithms

5.4.1 Index Construction

To support the function of XMAP, a revised R-tree index is needed to be built as discussed in Section 5.2.2. The revised R-tree index is built based on the layers generated. So in order to build the index, the layers need to be generated first. Algorithm 6 shows the main procedure to build the index.

Algorithm 6: buildIndex(XML, *window*, s)

```
input  : XML data tree $XML$, window size $window$, scale factor $s$
output : Revised R-tree index $index$
1  Database¡dewey, node-groupInfo¿ $layer$;
2  parseXML($XML$, layer[1]);
3  n = LOG(s, layer[1].getWidth()/window.width) + 1;
4  foreach $i \in 2$ to $n$ do
5      $compatibleList$ = findCompatibleSubtrees($layer[i-1]$);
6      $k$ = widthMeasure($compatibleList$, $s$, $layer[i-1]$);
7      $layer[i]$ = merge($k$, $compatibleList$, $layer[i-1]$);
8  contructRTreeLevel(1, $layer[n]$, $index$);
9  foreach $j \in n-1$ to 1 do
10     contructRTreeLevel($n-j+1$, $layer[j]$, $index$);
11 return $index$;
```

Since we need to record the layout of each layer, we may need to store a large amount of data if the XML data is huge. Therefore, we use a database to record the layout of each layer (line 1). The tuple of the database is in the form of *(dewey, node-groupInfo)*. The key is a Dewey [93] label *dewey* assigned to each node or group[4] to facilitate compatible subtree detection. The data part is the information about the node or group, including coordinates, summary information, etc. Next we will generate layer$_1$, which can be accomplished when the XML data tree is parsed. So we parse the XML data tree and generate layer$_1$ (line 2). With layer$_1$ generated, we can calculate the number of layers according to Equation 5.1 (line 3). Then we will generate layer$_2$ to layer$_n$ (line 4 to 7). First we need to check the previous layer to find out all the compatible subtrees that can be merged at this stage (line 5). Note that we find compatible subtrees in a bottom-up manner as discussed in Section 5.2.1. With the list of compatible subtrees, we can measure the width and figure out how many compatible subtrees should be merged into a group to shrink the layer by the scale factor (line 6). Then we can merge the

[4]For a group, which contains a set of compatible subtrees, we will arbitrarily choose one subtree and use its dewey label to represent this group. This dewey label is already sufficient to be used to detect the sibling relationship between this group and any other subtrees.

compatible subtrees to generate the layer (line 7).

With all the layers ready, now we can construct the revised R-tree index according to each layer as discussed in Section 5.2.2. We will construct level 1 based on layer$_n$ (line 8). After that we will construct level 2 to n accordingly (line 9 to 10).

5.4.2 Retrieving data from the index

In our index, not only the leaf entry, but also every level of entry has pointers pointing to storage data. Therefore the way we retrieve data will be a bit different. Algorithm 7 presents how the data will be retrieved from the index by a recursive function.

Algorithm 7: retrieveData(*coor*, *wind*, *level*, *indexNode*)

```
   input  : if a user request for an area of data, coor is the coordinates of the top
            left corner of the area; the user's window size wind; level is the level on
            which data will be retrieved; indexNode is for recursive call usage,
            initially pass in the root of the index tree
   output : data in the requested area
1  if $level < indexNode.level$ then
2      return;
3  requiredData ← $\emptyset$;
4  $area = (coor.x, coor.y, coor.x + wind.width, coor.y + wind.height)$;
5  foreach entry $e \in indexNode$ do
6      if $e.coordinates \in area$ then
7          if e is a node entry then
8              requiredData.add($e \to nodeData$);
9          else
10             retrieveData(coor, wind, level, $e \to pointerNextLevel$);
11 return requiredData;
```

The parameter *indexNode* is for recursive usage. Initially, the root of the index will be passed in as *indexNode*. When the level of *indexNode* is higher than the required level, function will return (line 1 to 2). Otherwise, the area, in which the data is requested, will be decided (line 4). The next step is to check all the entries in *indexNode* (line 5). If the entry represents some data in the required area (line 6), the data will be part of the return: (1) if the entry is a node entry, the data will be added to the return (line 7 to 8); (2) if the entry is a group entry, a recursive call is used to retrieve data from the index node which is pointed by the group entry (line 9 to 10).

5.5 Experiments

We have conducted some experiments to verify the performance of our new index. We mainly study two aspects of the index: index size and efficiency.

All experiments are conducted on a 2.83GHz Core 2 Quad machine with 3GB RAM running 32-bit windows 7. All codes are implemented in Java. The dataset we used is IMDB[5] 100MB, where around 200,000 movies of recent years are selected in our dataset. Each movie contains information like title, rating, director, cast, etc.

Table 5.1: Index Size for Subsets of IMDB dataset

Dataset Size	XML Nodes	Layers	Index Size (KB)
1MB	50,429	6	1,125
10MB	504,288	8	10,085
100MB	4,531,695	10	90,633

Index Size. First of all, we study the index size of our index introduced in Section 5.2.2. The IMDB dataset is 100MB. To study how the index size will increase when the data size increases, we extract a 1MB subset and a 10MB subset from the 100MB dataset as two new datasets. Then we build up the index with scale factor 4 on each of them. The number of layers needed is calculated according to Equation 5.1. Table 5.1 shows the index sizes for the three datasets.

[5]http://www.imdb.com/interfaces

As shown in the table, the index size increases linearly as the dataset size increases. This is a preferred property of our index. Here for the IMDB datasets, the index size is slightly larger or smaller than the dataset size. This is because the size of our index depends on the number of nodes in the XML as we are building index on the XML nodes. IMDB dataset is data centric and it contains a huge number of nodes, where each node only contains a short textual content, like “movie”, “The Matrix”, etc. Our index needs to store the coordinates of each node as discussed in Section 5.2.1. The coordinates are formed of four integers representing an area of a rectangle. So here for IMDB dataset, the size of the coordinates are almost the same as the size of the textual content of each node.

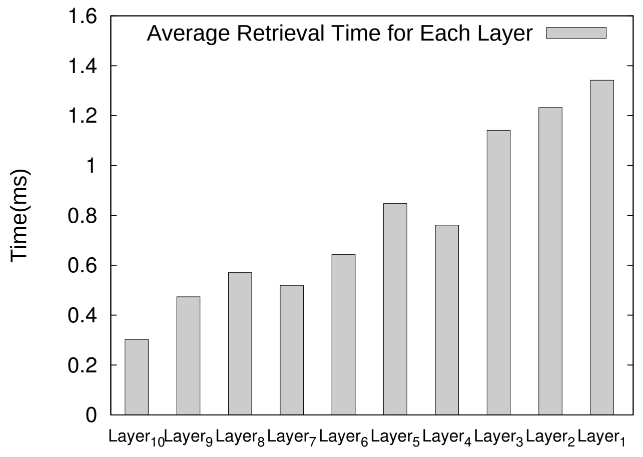


Figure 5.7: Average Retrieval Time for Each Layer

Efficiency. Secondly, we also test the efficiency of our index. We evaluate the retrieval time on the index of the 100MB dataset. The index indexed 10 layers of data. For each layer, we generate 100 random queries and each query retrieves a window of data from that layer. The window size is set to be 1024 * 768 (pixel).

Figure 5.7 shows the average retrieval time of the 100 random queries for each layer. We can find that accessing higher layer, which is more abstract, will cost less time. This is because of two reasons. Firstly, accessing a higher layer does not require to search until the leaf nodes of the index. E.g., retrieval of data from layer$_9$ only requires accessing the index from level 1 to level 2, as discussed in Section 5.2.2. Secondly, group entries represent a larger area than the node entries. So getting a window of data from a higher layer will get less number of entries.

5.6 XMAP Demo System

To overcome the shortcoming of existing techniques for displaying query results, we developed the system XMAP [101] to enhances users' search experience by 1) showing query results in a more human-understandable way in the global context of the whole XML document; 2) providing users an easy way to make adjustment to the query results without revising and resubmitting the keyword query. So far the system mainly focuses on XML data without ID references, as adding support for XML data with ID references to the system is one of our future work.

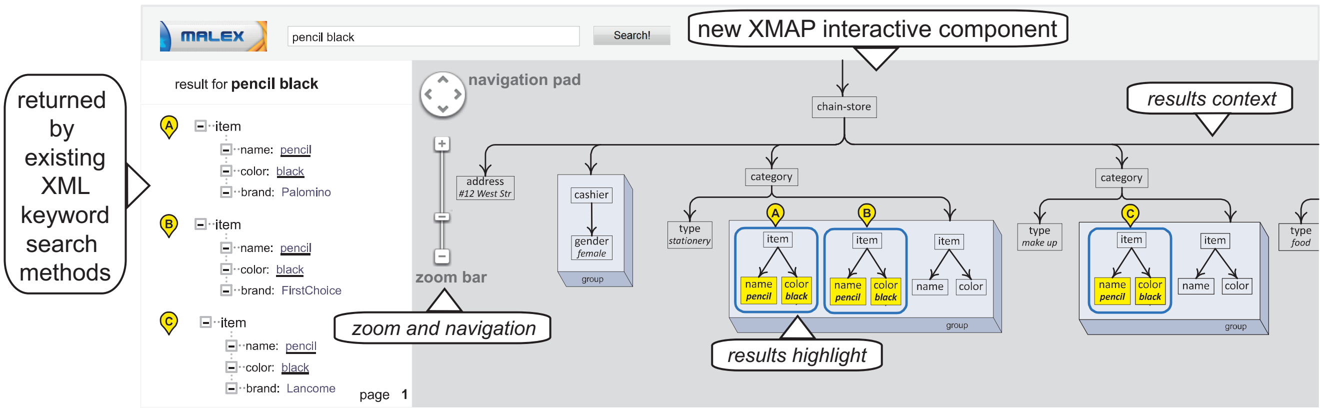
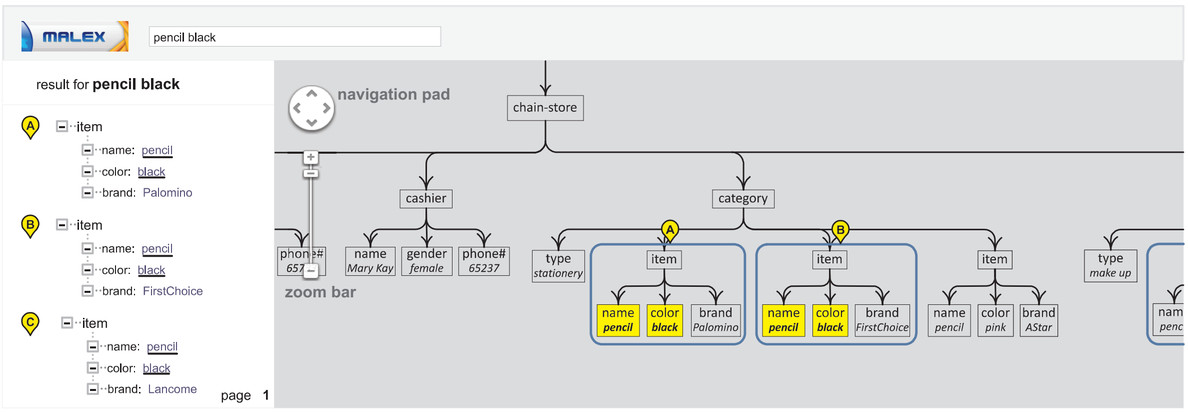


Figure 5.8: Screenshot of XMAP for the query in Example 5.1

Figure 5.8 shows a screenshot of XMAP for the query "pencil black" in Example 5.1. As we can see, on the left hand side, it shows the results returned by existing XML keyword search methods page by page. On the right hand side, the XMAP display window works as an interactive component for users to visualize, manipulate and further explore the query results.

Addressing Motivation 1 XMAP displays the results in a global context, which makes it much easier to digest the query results. As shown in Figure 5.8, for the query "pencil black", the three pencils being returned, namely *A*, *B* and *C*, are not all in the same category. From XMAP display, we can easily know that result *C* is a make up pencil rather than a normal pencil. This is not possible to know with the traditional result list without XMAP. On the left hand side of the XMAP display window, users can use the zoom slider bar to zoom in/out the results to see more details, as shown in Figure 5.9.

Addressing Motivation 2 In the XMAP display window, a dragging pad and zoom sliding bar are provided for user to move left/right/up/down and zoom in/out, to further explore the query results and XML data. In this way, users with different search intentions can easily adjust the query results to meet their information needs without revising and resubmitting the keyword query. E.g., for the query "Allen female" in Example 5.2, user can easily use the dragging pad to explore the information of a *cashier* or the *chain-store* just above it.



pencil black

result for **pencil black**

A item
- name: pencil
- color: black
- brand: Palomino

B item
- name: pencil
- color: black
- brand: FirstChoice

C item
- name: pencil
- color: black
- brand: Lancome

page 1

Figure 5.9: Screenshot of XMAP for the query in Example 5.1 (zoomed in)

Please refer to the Appendix B of the thesis for more details of the XMAP system.

5.7 Conclusion

In this chapter, we point out that returning a list of subtrees as the results of a keyword query on XML is not sufficient to meet the information needs for different users. Because the query results as subtrees are actually inter-connected in the XML document, where they may have silbling/containment relationships. It is imprecise and could be misleading without showing proper context of the results. To further bridge the mismatch gap between users' search intention and the query results, a way to easily adjust the query results and show the query results in the global context is in demand. To address the problem, we propose our map-like exploration model, XMAP, to work as a complementary component of the XML keyword search engine, in order to enhance users' search experience and bridge the mismatch gap. We proposed a visualization way to generate different layers for the XML data with different levels of details, such that users can zoom in/out the display to meet their needs with different levels of details. Meanwhile, all results are visualized in the context of the whole XML document. The context of the query results could act as an important part for users to understand the results. Besides, we also build a variant of R-tree index indexing all the layers generated, where each layer corresponds to one level in the R-tree index. After that we also provide solutions to highlight the query results by showing the structure of the results: either statically highlighting the query results in XMAP or dynamically generating a context display by assembling the result subtrees.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Due to the easy-to-use query interface, XML keyword search has been a hot research topic and abundantly studied in the last 10 years. In this thesis, we focus on improving the usability of XML keyword search. Existing works mainly focus on defining matching semantics and proposing efficient algorithms for computing query results for a particular matching semantics. However, how to reduce the mismatch gap between users' search intention and the query results remains a challenge. Users have to reformulate and resubmit their queries 40% to 52% of the time in order to get what they want [86] even for the mature web search. In order to bridge the mismatch gap, we try to tackle the challenges in the following three aspects:

- We find that traditional keyword search methods on XML tree will return a list of erroneous mismatch results to users when what they search for is unavailable in the data. We define it as the MisMatch problem in XML keyword search. Based on the abundant meta data and the structure in the XML document, we propose a novel concept called Target Node Type (TNT) to infer users' search intention from the query results. With the inferred search intention, we can easily check whether each query result complies with its corresponding TNT for MisMatch problem. Users need explanation and suggestion when the results are mismatched with their expectation. We propose a data-driven approach to discover the keywords which cause the MisMatch problem and generate some suggested queries based on another concept we introduced, distinguishability, which is used to measure the importance of query keywords. The suggested queries guarantee to have reasonable results.

- We then extend our mismatch solution from XML tree onto XML IDREF digraph considering the ID references in the XML data. Traditional keyword search methods on XML IDREF digraph adopt pure graph search or its variants. However, such pure graph search techniques suffer from efficiency issue since such techniques can be as inefficiency as NP-complete. We notice that XML IDREF digraph is actually mainly a tree plus a portion of ID reference edges. So We try to convert the XML IDREF digraph to a tree model such that we can apply the efficient XML tree search methods and apply our previous mismatch solution as well.

- Besides, we point out that the traditional way of showing a list of independent subtrees as query results is imprecise and could be misleading. We improve the existing result presentation model by showing the query results in the global context of the whole XML document and providing a way to easily further explore the query results.

After handling the mismatch caused by result retrieval and mismatch caused by result presentation, we theoretically and experimentally demonstrate the effectiveness of our approach in improving the usability of XML keyword search. We will conclude each of above aspects in the following sections.

MisMatch Problem in Keyword Search Over XML without ID References

In Chapter 3, we identify the MisMatch problem in XML keyword search. We develop a low-cost post-processing algorithm on the results of query evaluation to detect and solve the MisMatch problem specially for XML tree. Since all existing keyword search methods for XML tree are LCA-based, they will all try to return a set of subtrees (with or without ranking) containing all the query keywords as query results, regardless of users' search intention. Even what users search for is unavailable in the XML data, they are not able to be aware of such a fact and will still return a list of erroneous mismatch results to users. Therefore, to solve the MisMatch problem, we propose a post-processing methodology to detect the mismatch between users' search intention and the returned results.

(1) For detecting the MisMatch problem, we propose a novel concept called Target Node Type (TNT) to infer users' search intention for each query result. TNT is defined making use of the structure information in the XML data. With the TNT inferred, we then compare each query result with its corresponding TNT for the MisMatch problem. We choose to take a conservative approach: we only judge a query to have the MisMatch problem if none of the query results matches its Target Node Type. Such a conclusion holds for all users with different intentions. If the query is without the MisMatch problem, the original results will be returned without any suggestion. Otherwise, query suggestion will be generated for the queries with MisMatch problem.

(2) To generate suggestion to users for the MisMatch problem, we take the following approach: First, we propose a novel measure called distinguishability to measure the importance of query keywords. Second, we try to discover the approximate results containing the important keywords in the XML data. Third, to form the suggested queries, we will find some new keywords within the approximate results to replace the less important keywords in the original query. Both of these keywords are required to be from the same type of nodes in order to make sure the semantics of the keywords are the same. Finally, a score function is proposed to rank the suggested queries taking the following three factors into consideration: the number of keywords which need to be replaced, the sum of distinguishability of the keywords that need to be replaced and the compactness of the approximate result. Our approach generating the suggested queries is data-driven such that the suggested queries guarantee to have no MisMatch problem and have reasonable results.

(3) To discover the approximate results efficiently, we also propose a novel bitmap labeling scheme. The empirical study on three real datasets in experiments demonstrates the effectiveness and efficiency of our approach. It evaluates the detection accuracy and suggestion quality, as well as the efficiency and scalability.

(4) A search engine called XClear [104] which embeds the MisMatch problem solution is also built.

MisMatch Problem in Keyword Search Over XML with ID References

Later in Chapter 4, we observe that an XML IDREF digraph (with ID references considered) is mainly a tree structure with a portion of reference edges. We propose a novel method to transform an XML IDREF digraph with ID/IDREF to a tree model, such that we can exploit the XML tree search methods to work on XML IDREF digraph, and subsequently our MisMatch solution designed for XML tree still applies to the XML IDREF digraph. Existing keyword search methods for XML IDREF digraph compute the query results by graph traversal. Then the keyword search problem on an XML IDREF digraph is reduced to the problem of finding Minimal Steiner Tree (MST) or its variants in a digraph, where an MST in a digraph is defined as a minimal directed subtree containing all query keywords in either its leaves or root. Since finding all MSTs in a digraph is an NP-complete problem, efficiency is one of the notable issues. What is more, the matching semantics and the keyword search methods are different from those for an XML tree. So it poses new challenges when we try to extend our mismatch solution from XML tree mode to XML IDREF digraph .

(1) We propose a novel way to transform an XML IDREF digraph to a tree model by virtually replicating the subtrees being referenced. Our tree model consists of two parts: an XML tree and a table (called reachability table), which is capable of handling different kinds of reference patterns in an XML IDREF digraph. The experimental results show that our approach is orders of magnitude faster than the traditional search methods for XML IDREF digraph while generating a similar set of results as existing XML IDREF digraph search methods, which could be as inefficient as NP-complete.

(2) After we transfer an XML IDREF digraph to the tree model and adopted the XML tree search methods, the key steps of the mismatch solution, namely MisMatch problem detection and suggestion generation, can be applied here with necessary variation. The major variation is 1) if a *keyword match node* is within a replicated subtree, we need to infer its node type with the help of the reachability table; 2) since there could be exponentially many node types in an XML IDREF digraph, it is not a feasible solution to consider all those node types when we calculate distinguishability or build the exLabel for each node in the data. But we

notice that many node types are actually representing the same type of information (see Chapter 4 Section 4.6). So a feasible solution is that, when we calculate distinguishability or build exLabel, we use the node types without ID reference edge in its path, called *solid node types*, to simulate other node types which represent the same type of information.

Query Result Presentation of XML Keyword Search

In Chapter 5 we point out that the traditional method of presenting the query results as a list of independent subtrees is imprecise and misleading. Users could misunderstand the results because such subtrees are actually interconnected in the XML document and simply showing such subtrees is insufficient and misleading. In order to further reduce the gap between users' search intention and the query results, we need to present the query results in a precise way with proper context provided. Actually each query result of XML keyword search is a part of the XML data tree rather than a piece of independent information. Among the query results (subtrees), they may have sibling or containment relationships. Without showing such relationships, users may misunderstand the query results and digest the information wrongly.

(1) To improve the usability by addressing the above issues, we propose a map-like model, call XMAP, for presenting the query result in the global context of the whole XML document and in an interactive way. It can work as a complementary component of the XML keyword search engine. We propose a visualization way to generate different layers for the XML data with different levels of details, such that we can provide users an interactive mechanism to zoom in/out or navigate the display according to their needs viewing different levels of details. Meanwhile, the query results are presented in the context of the whole XML document. Therefore, users can clearly view the context and the relationship among the query results,

where the context of the query results is an important part for users to understand the results.

(2) Besides, we also build a variant of R-tree index indexing all the layers generated, where each layer corresponds to one level in the R-tree index. A demo system of XMAP has also been built [101].

Finally, to provide a complete experience of our research work done to enhance XML keyword search usability, we have also discussed about how to integrate the two demo system mentioned above, XClear and XMAP, in Appendix C.

6.2 Future Work

Improving the usability of XML keyword search is an important issue as it is necessary to make XML keyword search usable for average daily users. While in this thesis we have proposed several solutions, we would like to further explore this topic in several directions in the future.

Making Use of ORA-SS Model For MisMatch Problem

In this thesis, we proposed a solution for the MisMatch problem in XML keyword search. In our solution we assume that there is no outer semantics available. In other words, we only have the XML document itself with the accompanied schema specification, like DTD or XML Schema. As a future work, we would like to further explore the topic of using ORA-SS (Object-Relationship-Attribute Model for Semistructured Data) [27, 63] to further improve the MisMatch solution. ORA-SS is an XML data model which models an XML document by objects, object attributes, relationships, relationship attributes, etc. Most of the semantics captured by ORA-SS data model is not able to be expressed by DTD or XML

Schema. Because DTD or XML Schema only captures the hierarchical structure of the XML document but not the semantics of objects, relationships, etc. For example, relationship attributes in the data can only be expressed as normal attributes using DTD or XML Schema; n-ary relationship (n>2) cannot be expressed in DTD or XML Schema, etc.

In the MisMatch solution, one important step is to infer the Target Node Type (TNT) in order to know whether the query result is consistent with the TNT. Our current solution is inferring the TNT based on the structural property of the XML document, i.e. node types. If the XML data is model with ORA-SS model, it can capture the semantics that which nodes in the XML data represent objects, object attributes, relationships, relationship attributes, etc. With such semantics available, we can better infer Target Node Type and therefore improve the accuracy of the MisMatch problem detection. For example, suppose there is an XML document describing some information about employees in a company, each *employee* node can contain multiple *qualification* nodes as its children. Each *qualification* node contains a *university* node and a *graduate_year* node. In this case, if the data is modeled by ORA-SS model, we can know that *employee* nodes are object nodes while *qualification* nodes are not object nodes but multi-valued composite attributes of the *employee* object. With the information where the object nodes are located, we can avoid inferring non-object nodes as the Target Node Type, like *qualification* node in this case. Therefore with a more accurate Target Node Type, we can achieve better accuracy for the MisMatch problem detection.

So with the semantics provided by ORA-SS model, we can study how to make use of them to infer users' search intention in a more systematic and precise way and improve the MisMatch problem detection.

Making Use of ORA-SS Model For Result Presentation

If an XML document is model with ORA-SS model, it can greatly help with the result presentation. Since an XML document, or even a subtree of a query result, can be too huge to display to users. In that case, deciding what information to hide and what information to show is an unavoidable topic. With the help of ORA-SS model, we can improve the result presentation in many ways.

Knowing which nodes represent objects, object attributes, relationships, relationship attributes, etc., we can choose to show the object nodes which are users' search target, as well as the attributes of those desired objects. In such a manner, even a result subtree contains multiple levels of objects nested within the subtree, we can easily choose to show those desired object nodes and hide the other object nodes. For example, suppose there is a document describing some information of a shopping mall. Under the *mall* node there could be many *shop* nodes representing different shops. If users' search target is to find the information of a shopping mall, then we can only show the node which represents the mall object as well as the attribute nodes of the mall object, like address, phone number, etc. Under the *mall* node, there are many object nodes representing different shops in the shopping mall. We can hide them as they are not the desired objects. We can also provide a mechanism for users to expand those *shop* nodes if they want.

From ORA-SS we can also know which attribute is the ID attribute of an object. With such information, we can avoid showing the same object multiple times for query result presentation, by removing those duplicated objects with the same ID attribute value. For example, suppose there is an XML document describing some suppliers and the parts they supply, where there is an $m : m$ binary relationship between object *supplier* and object *part*. If *part* nodes are designed to be under the *supplier* nodes, then the same *part* could be duplicated under many different *supplier* nodes because many *suppliers* could supply the same *part*. In this case, if users want to find the information of a particular part and input the name of the part, there will be many duplicated answers being returned as the same part could be duplicated many times under different *supplier* nodes in the XML data. Knowing the ID attribute of an object can help us removing those duplicated objects with the same ID attribute value when we display the query results. Note that here the data of *supplier* and *part* is stored without using ID references and there is an $m : m$ relationship among *supplier* and *part*, so the same *part* has to be duplicated multiple times under different *suppliers*. In this case, we are not able to use DTD or XML Schema to specify the ID attributes of objects , as DTD or XML Schema can only define ID attributes for reference purpose and does not allow two nodes in the XML data with the same ID attribute value.

Besides, being able to locate relationship attribute nodes can also greatly help with displaying the results in an understandable way. E.g., if users are trying to find the price of a laptop, we need to show the node representing the laptop object as well as the price attribute of the laptop. Meanwhile, it is also important to show the node representing the shop object which sells that laptop. Because *price* is usually designed to be a relationship attribute between a shop object and a laptop object.

Thus, we can try to make full use of ORA-SS model to improve result presentation for XML keyword search and it can be a promising direction.

MisMatch Solution for Schema-independent Keyword Search Method

Currently, most of the XML keyword search techniques are sensitive to the schema of the XML data. In other words, even for the same keyword query, query results could be very different if the schema is designed differently. However, keyword search is becoming more and more popular because users do not have to learn the schema of the data before they can issue a query. So normally users do not know the schema of data and the query results they expect from the search engine are not depending on the schema design.

For example, suppose we need to store some data about teachers and students. We can design the schema of XML in two possible ways: 1) teacher nodes are on top of student nodes; 2) student nodes are on top of teacher nodes. For a query containing two student ID, “stuID1 stuID2”, existing LCA-based search techniques can find a common teacher who teaches those two students in the first schema design. However, for the second schema design, the query will get the root of the document as the query result. This is because LCA-based search techniques are trying to find some subtrees containing all query keywords as results. Being sensitive to schema design will require users to dig into the schema design of the XML data, which is obviously not a desired feature.

Recently, [49] proposed a schema-independent keyword search method for XML data, which is not sensitive to the schema design when generating results for a keyword query, i.e., it returns the same set of answers even for different schemas of the same data content. This is done by defining a new matching semantics called CR (Common Relative), where ancestors and descendants of a node u are also relatives of u. Since our MisMatch solution is for the traditional Lowest-Common-Ancestor-based matching semantics, how to extend our MisMatch solution onto the recent schema-independent keyword search method needs to be further studied.

BIBLIOGRAPHY

[1] *Berkeley DB*. http://www.sleepycat.com.

[2] *Bing Maps*. http://maps.bing.com.

[3] *Google Maps*. http://maps.google.com.

[4] *jQuery JavaScript Library*. http://jquery.com.

[5] *Raphael JavaScript Library*. http://raphaeljs.com.

[6] *XML ClearMap*. http://xclearmap.comp.nus.edu.sg.

[7] *Yahoo Maps*. http://maps.yahoo.com.

[8] Zhifeng Bao, Tok Wang Ling, Bo Chen, and Jiaheng Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, pages 517–528, 2009.

[9] Zhifeng Bao, Jiaheng Lu, Tok Wang Ling, Liang Xu, and Huayu Wu. An effective object-level xml keyword search. In *DASFAA*, pages 93–109, 2010.

[10] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[11] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernndez, Michael Kay, Jonathan Robie, and Jrme Simon. Xml path language (xpath) 2.0. In *W3C Recommendation*, 2010.

[12] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[13] Scott Boag, Don Chamberlin, Mary F. Fernndez, Daniela Florescu, Jonathan Robie, and Jrme Simon. Xquery 1.0: An xml query language. In *W3C Recommendation*, 2010.

[14] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. In *ACL*, pages 286–293, 2000.

[15] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[16] Stuart K. Card, George G. Robertson, and William York. The webbook and the web forager: An information workspace for the world-wide web. In *CHI*, pages 111–117, 1996.

[17] S. Jeromy Carrière and Rick Kazman. Webquery: searching and visualizing the web through connectivity. In *Selected papers from the sixth international conference on World Wide Web*, pages 1257–1267, 1997.

[18] Adriane Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[19] Hsinchun Chen, Haiyan Fan, Michael Chau, and Daniel Dajun Zeng. Metaspider: Meta-searching and categorization on the web. *JASIST*, 52(13):1134–1147, 2001.

[20] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[21] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD*, pages 455–466, 2005.

[22] Joel Coffman and Alfred C. Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Trans. Knowl. Data Eng.*, 26(1):30–42, 2014.

[23] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[24] J Cugini, C Piatko, and S Laskowski. Interactive 3d visualization for document retrieval. In *Proceedings of the Workshop on New Paradigms in Information Visualization and Manipulation*, pages 3–10, 1996.

[25] Ricardo da Silva Torres, Celmar G. Silva, Claudia Bauzer Medeiros, and Heloisa Vieira da Rocha. Visual structures for image browsing. In *CIKM*, pages 49–55, 2003.

[26] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[27] Gillian Dobbie, Wu Xiaoying, Tok Wang Ling, and Mong Li Lee. Ora-ss: Object-relationship-attribute model for semistructured data. In *Technical*

Report TR 21/00, School of Computing, National University of Singapore, Singapore, 2001.

[28] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. In *Networks*, pages 195–207, 1971.

[29] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[30] Jiafeng Guo, Gu Xu, Hang Li, Xueqi Cheng, and Xueqi Cheng. A unified and discriminative model for query refinement. In *SIGIR*, pages 379–386, 2008.

[31] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, pages 16–27, 2003.

[32] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[33] Marios Hadjieleftheriou, Amit Chandel, Nick Koudas, and Divesh Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

[34] Susan Havre, Elizabeth G. Hetzler, Kenneth A. Perrine, Elizabeth Jurrus, and Nancy Miller. Interactive visualization of multiple query results. In *INFOVIS*, pages 105–112, 2001.

[35] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[36] Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, and Divesh Srivastava. Keyword proximity search in xml trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[37] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.

[38] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[39] Yu Huang, Ziyang Liu, and Yi Chen. Query biased snippet generation in xml search. In *SIGMOD*, pages 315–326, 2008.

[40] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[41] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of twig queries with or-predicates. In *SIGMOD*, pages 59–70, 2004.

[42] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed xml documents. In *VLDB*, pages 273–284, 2003.

[43] Rosie Jones, Benjamin Rey, Omid Madani, and Wiley Greiner. Generating query substitutions. In *WWW*, pages 387–396, 2006.

[44] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[45] M. Kickmeier and D. Albert. The effects of scanability on information search: An online experiment. In *Proceedings of HCI 2003: Designing for Society*, pages 33–36, 2003.

[46] Benny Kimelfeld and Yehoshua Sagiv. Combining incompleteness and ranking in tree queries. In *ICDT*, pages 329–343, 2007.

[47] Daisuke Kitayama, Takahiro Teratani, and Kazutoshi Sumiya. Digital map restructuring method based on implicit intentions extracted from users' operations. In *ICUIMC*, pages 45–53, 2008.

[48] Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, pages 377–439, 1992.

[49] Thuy Ngoc Le, Zhifeng Bao, and Tok Wang Ling. Schema-independence in xml keyword search. In *ER*, 2014 (to appear).

[50] Thuy Ngoc Le, Huayu Wu, Tok Wang Ling, Luochen Li, and Jiaheng Lu. From structure-based to semantics-based: Towards effective xml keyword search. In *ER*, pages 356–371, 2013.

[51] Ki-Hoon Lee, Kyu-Young Whang, Wook-Shin Han, and Min-Soo Kim. Structural consistency: enabling xml keyword search to eliminate spurious results consistently. *VLDB J.*, 19(4):503–529, 2010.

[52] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.

[53] Anton Leuski and James Allan. Lighthouse: Showing the way to relevant information. In *INFOVIS*, pages 125–129, 2000.

[54] Changqing Li and Tok Wang Ling. Qed: a novel quaternary encoding to completely avoid re-labeling in xml updates. In *CIKM*, pages 501–508, 2005.

[55] Changqing Li, Tok Wang Ling, and Min Hu. Efficient processing of updates in dynamic xml data. In *ICDE*, pages 13–22, 2006.

[56] Guoliang Li, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.

[57] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[58] Luochen Li, Thuy Ngoc Le, Huayu Wu, Tok Wang Ling, and Stéphane Bressan. Discovering semantics from data-centric xml. In *DEXA*, pages 88–102, 2013.

[59] Mu Li, Muhua Zhu, Yang Zhang, Ming Zhou, and Ming Zhou. Exploring distributional similarity based models for query spelling correction. In *ACL*, pages 1025–1032, 2006.

[60] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[61] Wen-Syan Li, K. Selçuk Candan, Quoc Vu, and Divyakant Agrawal. Retrieving and organizing web pages by "information unit". In *WWW*, pages 230–244, 2001.

[62] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free xquery. *VLDB*, pages 72–83, 2004.

[63] Tok Wang Ling, Mong Li Lee, and Gillian Dobbie. Applications of ora-ss: An object-relationship-attribute data model for semistructured data. In *IIWAS '01: Proceedings of 3rd International Conference on Information Integration and Web-based Applications and Serives.*

[64] Ziyang Liu and Yi Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD*, pages 329–340, 2007.

[65] Ziyang Liu and Yi Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.

[66] Ziyang Liu, Peng Sun, and Yi Chen. Structured search result differentiation. *PVLDB*, 2(1):313–324, 2009.

[67] Jiaheng Lu, Zhifeng Bao, Tok Wang Ling, Xiaofeng Meng, and Xiaofeng Meng. Xml keyword query refinement. In *KEYS*, pages 41–42, 2009.

[68] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[69] Yifei Lu, Wei Wang, Jianxin Li, and Chengfei Liu. Xclean: Providing valid spelling suggestions for xml keyword queries. In *ICDE*, pages 661–672, 2011.

[70] Ye Ma, Derong Shen, Yue Kou, and Wei Liu. An effective query relaxation solution for the deep web. In *APWeb*, pages 649–659, 2008.

[71] Toshiyuki Masui, Mitsuru Minakuchi, George R. Borden IV, and Kouichi Kashiwagi. Multiple-view approach for smooth information retrieval. In *ACM Symposium on User Interface Software and Technology*, pages 199–206, 1995.

[72] Sougata Mukherjea and Yoshinori Hara. Visualizing world-wide web search engine results. In *IV*, pages 400–407, 1999.

[73] Ion Muslea. Machine learning for online query relaxation. In *KDD*, pages 246–255, 2004.

[74] Ion Muslea and Thomas J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.

[75] MySQL. http://www.mysql.com.

[76] Ullas Nambiar and Subbarao Kambhampati. Answering imprecise queries over autonomous web databases. In *ICDE*, pages 45–54, 2006.

[77] Bo Ning, Guoren Wang, and Jeffrey Xu Yu. A holistic algorithm for efficiently evaluating xtwig joins. In *DASFAA*, pages 571–579, 2008.

[78] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD*, pages 903–908, 2004.

[79] Fuchun Peng, Nawaaz Ahmed, Xin Li, Yumao Lu, and Yumao Lu. Context sensitive stemming for web search. In *SIGIR*, pages 639–646, 2007.

[80] Ken Q. Pu and Xiaohui Yu. Keyword query cleaning. *PVLDB*, 1(1):909–920, 2008.

[81] Yonggang Qiu, Hans-Peter Frei, and Hans-Peter Frei. Concept based query expansion. In *SIGIR*, pages 160–169, 1993.

[82] Knut Magne Risvik, Tomasz Mikolajewski, Peter Boros, and Peter Boros. Query segmentation for web search. In *WWW (Posters)*, 2003.

[83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[84] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *ALENEX*, pages 71–83, 2007.

[85] Albrecht Schmidt, Martin L. Kersten, and Menzo Windhouwer. Querying xml documents made easy: Nearest concept queries. In *ICDE*, pages 321–329, 2001.

[86] Amanda Spink, Bernard J. Jansen, Dietmar Wolfram, and Tefko Saracevic. From e-sex to e-commerce: Web search changes. *IEEE Computer*, 35(3):107–109, 2002.

[87] Anselm Spoerri. Rankspiral: Toward enhancing search results visualizations. In *INFOVIS*, pages 18–19, 2004.

[88] Chong Sun, Chee Yong Chan, and Amit K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[89] Yufei Tao, Stavros Papadopoulos, Cheng Sheng, and Kostas Stefanidis. Nearest keyword search in xml documents. In *SIGMOD*, pages 589–600, 2011.

[90] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[91] Arash Termehchy and Marianne Winslett. Using structural information in xml keyword search effectively. *ACM Trans. Database Syst.*, 36(1):4–52, 2011.

[92] Quoc Trung Tran and Chee-Yong Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[93] V. Vesper. http://www.mtsu.edu/vvesper/dewey.html.

[94] Bienvenido Vlez, Ron Weiss, Mark A. Sheldon, David K. Gifford, and David K. Gifford. Fast and effective query refinement. In *SIGIR*, pages 6–15, 1997.

[95] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, pages 443–454, 2003.

[96] Jinxi Xu, W. Bruce Croft, and W. Bruce Croft. Query expansion using local and global document analysis. In *SIGIR*, pages 4–11, 1996.

[97] Liang Xu, Zhifeng Bao, and Tok Wang Ling. A dynamic labeling scheme using vectors. In *DEXA*, pages 130–140, 2007.

[98] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. Dde: from dewey to a fully dynamic xml labeling scheme. In *SIGMOD*, pages 719–730, 2009.

[99] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, pages 537–538, 2005.

[100] Yu Xu and Yannis Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.

[101] Yong Zeng, Zhifeng Bao, Guoliang Li, and Tok Wang Ling. Exploring xml data is as easy as using maps. In *CIKM*, pages 2497–2500, 2013, demo paper. `http://zengyong.comp.nus.edu.sg/xmap.jsp`.

[102] Yong Zeng, Zhifeng Bao, Tok Wang Ling, H. V. Jagadish, and Guoliang Li. Breaking out of the mismatch trap. In *ICDE*, pages 940–951, 2014.

[103] Yong Zeng, Zhifeng Bao, Tok Wang Ling, and Guoliang Li. Efficient xml keyword search: From graph model to tree model. In *DEXA*, pages 25–39, 2013.

[104] Yong Zeng, Zhifeng Bao, Tok Wang Ling, and Guoliang Li. Removing the mismatch headache in xml keyword search. In *SIGIR*, pages 1109–1110, 2013, demo paper. `http://xclear.comp.nus.edu.sg`.

[105] Yong Zeng, Zhifeng Bao, Tok Wang Ling, and Luochen Li. Malex: a map-like exploration model on xml database. In *SIGMOD workshop KEYS*, pages 32–38, 2012.

[106] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.

[107] Wei Vivian Zhang, Xiaofei He, Benjamin Rey, and Rosie Jones. Query rewriting using active learning for sponsored search. In *SIGIR*, pages 853–854, 2007.

[108] Junfeng Zhou, Zhifeng Bao, Wei Wang, Tok Wang Ling, Ziyang Chen, Xudong Lin, and Jingfeng Guo. Fast slca and elca computation for xml keyword queries based on set intersection. In *ICDE*, pages 905–916, 2012.

[109] Junfeng Zhou, Xingmin Zhao, Wei Wang, Ziyang Chen, and Jeffrey Xu Yu. Top-down keyword query processing on xml data. In *CIKM*, pages 2225–2230, 2013.

[110] Rui Zhou, Chengfei Liu, and Jianxin Li. Fast elca computation for keyword queries on xml data. In *EDBT*, pages 549–560, 2010.

APPENDIX A: XCLEAR DEMO SYSTEM

Addressing the MisMatch problem, we have built an interactive XML keyword search engine called XClear [104], following our research work in Chapter 3. It can detect the MisMatch problem and show users why the MisMatch problem exists, as well as providing result-driven suggested queries to users. The system mainly focuses on XML data without ID references, while it will be one of our future work to add support for XML data with ID reference according to the solution in Chapter 4.

The architecture of XClear is shown in Figure 1. The *Index Constructor* constructs indexes for efficiently retrieving query results and maintaining node type information for the nodes in the XML data. The *Results Searcher* generates query results for the keyword query. After query results are generated, *Results Ranker* will rank the query results. The key feature of XClear lies in the *MM Component*, which has the original query and its results as input. It consists of three parts: (1) **MisMatch Problem Detector** infers the potential search targets and checks the MisMatch problem from the query results as discussed in Chapter 3. If the MisMatch problem exists, the *Suggested Query Generator* will be triggered. (2) **Suggested Query Generator** generates the suggested queries and a sample result for each suggested query, for user to verify its quality. (3) **Suggested Query Ranker** ranks all suggested queries according to our ranking model (Chapter 3). The ranked suggested queries and the corresponding sample query results will be returned to the user.

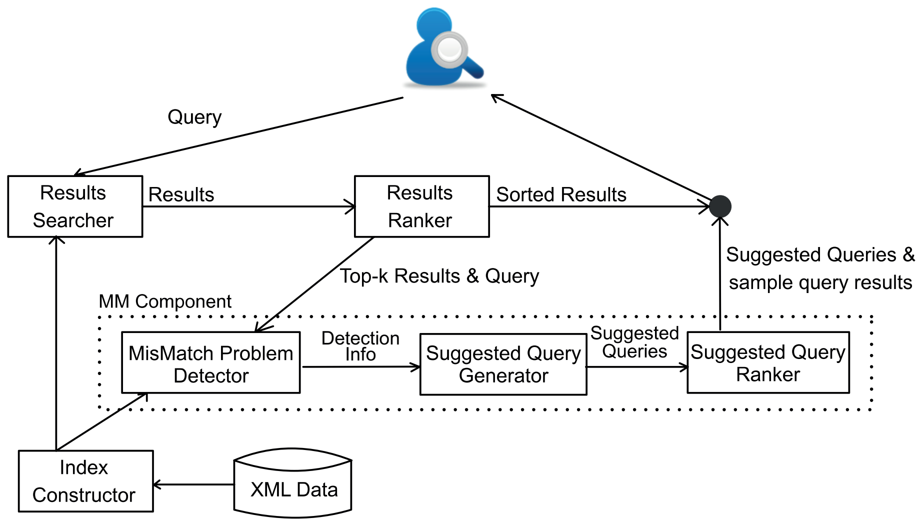
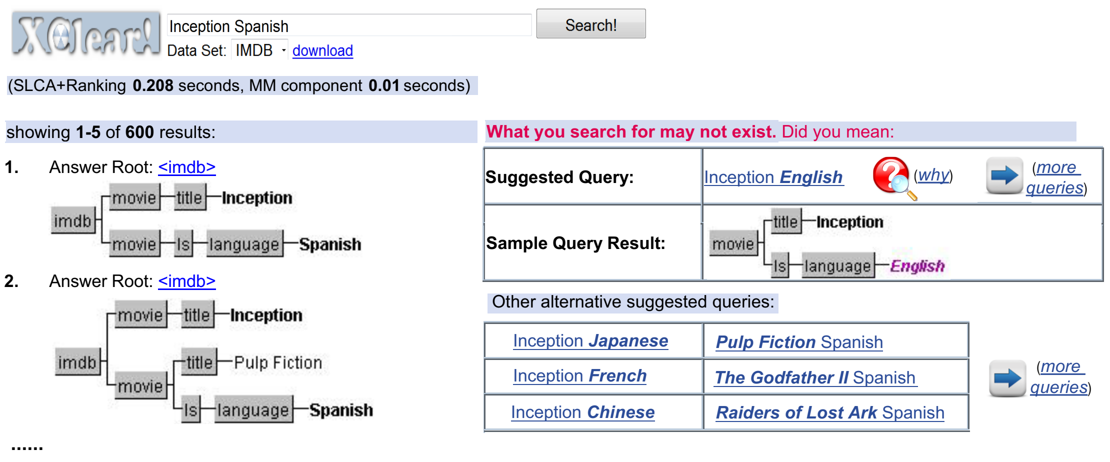


Figure 1: Architecture of XClear System

Next we will show how XClear can greatly enhance user's search experience in terms of three aspects: efficient, effective and user-friendly. As the ultimate goal, we want to demonstrate the ability of XClear in (1) showing the user why the mismatch exists and (2) providing result-driven suggested queries to bridge the mismatch gap.

We would like to highlight the UI design on how to further improve users' search experience. Figure 2 shows a screenshot for a query Q='Inception Spanish' in order to find the Spanish version of a movie Inception.

First, as shown in the left part of Figure 2, after the query results are computed


Inception Spanish Search!

Data Set: IMDB · download

(SLCA+Ranking **0.208** seconds, MM component **0.01** seconds)

showing **1-5** of **600** results:

1. Answer Root: <imdb>

2. Answer Root: <imdb>

......

What you search for may not exist. Did you mean:

Suggested Query:	Inception ***English*** (why) (more queries)
Sample Query Result:	

Other alternative suggested queries:

Inception ***Japanese***	***Pulp Fiction*** Spanish
Inception ***French***	***The Godfather II*** Spanish
Inception ***Chinese***	***Raiders of Lost Ark*** Spanish

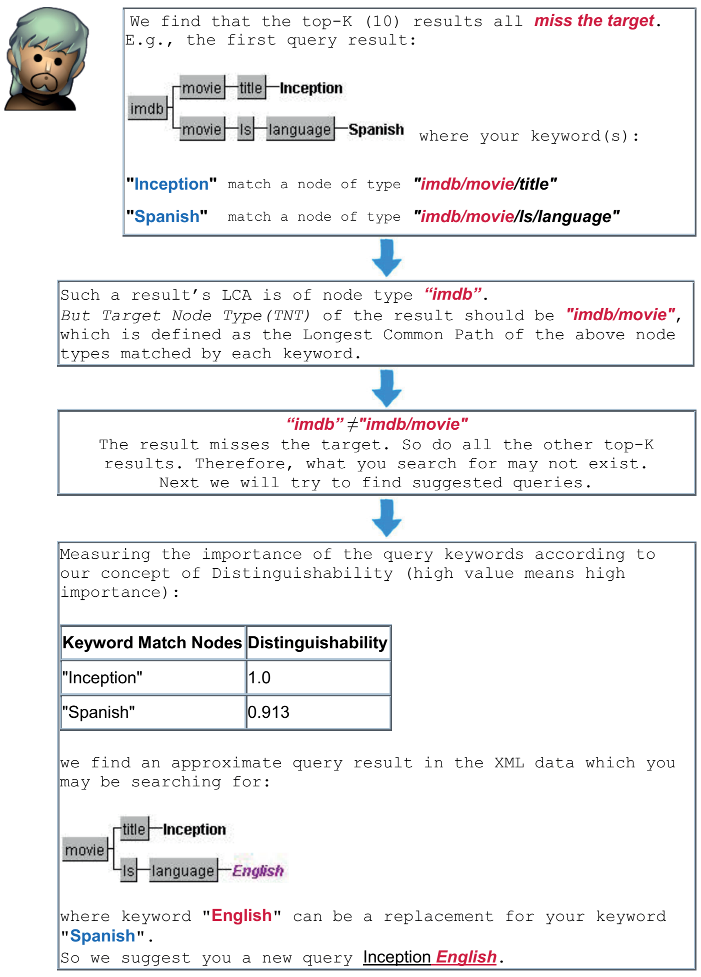
(more queries)

Figure 2: Suggested Queries & Sample Query Result

and ranked, each result is displayed as a tree rather than plain text, which makes the query results more highlighted and intuitive to user. Nodes in the XML data are represented as rectangles and values are represented as text. Each query keyword contained in the keyword match node will be shown in bold font such that the user can easily judge how her keywords are related to each other and whether the results are of her interest.

Next, *MM Component* will check all the retrieved results for the MisMatch problem. If the query has the MisMatch problem (see Chapter 3), XClear proceeds to generate and rank the suggested queries. Here for the query 'Inception Spanish', there is no Spanish version of the movie Inception in our database. So as we can see in Figure 2, the answer root for each result is *imdb*, where the language 'Spanish' matches one movie while the movie name 'Inception' matches another. Thus, what the user searches for does not exist and Q has the MisMatch problem.

As shown in the right part of Figure 2, *first*, a notification "What you search for may not exist" is displayed to the user. *Second*, the best suggested query and its sample result are provided. In the sample query result, the new keywords for replacement are highlighted in pink color and italic font, so that user can easily find



We find that the top-K (10) results all ***miss the target***.
E.g., the first query result:

where your keyword(s):

"**Inception**" match a node of type ***"imdb/movie/title"***

"**Spanish**" match a node of type ***"imdb/movie/ls/language"***

Such a result's LCA is of node type ***"imdb"***.
But Target Node Type(TNT) of the result should be ***"imdb/movie"***, which is defined as the Longest Common Path of the above node types matched by each keyword.

"imdb" $\neq$ ***"imdb/movie"***
The result misses the target. So do all the other top-K results. Therefore, what you search for may not exist.
Next we will try to find suggested queries.

Measuring the importance of the query keywords according to our concept of Distinguishability (high value means high importance):

Keyword Match Nodes	**Distinguishability**
"Inception"	1.0
"Spanish"	0.913

we find an approximate query result in the XML data which you may be searching for:

where keyword "**English**" can be a replacement for your keyword "**Spanish**".
So we suggest you a new query Inception ***English***.

Figure 3: Reasoning of "why"

out the difference between the new query and the original query. *Third*, a "*why*" button (next to the suggested query) is provided for user to get further *reasoning* on why we generate this suggested query. If the user agrees on the suggested query after viewing the sample result, she can submit the new query by simply clicking on the suggested query; otherwise, users can also view some other *alternative* suggested queries or even find more suggested queries by clicking the *"more queries"* button. All the suggested queries are derived from the XML data and guaranteed to have reasonable query results. E.g., the movie Inception has four languages in the data: English, Japanese, French and Chinese, which correspond to four of the suggested queries provided on the right of Figure 2.

Figure 3 shows the reasoning behind the suggested query after user click the "*why*" button in Figure 2. Such a step-by-step reasoning provides an intuitive yet clear way to illustrate how a suggested query is derived. It starts from the reason why MisMatch problem exists, and then displays the approximate results and highlights the 'important' query keywords, and finally shows how the suggested query is inferred. The detailed reasoning can give the user a comprehensive understanding on how we generate the suggested query.

APPENDIX B: XMAP DEMO SYSTEM

To tackle the drawbacks of the traditional way of displaying query results, we have developed the system XMAP [101] following our research work in Chapter 5, which offers a new and visual way for users to explore XML data and enhances users' search experience by 1) providing users an easy way to make adjustment to the query results without revising and resubmitting the keyword query; 2) showing the query results in a more precise and human-understandable way in the global context. The system mainly focuses on XML data without ID references. Support for XML data with ID references in the system will be one of our future work.

The system architecture of XMAP is shown in Figure 4. All the functionalities are supported by the components running at two sides: browser end and server end.

At browser end, it includes three components: *UI controller*, *MapPainter* and *Cache Manager*. *UI controller* captures the operations of the user. If the operations require to change the display in user's window, e.g. a zoom-in operation, it will pass a command to *MapPainter*, which is in charge of drawing the display according to the parameters (such as the number of current layer, the region needed to be displayed etc.), and highlighting the query results in the display. If some data is not available locally at the browser end (cache), it will inform the *Cache Manager* to load in the missing data. Each component in browser end is implemented in JavaScript.

At the server end, there are two main components: *Index Constructor* and *Request Handler*. *Index Constructor* constructs an R-tree liked index (see Chapter 5) for indexing the layers generated, so that MALEX can efficiently locate a specific region of data on a specific layer. *Request Handler* is a component handling all the data requests from the browser end. It will extract the required area of data through the index and send them to the user.

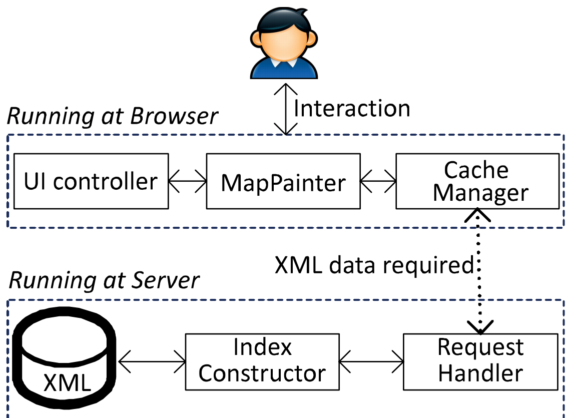


Figure 4: Architecture of XMAP

Figure 5 shows a screenshot of XMAP for the query “pencil black” in Chapter 5 Example 5.1. As we can see, on the left hand side, it shows the results returned by existing XML keyword search methods page by page. On the right hand side, the XMAP display window works as an interactive component for users to visualize, manipulate and further explore the query results.

XMAP Display (with Dynamically-loaded Data) On the right hand side of Figure 5, a XMAP display window is available to enhance users’ search experience. In the display window, users can see the XML data from a specific layer (see Chapter

5) in a map-like style. Data needed for display is dynamically loaded. For each XML node, the content of the node is shown in a 2-D rectangle, where tag names are shown in normal font and values are shown in italic font. The 3-D rectangles represent *groups*, each of which is a group of compatible subtrees as discussed in Chapter 5. On the surface of the 3-D rectangles, a summary of the group will be shown and the query results will also be highlighted.

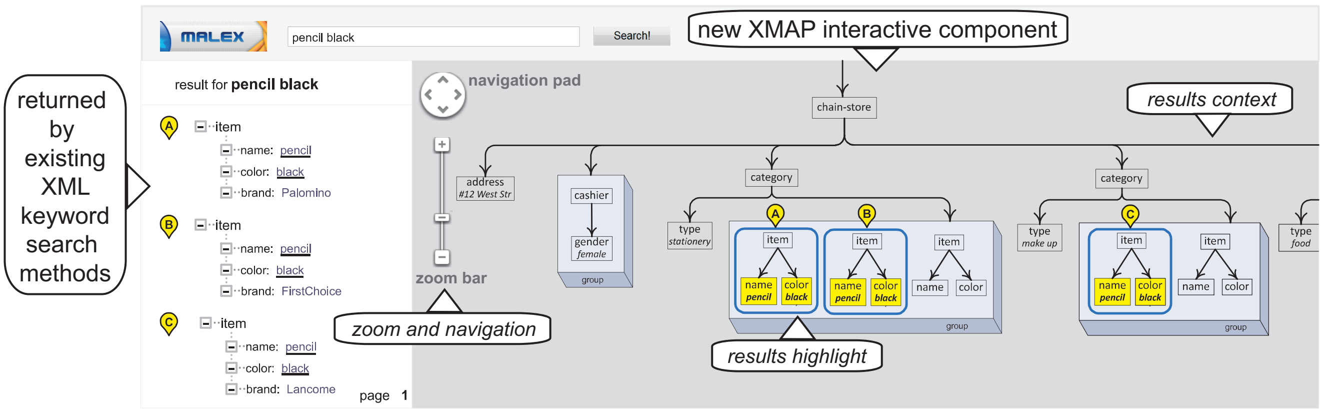


Figure 5: Screenshot of XMAP for a query “pencil black” addressing Motivation 1

Note that in Figure 5, on the left pane, the query results are displayed page by page if there are too many results. The results on the current page will be highlighted at the XMAP display, which is located at the right pane. Each query result is highlighted by an orange rectangle. The letter assigned to each result is

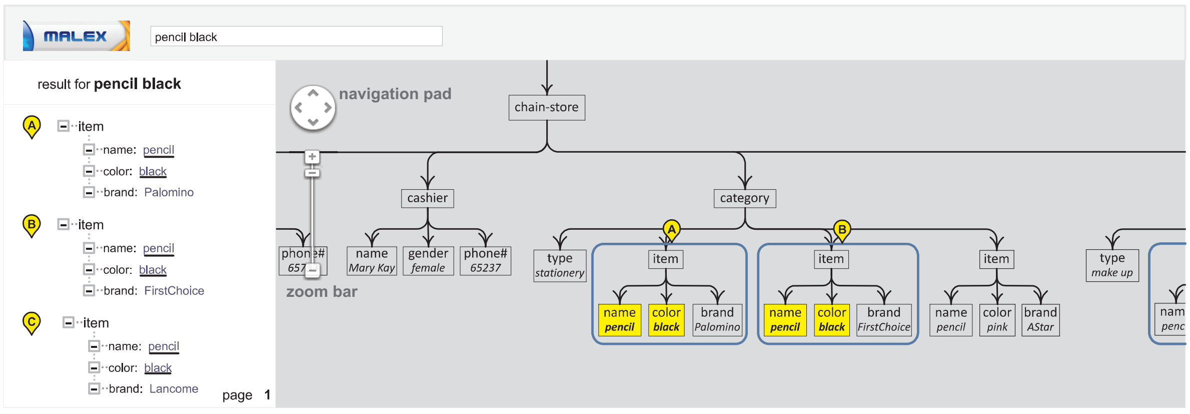


Figure 6: Screenshot of XMAP for a query “pencil black” (zoomed in)

also shown to help users distinguish the query results easily. Once user clicks on a particular result on the left pane, it will automatically take her to the corresponding subtree in the right pane (similar to Google Map).

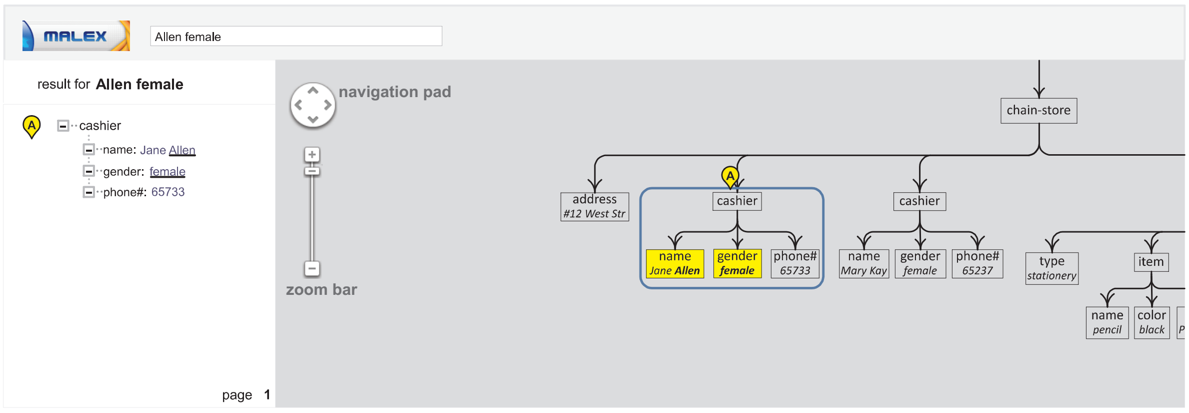


Figure 7: Screenshot of XMAP for a query "Allen female" addressing Motivation 2

Addressing Motivation 1 in Chapter 5, which leads to a demand of showing the relationships among the query results and the context of the query results. XMAP displays the results in a global context, which makes it much easier to digest the query results. E.g., for the query "pencil black" in Figure 5, the three pencils being returned, namely A, B and C, are not all in the same category. From XMAP display, we can easily know that result C is a make up pencil rather than a normal pencil. This is not possible to know with the traditional result list without XMAP. On the left hand side of the XMAP display window, users can use the zoom slider bar to zoom in/out the results to see more details, as shown in Figure 6. After zoomed in, users can now see the full subtree of the results.

Addressing Motivation 2 in Chapter 5, which calls for an easier way for users to further explorer the query results to find what they want. In the XMAP display window, a dragging pad and sliding bar are provided for user to move left/right/up/down and zoom in/out, to further explore the query results and XML data. In this way, users with different search intentions can easily adjust the query results to meet their information needs without revising and resubmitting the keyword query. As shown in Figure 7, for a query "Allen female", the user can easily use the dragging pad to explore the information of a *cashier* or the *chain-store* just above it. The user can see very easily from XMAP that the chain store which Allen works in is located at #12 West Str.

APPENDIX C: INTEGRATING XCLEAR AND XMAP

Since XClear and XMAP are two different systems focusing on query result refinement and query result visualization respectively, to provide a complete experience of all features in both systems, we have tried to integrate XClear and XMAP into one single system. We gave it a name *XML ClearMap* [6] which is coming from both the name *XClear* and *XMAP*. It is built by integrating XClear and XMAP with various enhancement. The major task for such an integration is to build a communication module to let XClear and XMAP work together. Besides, during the integration, the user interface of XMAP component is slightly different from the user interface mentioned in Appendix B because some UI implementation library has been changed [1]. So far the system mainly focuses on XML data without ID references, as adding support for XML data with ID references to the system is one of our future work.

[1]We have changed the web page UI library from jQquery v1.9.0 [4] to Raphael 1.11.1 [5] , considering that the latter can provide better efficiency. Such a change caused the UI to be slightly different.

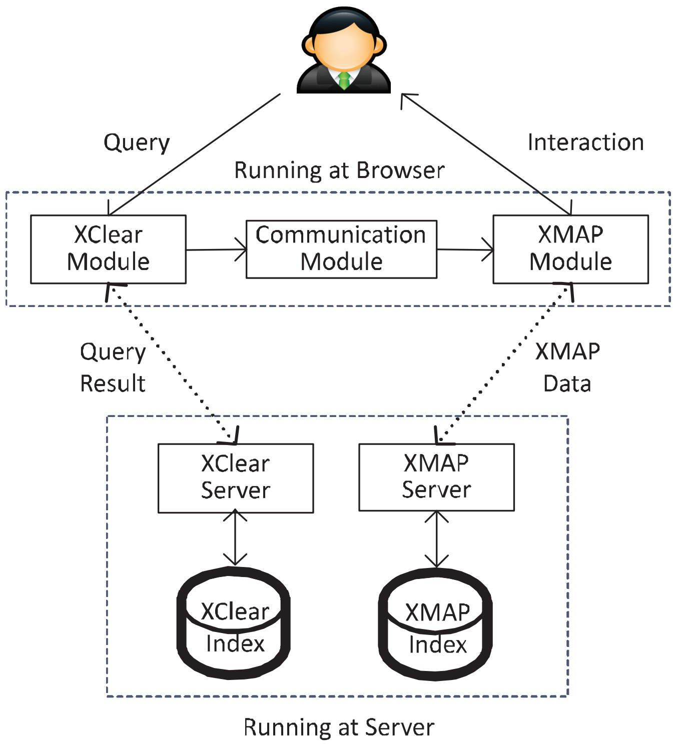
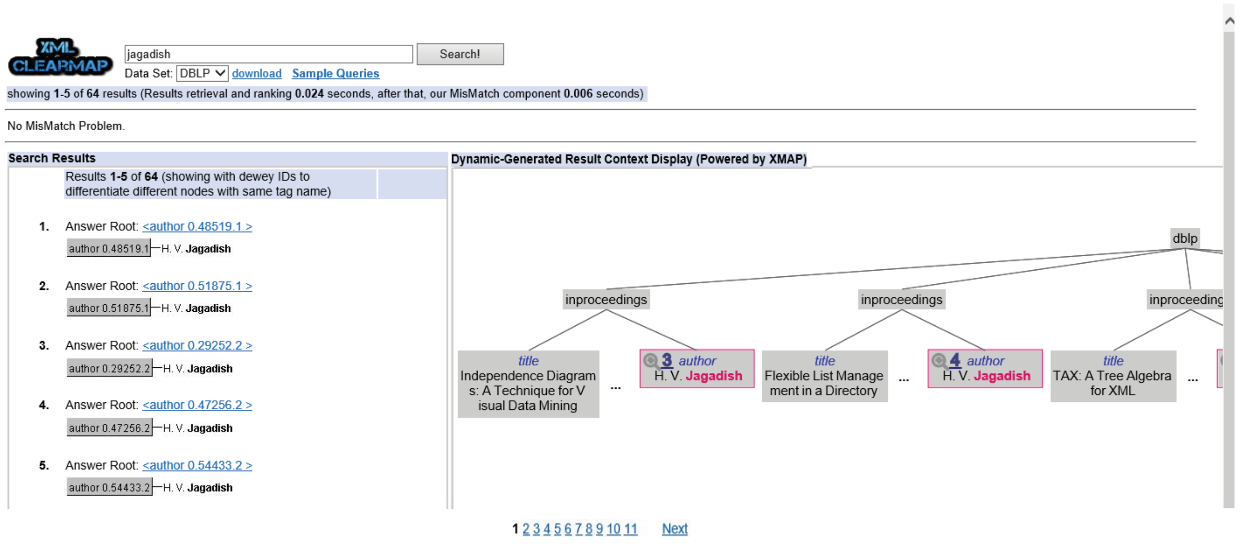


Figure 8: Architecture of XML ClearMap

Figure 8 shows the system architecture of XML ClearMap. The whole system consists of two parts. One part is the front end running at users' browsers. The other part is the back end running on the server.

At the front end, which runs at users' browsers, it contains an XClear module, an XMAP module and a Communication Module. Both of the XClear Module and the XMAP Module work in the same way as they do in the XClear demo system and XMAP demo system. So here we will only talk about how they will cooperate with each other rather than breaking down these two components. XClear Module gets users' keyword query and then send it to the server side for query evaluation (server end will be talked about later). After the server finishes the query evaluation, query results and query suggestion (if the query has MisMatch problem) will be returned to the XClear Module. Then XClear Module will show the query results and the query suggestion (if any) to the users. Since XMAP is to help visualize the query results, XClear Module needs to pass the query results to the XMAP Module. This process is done by the Communication Module, which will take the query results and convert them to the format required by the XMAP Module. When the XMAP Module get the query results with required format from the Communication Module, it will visualize the results in XMAP. Besides, XMAP is an interactive visualization module. Users can further explore the query results in XMAP. Sometimes the data which users are exploring may not be available at the browser side. In such a case, XMAP needs to dynamically load the missing data from the server side. It will send a request to the server side, i.e. XMAP Server in Figure 8. Then the required data will be sent to the XMAP Module.

At the server end, it contains two servers: XClear server and XMAP server. Both of them work independently and provide data to the XClear Module and XMAP Module respectively at the browser end. These two parts are the same as in the XClear demo system and XMAP demo system, details of which can be found in Appendix A and Appendix B.



showing 1-5 of 64 results (Results retrieval and ranking 0.024 seconds, after that, our MisMatch component 0.006 seconds)

No MisMatch Problem.

Search Results

Results 1-5 of 64 (showing with dewey IDs to differentiate different nodes with same tag name)

1. Answer Root: <author 0.48519.1 >
author 0.48519.1 — H. V. Jagadish

2. Answer Root: <author 0.51875.1 >
author 0.51875.1 — H. V. Jagadish

3. Answer Root: <author 0.29252.2 >
author 0.29252.2 — H. V. Jagadish

4. Answer Root: <author 0.47256.2 >
author 0.47256.2 — H. V. Jagadish

5. Answer Root: <author 0.54433.2 >
author 0.54433.2 — H. V. Jagadish

Dynamic-Generated Result Context Display (Powered by XMAP)

dblp
inproceedings
inproceedings
inproceeding
title
Independence Diagrams: A Technique for Visual Data Mining
...
3 author
H. V. Jagadish
title
Flexible List Management in a Directory
...
4 author
H. V. Jagadish
title
TAX: A Tree Algebra for XML
...
1 2 3 4 5 6 7 8 9 10 11 Next

Figure 9: XML ClearMap for Query without MisMatch Problem

Figure 9 shows a screenshot of XML ClearMap for a query “Jagadish”, which is without the MisMatch problem. As we can see, search bar is on the top of the page. If the query is without the MisMatch problem, there will be a hint under the search bar telling users that the query is without the MisMatch problem. On the left hand side of the screenshot, it is the result displaying window, where the result is shown in a traditional way. Each result is a subtree. In order to offer a way for users to see how the results relate to each other and to further explore the query results, we have a *Result Context Display* window on the right hand side of Figure 9, which is dynamically generated by the XMAP module based on the query results. Each grey rectangle represents a node in the XML data. The result subtrees are highlighted with pink border and result number. Besides of the result subtrees, the paths which connect each subtree are also shown. E.g., the query "Jagadish" in Figure 9 get a lot of author nodes as query results. But they are interconnected in the XML data rather than some independent subtrees. This is well expressed in the Result Context Display window, where we can see they are interconnected and under different *inproceedings* nodes. Besides, for each *inproceedings* node, we also show an attribute which can identify it, i.e., showing the *title* node under the *inproceedings* node.

In the Result Context Display window in Figure 9, users can click the result by the result number to further explore a particular result. After users' click, a new window called *Result Exploration Display* window will appear on the top of the current window, as shown in Figure 10. This window is to help users further explore the query result which users just clicked. It will locate the result subtree in the whole XML data. Users can explore and see any part of the XML data by navigating using a mouse. Users can drag the display to see the part which is not showing in the window. To zoom in, users can click the "zoom" icon or the suspension point. Then the display will be zoomed in and locate to the part which

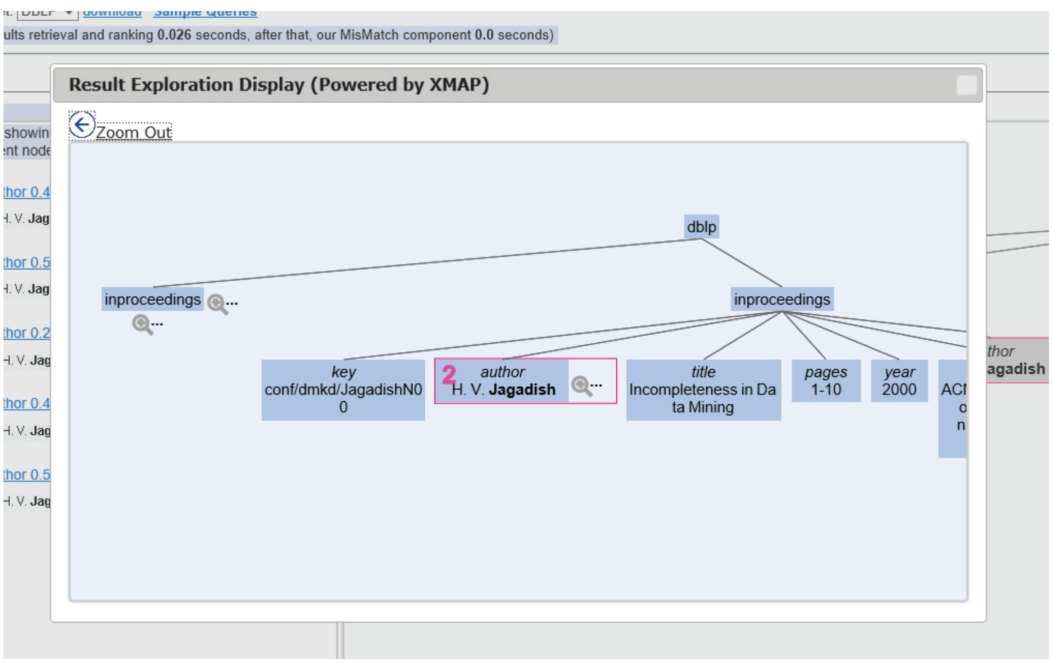
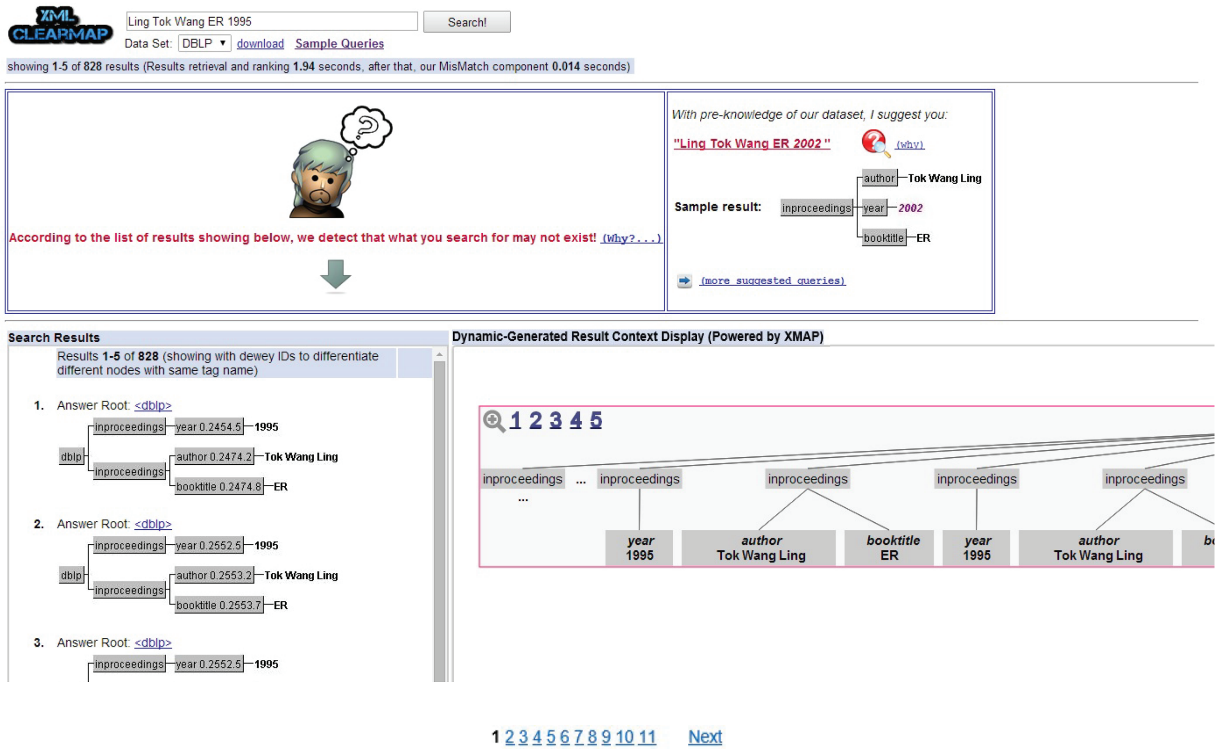

ults retrieval and ranking 0.026 seconds, after that, our MisMatch component 0.0 seconds)
Result Exploration Display (Powered by XMAP)
Zoom Out
dblp
inproceedings
inproceedings
key
conf/dmkd/JagadishN00
author
H. V. Jagadish
title
Incompleteness in Data Mining
pages
1-10
year
2000

Figure 10: Result Exploration Display of XML ClearMap

is being clicked by users. To zoom out, users can click the "zoom out" button on the top of the Result Exploration Display window. To close the Result Exploration Display window, users can click the close button on the top right corner of the Result Exploration Display windows.

For queries with the MisMatch problem, the XClear server will detect the problem and generate useful suggestion and return them to users. As shown in Figure 11, a box with suggestion for MisMatch problem will be shown under the search bar, which is the same as the XClear demo system. It includes hint, suggested queries, sample result, reasoning, etc. Since the suggestion is similar to the XClear demo system, we will not explain them here. Please refer to Appendix A for more detail.

The XML ClearMap embeds the XClear component and XMAP component to provide a complete experience of our research work enhancing the usability of XML keyword search. It can detect the MisMatch problem and give useful suggestion


Ling Tok Wang ER 1995 Search!

Data Set: DBLP download Sample Queries

showing 1-5 of 828 results (Results retrieval and ranking 1.94 seconds, after that, our MisMatch component 0.014 seconds)

According to the list of results showing below, we detect that what you search for may not exist! (Why?...)

With pre-knowledge of our dataset, I suggest you:

"Ling Tok Wang ER 2002" (why)

Sample result:

(more suggested queries)

Search Results

Results 1-5 of 828 (showing with dewey IDs to differentiate different nodes with same tag name)

Dynamic-Generated Result Context Display (Powered by XMAP)

Figure 11: XML ClearMap for Query with MisMatch Problem

to users. Meanwhile, it also provides an easy and interactive way for users to understand how the query results relate to each other and further explore the query results. It greatly enhance usability and move XML keyword search one step forward to be built as a user-friendly and industrialized product.